

Tangle Documentation

Show control, AV and building control on one engine.

Version 0.2 · 2026-10-27 · tangle.systems/docs

GET STARTED

Introduction

Install

Quick start

Concepts

Tutorial: a gallery installation

GUIDES

Designer

Devices

Device catalog

Timelines

Automations

Dashboards

Redundancy

Sentinel

Licensing

Simulator

Operations

End-user views

Listeners

Device creator

REFERENCE

REST API

WebSocket

MCP

Command line

Project file

EXTEND

Module SDK

Engine plugins

GET STARTED

Introduction

What Tangle is, who it is for, how its six parts fit together, and what each edition includes.

Tangle is a control system for shows, audiovisual installations, and rooms and buildings. One engine runs your project: it talks to projectors, media servers, lighting, audio processors, players and computers, plays timelines, runs automations and serves the dashboards people use to operate it all.

Who it is for

- **Show technicians and designers.** Timelines with timecode or cue lists, GO buttons, timecode chase and generation, redundant engines that take over within a second.
- **AV integrators.** One project per customer site: rooms, devices, scenes and touch panels, built in a visual designer and installed as a service on a rack machine.
- **Building and IT teams.** Time-of-day automations, monitoring with alerts by email, SMS, push, Slack or Teams, and an API for other systems.
- **Developers.** A REST API, a WebSocket protocol, an MCP server for AI assistants, and SDKs to write device modules in JavaScript, TypeScript or Python.

The six parts

PART	WHAT IT DOES
tangle-core	The engine. Keeps a library of projects and runs one, runs device modules, turns intents into device actions, plays timelines, runs automations, listeners and Sentinel. Serves the REST API (`/api/v1`), the WebSocket (`/ws`), MCP (`/mcp`), the designer (`/designer`) and dashboards (`/d/<id>`), all on port 7340.
tangle-connect	Device modules (over 100 drivers), the device creator and the `tangle` command. A module is a small program that speaks one device's protocol. `tangle` installs modules, adds and imports devices, tests connections and scans the network.
tangle-interact	Dashboards: a builder, a runtime for browsers, tablets and wall panels, and a generator that makes room, show and overview dashboards from the project. Two skins, Console and Lobby, and four control styles.

PART	WHAT IT DOES
tangle-designer	The visual editor for a project: rooms, devices, scenes, timelines, automations, variables, listeners, maps, dashboards, lighting and Sentinel, in one project tree. Runs in a browser at `/designer`, or as a desktop app for macOS, Windows and Linux that includes its own core.
tangle-sentinel	Monitoring inside core: watches devices, core nodes, network probes and pushed metrics, raises alerts and sends notifications. Pro and Ultimate editions.
tangle-simulator	Virtual devices (`tangle-sim`) that speak real protocols on real ports, so you can build and test a project without hardware.

How they fit together

From the top down:

1. **Clients**: the designer, dashboards on screens and panels, your scripts and AI assistants. They all talk to core over REST, WebSocket or MCP on port 7340.
2. **tangle-core**: holds the project and live device state, runs intents, timelines, automations and Sentinel.
3. **Device modules**: one process per module, started and supervised by core. Installed from the designer's Driver library or with the `tangle` command.
4. **Devices**: projectors, media servers, lighting, DSPs, players and computers, or tangle-simulator virtual devices while you build.

Everything is stored in one project file (JSON or YAML) that core loads from its project library. The designer, dashboards and API all change the project through core, which checks every change before applying it and writes it back to the file. Device modules run as separate processes, so a module that crashes is restarted without stopping the show. For redundancy, two or more cores run the same project and one takes over if the primary fails.

Read Concepts for the vocabulary: devices, capabilities, intents, scenes, timelines, automations and dashboards.

Editions

Tangle comes in three editions: Lite, Pro and Ultimate. The edition is set by your license and decided when core loads a project; a license change applies at the next start and never interrupts a running show.

	LITE	PRO	ULTIMATE
Controlled devices	20	100	Unlimited
DMX universes	2	Unlimited	Unlimited

	LITE	PRO	ULTIMATE
Timelines and cue lists (total)	2	Unlimited	Unlimited
Timecode chase and generation (LTC, MTC, Art-Net timecode)	No	Yes	Yes
PTP clock sync	No	Yes	Yes
Sentinel monitoring	No	Yes	Yes
External API (REST and MCP for third-party clients)	No	Yes	Yes
Redundancy	No	Add-on, per redundant host	Included, unlimited hosts
Designer, dashboards, scenes, automations	Yes	Yes	Yes

The built-in designer and dashboards always work, whatever the edition. DMX universes are counted as distinct `universe` values in device settings.

Without a license

Core runs without a license in one of two modes:

- **Free designer** (default). Program a project on one computer, with no limit on devices, universes or timelines. Core serves this computer only, the external API and Sentinel are off, and device connections and timelines pause at the end of every hour until you resume.
- **Demo** (`tangle-core run --demo`). Every feature except redundancy, up to 10 devices, other machines can connect. Sessions last one hour.

See Licensing to activate a license, move one, or activate a computer that has no internet.

Where to go next

- Install: download the desktop app, the core service, the CLI and modules.
- Quick start: build a first project with a simulated projector in about 15 minutes.
- Tutorial: a gallery installation: a complete project from scratch, with the simulator, automations and a staff dashboard.
- Concepts: the words used across Tangle.
- Designer: the visual editor, section by section.
- Devices and the device catalog: add and configure devices.
- Timelines, automations and listeners: shows, building logic and triggers from other systems.
- Device creator: drivers for devices Tangle does not know yet.

- Dashboards: control panels for operators, rooms and walls.
- Redundancy, Sentinel and operations: run Tangle in production.
- REST API, WebSocket, MCP, CLI and module SDK: integrate and extend.

GET STARTED

Install

Download and install the Tangle desktop app, the core service, the tangle CLI, device modules and the simulator on macOS, Windows or Linux.

Tangle ships as separate pieces so each machine gets only what it needs: an operator laptop runs the all-in-one desktop app, a rack server runs only the core service, a booth PC only needs the `tangle` command. This page lists the downloads, what each machine needs, and where Tangle keeps its files.

What you can download

Each release publishes the files below with a `SHA256SUMS` file to check them. `<v>` is the version and `<target>` the platform, for example `aarch64-apple-darwin`.

DOWNLOAD	WHAT IT IS	PLATFORMS
Desktop app (`.dmg`, `.msi` or `.exe`, `.AppImage`, `.deb` or `.rpm`)	The designer with its own core, the web UIs, the official device modules and a Node.js runtime. The macOS `.dmg` also holds `tangle-sim`.	macOS (Apple silicon, Intel), Windows x64, Linux (x64, arm64)
`tangle-core-<v>-<target>` (`.tar.gz`, `.zip` on Windows)	The core service and its web UI (designer and dashboards)	Same as above
`tangle-cli-<v>-<target>` (`.tar.gz`, `.zip` on Windows)	The `tangle` command	Same as above
`modules/tangle-module-<id>-<v>.tar.gz`	One device module. Works on any OS.	Any
`simulator/tangle-simulator-<v>.tar.gz`	The simulator as a script, for any OS with Node.js 20 or newer	Any
`simulator/tangle-sim-<v>-<target>`	The simulator as one executable with Node.js built in	Per platform

Which pieces you need

MACHINE	INSTALL
Laptop to design and run a project	Desktop app

MACHINE	INSTALL
Show or building server	Core service, CLI, and the module packages your devices use
Second core for redundancy	Same as the server
A computer that only manages modules or device lists	CLI
Testing without hardware	Simulator

System requirements

Tangle runs on macOS, Windows and Linux. What else a machine needs depends on the device modules it runs:

MODULE RUNTIME	NEEDS	NOTES
Node.js (most modules)	Node.js 20 or newer; some modules need 22	The desktop app includes Node.js 22. The core service uses `node` from the `PATH`, or the program set in `TANGLE_NODE`.
Python (for example `misc.ssh.computer`, `microsoft.powershell.computer`, `microsoft.psexec.computer`)	Python 3.11 or newer	Not bundled anywhere, including the desktop app. Core runs `python3` (`python` on Windows). Set `TANGLE_PYTHON` to use another interpreter, such as a virtualenv.
Native	Nothing	The module is an executable.

When a module's runtime is missing or too old, the module does not start and the error names the program and the variable to set.

To check which modules a core can see:

```sh
tangle-core modules
```

Desktop app

1. Download the installer for your platform and open it.
   - **macOS**: open the `.dmg` and drag Tangle to Applications. The disk image also holds `tangle-sim`; copy it to a folder if you want the simulator.
   - **Windows**: run the `.msi` or `.exe`.

- **Linux**: install the `.deb` or `.rpm`, or run the `.AppImage`.

2. Open Tangle. It starts its own core on `127.0.0.1` on a free port, creates a token for itself and signs the designer in. You land on the project that was open last.

The app keeps its core's data in `core/` in the app's data folder: the project library (`core/projects/<project id>/project.json`), modules, tokens and logs. Projects are opened, created, imported and exported from the designer's project menu.

The app's data folder:

MACOS	WINDOWS	LINUX
~/Library/Application Support/io.tangle.designer	%APPDATA%\io.tangle.designer	~/.local/share/io.tangle.designer

The app's core stops when you quit the app. It does not conflict with a core service on the same machine: it uses its own port and data folder.

To work on another core instead (a server on the network), sign out and enter that core's address and a token. The sign-in screen offers **Use the core on this computer** to go back. See Designer.

> **Note:** Until you activate a license in **Settings > License**, the app's core runs as the free designer: this computer only, and device connections and timelines pause every hour. To run a show for other machines, install the core service with a license.

Core service

1. Extract `tangle-core—<v>-<target>`. Keep the `ui` folder next to the `tangle-core` program.
2. Start it:

```sh
./tangle-core run                          # the project open last, from the library, or a new one
./tangle-core run --project my-show.json   # or a project file of your own, edited in place
```

Without `--project`, core keeps its projects in a library in its data folder (`tangle-core projects list`), and the designer creates, opens and imports them. With `--project`, it runs that file. A project file needs at least an id and a name:

```json
{ "id": "my-show", "name": "My show" }
```

3. To install drivers from the designer, put the module packages in a folder named `modules` next to `tangle-core` (one sub-folder per module), or point `--catalog` at another folder. Without it, install modules with

`tangle module add`.

4. Open `http://127.0.0.1:7340/designer`. Create a token to sign in:

```sh
./tangle-core token create --name designer --scopes read,control,configure
```

Accept other machines

By default core listens on `127.0.0.1:7340` and only this computer can connect. To let dashboards on tablets and other machines connect:

```sh
./tangle-core run --project my-show.json --listen 0.0.0.0:7340 --allow-remote --insecure-http
```

> **Warning:** Traffic is not encrypted (no TLS yet). Use this on trusted show and control networks only.

Install as a service

The service starts at boot, restarts after a crash and stops cleanly. Arguments after `--` are passed to `run`.

macOS / Linux:

```sh
sudo ./tangle-core service install -- --project /shows/main.json --listen 0.0.0.0:7340 --
allow-remote --insecure-http
```

Windows (administrator PowerShell):

```sh
.\tangle-core.exe service install -- --project C:\Shows\main.json --listen 0.0.0.0:7340 --
allow-remote --insecure-http
```

It uses launchd on macOS, systemd on Linux and the Windows Service manager on Windows. Paths become absolute, and `PATH` and `TANGLE_*` variables are kept so `node` and `python3` are found. On macOS and Linux, `--user` installs it for the current user, starting at login instead of boot.

To remove it:

```sh
sudo tangle-core service uninstall
```

A service runs as root on macOS and Linux and uses the machine-wide folders (see file locations). Manage its tokens and modules with `sudo`, for example `sudo tangle-core token create ...`.

For redundant cores, see Redundancy.

The tangle CLI

Extract `tangle-cli-<v>-<target>` and put `tangle` (`tangle.exe` on Windows) somewhere on your `PATH`, for example `/usr/local/bin`.

```sh
tangle module list
tangle discover
```

Run it with `sudo` (an administrator shell on Windows) to manage the modules of a core installed as a service. See CLI.

Device modules

The desktop app includes the official modules. On a core service, install the modules your devices use, one package each:

```sh
tar -xzf tangle-module-misc.pjlink.projector-1.0.0.tar.gz
sudo tangle module add ./misc.pjlink.projector
```

Installing needs no internet and no `npm install` or `pip install`. `tangle module add` asks you to grant the permissions the module requests (network, serial, USB, MIDI, file system); `--yes` grants them without asking. Module ids look like `<manufacturer or misc>.<product or protocol>.<category>`, for example `misc.pjlink.projector`. See the device catalog.

Simulator

The simulator runs virtual devices on this machine. Pick one form:

```sh
# Executable (Node.js built in)
./tangle-sim run examples/demo.json

# Script package (Node.js 20 or newer)
node tangle-sim.mjs run examples/demo.json
```

On macOS, if you copied `tangle-sim` from the `.dmg`, write the example configuration first:

```sh
./tangle-sim init
./tangle-sim run
```

The web UI opens at `http://127.0.0.1:4380`. See Simulator.

File locations

Core and the CLI share these folders. `TANGLE_HOME` moves all of them into one folder.

	MACOS	WINDOWS	LINUX
Data (installed modules, `tokens.json`, dashboard versions)	~/Library/Application Support/Tangle	%ProgramData%\Tangle	~/.local/share/tangle
Installed modules	~/Library/Application Support/Tangle/modules	%ProgramData%\Tangle\modules	~/.local/share/tangle/modules
Logs and `audit.log`	~/Library/Logs/Tangle	%ProgramData%\Tangle\logs	~/.local/state/tangle/logs

Run as root (a service on macOS or Linux), core uses machine-wide folders instead:

	MACOS	LINUX
Data	/Library/Application Support/Tangle	/var/lib/tangle
Logs	/Library/Logs/Tangle	/var/log/tangle

Device passwords and keys are not stored in the project file. They go to the OS credential store (Keychain, Credential Manager, Secret Service). Where no store answers, such as a Linux service without a desktop session, they go to `secrets.json` in the data folder, readable by its owner only. `TANGLE_SECRETS=file` forces that.

Environment variables

VARIABLE	EFFECT
`TANGLE_HOME`	One folder for data, modules, tokens and logs
`TANGLE_NODE`	Node.js program used for JavaScript modules
`TANGLE_PYTHON`	Python program used for Python modules (for example a virtualenv's `python`)
`TANGLE_UI`	Folder of the web UI, instead of `ui` next to `tangle-core`
`TANGLE_LOG`	Log level, for example `debug`

VARIABLE	EFFECT
`TANGLE_SECRETS`	`file` stores device secrets in `secrets.json`
`TANGLE_CLUSTER_KEY`	Shared secret for redundant cores

Licensing

Without a license, core runs as the free designer (this computer only, pauses every hour) or, with `--demo`, in demo mode (10 devices, one-hour sessions). Activate a license on each machine that runs a show:

```sh
tangle-core license activate <key> --name stage-a
tangle-core license status
```

See Licensing for editions, offline activation and moving a license.

Ports to open

PORT	PROTOCOL	USED FOR	OPEN BETWEEN
7340	TCP	REST, WebSocket, MCP, designer and dashboards	Clients (designer, panels, scripts) and core
7341	UDP and TCP	Redundancy heartbeats and state mirroring (change with `--cluster-listen`)	All core nodes of a cluster
7342	UDP and TCP	Core discovery (multicast beacons) and pairing	Cores on the same local network that you want to pair
4380	TCP	Simulator web UI and control API (listens on 127.0.0.1 unless `--ui-host`)	Local only
Device ports	Per protocol	Each module connects to its device, for example PJLink on TCP 4352	Core and devices

Network discovery (`tangle discover` and the designer's scan) uses mDNS, SSDP (UDP 1900) and searches done by modules, so multicast must reach the core's network. On macOS, allow incoming connections when asked if other machines need to reach core or the simulator.

GET STARTED

Quick start

Build a first Tangle project with a simulated PJLink projector, a scene and a dashboard, without any hardware.

This walkthrough builds a small working project in about 15 minutes: a room with a simulated projector, a scene, and a dashboard you can open on a phone or tablet. You need no hardware: the Tangle simulator plays the projector, speaking real PJLink on a real network port.

You will:

1. Start the simulator with its demo devices.
2. Start core, with the desktop app or the core service.
3. Open the designer and sign in.
4. Add a room and a PJLink projector, and test the connection.
5. Make a scene.
6. Make a dashboard, publish it and open it.

Before you start

Download these from the release (see Install):

- The **desktop app** for your platform, or the **core service** archive, the **CLI** archive and the `misc.pjlink.projector` module package.
- The **simulator**. On macOS it is inside the desktop app's `.dmg` (`tangle-sim`).

---

1. Start the simulator

Open a terminal in the folder where you put the simulator.

If you have the `tangle-sim` executable (for example copied from the macOS `.dmg`):

```sh
./tangle-sim init        # writes tangle-sim.json: two projectors, a video matrix, a DMX receiver
./tangle-sim run
```

If you have the script package (needs Node.js 20 or newer):

```sh
node tangle-sim.mjs run examples/demo.json
```

The simulator prints the port of each virtual device and opens its web UI at `http://127.0.0.1:4380`. The demo starts:

DEVICE	TYPE	PORT	NOTES
Hall projector (`pj-1`)	PJLink	TCP 4352	No password, 5 s warm-up and cool-down
Stage projector (`pj-2`)	PJLink	TCP 4353	Password `secret`
Video matrix 4x2	ASCII over TCP	5023	
Dimmer rack	Art-Net and sACN receiver	UDP 6454 / 5568	

Leave the simulator running. Keep its web UI open in a browser tab: you will watch the projector react there.

> **Note:** macOS asks once whether `node` (or `tangle-sim`) may accept incoming connections. Allow it if core runs on another machine.

---

2. Start core

Choose one option.

Option A: the desktop app

Open Tangle. It starts its own core and signs in. On a first start, the new empty project opens on **Start a project**: click **Start empty**. Skip to step 4.

Option B: the core service

1. Extract the core and CLI archives, and put `tangle` on your `PATH`.
2. Install the PJLink module:

```sh
tar -xzf tangle-module-misc.pjlink.projector-1.0.0.tar.gz
tangle module add ./misc.pjlink.projector --yes
```

3. Create a project file named `quick-start.json`:

```JSON
{ "id": "quick-start", "name": "Quick start" }
```

4. Start core from the folder where you extracted it:

```SH
./tangle-core run --project quick-start.json
```

Core listens on `http://127.0.0.1:7340`. Leave it running and open a second terminal for the next step.

---

3. Open the designer and sign in

With the desktop app you are already signed in. With the core service:

1. Create a token that can edit the project:

```SH
./tangle-core token create --name designer --scopes read,control,configure
```

The token is printed once. Copy it.

2. Open `http://127.0.0.1:7340/designer` in a browser.
3. Paste the token in **API token** and click **Sign in**.

The designer opens on the **Overview**: counts of rooms, devices, scenes, timelines, automations and dashboards, and a **Needs attention** list. The project tree on the left holds everything in the project.

---

4. Add a room and a projector

Add a room

1. In the project tree, right-click **Rooms and devices** and choose **New room...**.
2. Type `Hall` and click **Add**.

Add the projector

1. Right-click **Hall** in the project tree and choose **Add device**.
2. Fill in the form:

FIELD	VALUE
Name	`Hall projector` (the id `hall-projector` is filled in for you)

FIELD	VALUE
Driver	PJLink projectors (`misc.pjlink.projector`). If it is not in the list, click **Driver library: find, install or share drivers...**, search for `PJLink`, click **Use**, then **Install**.
Room	Hall
host	`127.0.0.1` (or the simulator machine's IP address)
port	`4352`

The **Settings** part of the form comes from the driver: PJLink asks for `host`, `port` and an optional `password`.

3. Click **Create device**. The PJLink setup assistant opens ("Turn on network control"): it lists what to switch on in a real projector. The simulator needs none of it, so click **Finish later**.

Within a few seconds the status next to the device name turns **online**. The device page shows its live **Status**, ready-made **Controls** (press **Power** to try it) and its **Log**.

Test the connection

Click **Test connection**. Core starts the module on its own with the settings in the form (without saving them) and reports, for example, `Connected (online, 35 ms)`. If it says `Not connected`, check that the simulator is running and that host and port match.

> **Note:** To try a failure, run `./tangle-sim fault pj-1 offline` in the simulator folder (or switch on **offline** in its web UI). The device turns **offline** in the designer. `./tangle-sim clear pj-1` brings it back.

Optional: find it on the network

Click **All devices** in the project tree, then **Scan the network**. After 5 seconds the list shows what answered, including the simulated projectors (PJLink class 2 search). **Add** opens the device form with the host filled in and, when a module matches, the module chosen (**Add with PJLink projectors**).

---

5. Make a scene

A scene is a set of device actions recalled together.

1. Click **Scenes** in the project tree, then **New scene**.
2. Name it `Presentation`.
3. Click **Add action**. Choose **Device**: Hall projector, **Action**: `power on`.
4. Click **Add action** again. Choose Hall projector, `input select`, and type `31` in **input** (HDMI 1 on PJLink).
5. Click **Create scene**.

Make a second scene named `Off` with one action: Hall projector, `power off`.

---

6. Make a dashboard

1. Click **Dashboards** in the project tree.
2. In **Generate**, choose the room **Hall** and click **Generate room dashboard**.

The builder opens with a page already laid out: a title, a **Scenes** list with Presentation and Off, and the projector's controls. Generated room dashboards use the Lobby skin.

To place a device yourself instead: drag **Hall projector** from the **Devices** list in the palette onto the page. The builder asks which layout to use (Compact, Operator or Full for PJLink) and which controls to show, then places them as a group bound to the projector.

3. Click **Preview** to see the dashboard as screens will. Presses in preview are only logged. Tick **Live control** to send them to core.
4. Click **Publish**. The builder confirms "Published version 1".

Open the dashboard

Click ← to go back to the dashboard list, then **Open** next to your dashboard. It opens at `/d/room-hall` on your core: `http://127.0.0.1:7340/d/room-hall` with the core service. The desktop app's core uses another port, which the **Open** link already includes.

Try it:

- Press **Presentation**. Watch the simulator's web UI: the Hall projector warms up for 5 seconds, then turns on with input 31.
- Press the **Power** toggle. It asks for a second press ("Press again to confirm") because the module marks power as a confirmed action.
- Press **Off** to turn the projector off.

To open the dashboard on a phone or tablet, core must accept other machines: start it with `--listen 0.0.0.0:7340 --allow-remote --insecure-http` and a license or `--demo`, then open `http://<core IP>:7340/d/room-hall`.

> **Note:** Without a license, core runs as the free designer: it serves this computer only, and at the end of each hour device connections and timelines pause until you resume. Start core with `--demo` to try other machines (up to 10 devices, one-hour sessions). See Licensing.

Where the project lives

Every change is saved to the project file as you make it: `quick-start.json` for the core service, or the project in the app's project library for the desktop app (**Export…** in the project menu makes a `.tangle` copy). Open it to see the room, device, scenes and dashboard you made. The PJLink password, if you set one, is not in the file: it goes to the OS credential store.

Next steps

- Follow Tutorial: a gallery installation for a complete project: several devices, automations that start and stop a room on a schedule, and a staff dashboard.
- Add the Stage projector (`127.0.0.1`, port `4353`, password `secret`) and start the simulator scenario `stage-pj-dies` to see a device fail and recover.
- Build a show in Timelines.
- React to events, the clock and buttons with Automations.
- Learn the vocabulary in Concepts.
- Customize the dashboard: pages, screen sizes, PINs and wall panels in Dashboards.

GET STARTED

Concepts

The building blocks of a Tangle project - rooms, devices, modules, capabilities, state, intents, scenes, timelines, automations, variables, listeners, dashboards and core nodes.

This page explains the words used across Tangle and how the pieces relate. Every part of Tangle (core, modules, dashboards, the designer, the API) shares one set of definitions, the Tangle contract, written as JSON Schema. The current contract version is 0.2.43.

Project

A project is everything core needs to run one site or one show, in one file (JSON or YAML). It holds:

SECTION	CONTENTS
`id`, `name`	Required. The id is used in folder names and dashboard storage.
`rooms`	Places that group devices
`devices`	Configured devices
`scenes`	Sets of device actions recalled together
`timelines`	Shows: tracks of timed cues, or cue lists
`automations`	Step lists run by triggers, buttons, listeners or by hand
`variables`	Values core keeps for the whole project
`listeners`	Ports other systems send their own messages to
`dashboards`	Published control surfaces
`maps`, `views`	Floor plans with devices, and the menus end users open
`lighting`	Lighting engine: groups, presets, sequences, executors
`cluster`, `units`, `controls`	How a cluster shares the work, how values are shown, the default control style

SECTION	CONTENTS
`alertRules`, `notificationChannels`, `maintenanceWindows`, `probes`	Sentinel monitoring

The smallest valid project:

```json
{ "id": "my-show", "name": "My show" }
```

Core keeps a library of projects on its computer (**Open or new project…** in the designer, `tangle-core projects` on the command line) and runs one at a time. It loads the project at start, checks it against the schema and checks references (a scene that names a missing device is an error). Changes made through the designer, dashboards, REST or MCP are checked the same way, applied live, written back to the file and recorded in `audit.log`. Passwords and keys never go in the file. See Project file.

Identifiers

Ids of rooms, devices, scenes, timelines, cues, dashboards and modules are lowercase: letters, digits, `.`, `_` and `-`, starting with a letter or digit, up to 128 characters. `room-a.projector-1` is valid; `Room A` is not. Times are integer milliseconds since the Unix epoch (UTC).

Rooms

A room groups devices by place: a meeting room, a gallery, the stage. A room has an `id`, a `name` and optional `tags`. A device belongs to at most one room. Rooms are what you target to control a whole space ("turn the Hall off"), and what room dashboards are generated from.

Devices

A device is one controlled thing in the project: a projector, a media server, a DSP, a lighting device, a computer. A device is one module plus one configuration:

```json
{
  "id": "hall-projector",
  "name": "Hall projector",
  "moduleId": "misc.pjlink.projector",
  "room": "hall",
  "tags": ["projectors"],
  "config": { "host": "10.0.0.21", "port": 4352 }
}
```

FIELD	MEANING
`moduleId`	The module that drives it
`config`	Settings, checked against the module's settings schema. Secret settings (passwords) are stored in the OS credential store, not here.
`room`	Optional room id
`tags`	Optional labels, used as intent targets (`projectors`, `stage-left`)
`folder`	Optional folder in the device list (`Stage/Truss 1`)
`planned`	Optional. A device not on site yet: kept in the project, not started
`node`	With redundancy: the core node the device's hardware is attached to (serial port, USB DMX, LTC output). Omit for networked devices.

Status

Every device reports a connection status:

STATUS	MEANING
`connecting`	First connection in progress
`online`	Connected and answering
`degraded`	Answering, but reporting a problem
`offline`	Unreachable, refused or disconnected
`timeout`	Reachable or connected, but not answering in time
`error`	Needs the operator, for example a bad password or a missing setting

See Devices.

Modules

A device module is the driver for a kind of device. It runs as its own process, started and supervised by core, and talks to core over a simple JSON protocol on standard input and output. A crash affects only that module: core restarts it with a backoff. Core also limits every action to 15 seconds.

Module ids follow `<manufacturer or misc>.<product or protocol>.<category>[.<version>]`, for example `misc.pjlink.projector`, `qsc.core.dsp` or `dataton.watchout.mediaserver.v7`. Categories include timecode, projector, display, mediaserver, engine, player, computer, io, dsp, lighting, console, switcher, control, generic, gateway, network, microphone, camera, codec, presentation, avoip, routing, amplifier, building, power and scheduling, plus `listener` and `library` for modules that open listeners or add functions.

Each module ships a manifest (`tangle-module.json`) that declares:

- the runtime: Node.js, Python or a native executable;
- the capabilities its devices support, standard and custom;
- the settings schema, which the designer turns into a form;
- the permissions it needs (network, serial, USB, MIDI, file system);
- optionally, network discovery, ready-made dashboard controls and layouts, setup assistants for the device side, documentation links and several connections (LAN and RS-232, say).

Modules are installed from the designer's **Driver library**, or with `tangle module add`, and written with the module SDK or, without code, with the device creator. See the device catalog for the official modules.

Capabilities

A capability is a reusable set of state keys and actions. Capabilities let core, scenes and dashboards control any device the same way, whatever its protocol: every device with `tangle.power` turns on with the same action.

Standard capabilities:

CAPABILITY	STATE	ACTIONS
`tangle.power`	`on` (boolean)	`on`, `off`, `toggle`
`tangle.level`	`level` (0 to 1)	`set` (`level`, optional `fadeMs`)
`tangle.mute`	`muted` (boolean)	`mute`, `unmute`, `toggle`
`tangle.input`	`input` (string)	`select` (`input`)
`tangle.transport`	`status` (stopped, playing, paused), `positionMs`, `clip`	`play`, `pause`, `stop`, `seek` (`positionMs`), `load` (`clip`)
`tangle.preset`	`current` (string)	`recall` (`preset`)

CAPABILITY	STATE	ACTIONS
`tangle.timecode`	`timecode`, `fps`, `running`, `signal`	`start`, `stop`, `locate` (`timecode`)
`tangle.playhead`	`timecode`, `positionMs`, `durationMs`, `running`, `rate`, `fps`, `playheads`	none: where a media player or server is in time, one playhead per timeline for media servers. Timelines can chase it.
`tangle.preview`	`streamUrl`, `snapshotUrl`, `available`	none: a picture of what the device shows, for Camera controls
`tangle.setup`	`needed`, `configurator`, `reason`	none: the device answers but is not set up for Tangle yet; the designer offers its setup assistant

A module can add custom capabilities under its own prefix. The PJLink module adds `pjlink.lamp` (lamp hours), `pjlink.status` (power including warming and cooling) and `pjlink.health` (fan, lamp, temperature, cover, filter errors).

State values and action parameters are typed: boolean, number (with optional min, max, step and unit), string, enum or JSON. The designer uses these types to build its pickers, for example a number field limited to 0 to 1 for `tangle.level` `set`.

State

State is what a device reports right now, per capability: `tangle.power` `on: true`, `tangle.level` `level: 0.8`, `pjlink.lamp` `hours: 1790`. Modules push state as it changes or as they poll. Core keeps the latest state of every device and sends changes to clients. Dashboards display state through bindings; automations trigger on it and test it in conditions.

Actions

An action is one concrete command for one device: device, capability, action name and parameters.

```JSON
{ "deviceId": "hall-projector", "capability": "tangle.input", "action": "select", "params": { "input": "31" } }
```

Scenes are lists of actions. Timeline cues and automations can run an action directly or send an intent.

Events

Core publishes events when something changes. Each event has a topic `<type>:<subject id>`, and clients subscribe to the topics they need (wildcards such as `device.state:*` are allowed).

EVENT	TOPIC	WHEN
`device.state`	`device.state:<deviceId>`	A state value changed
`device.status`	`device.status:<deviceId>`	A device's status changed
`device.event`	`device.event:<deviceId>`	A device reported something, for example a button press or an alarm
`intent.status`	`intent.status:<intentId>`	An intent progressed
`timeline.state`	`timeline.state:<timelineId>`	A timeline played, paused, moved or reached a cue
`node.heartbeat`	`node.heartbeat:<nodeId>`	A core node reported its health
`project.changed`	`project.changed:<projectId>`	The project was changed (with a revision number and the list of changes)
`module.status`, `module.health`	`module.status:<moduleId>`	A module started, crashed or restarted; resource use and latency every 10 s
`device.action`	`device.action:<deviceId>`	An action was sent, is in progress, done or failed
`device.log`	`device.log:<deviceId>`	A device log entry was added
`automation.run`	`automation.run:<automationId>`	An automation run started or ended
`variable.changed`	`variable.changed:<name>`	A project variable changed
`control.changed`, `control.event`	`control.changed:<controlId>`, `control.event:<controlId>`	A dashboard control's value or properties changed, or it was clicked, pressed, released or held
`listener.message`	`listener.message:<listenerId>`	A listener received a message
`dashboard.draft`	`dashboard.draft:<dashboardId>`	A dashboard draft was saved (other designer windows refresh)
`sentinel.alert`	`sentinel.alert:<ruleId>`	An alert changed or a notification was sent

EVENT	TOPIC	WHEN
`license.status`	`license.status:core`	The license changed, or a session paused or resumed

See WebSocket.

Intents

An intent says what should happen, not how: "power on the Hall", "recall Presentation in room A". Core resolves the target to devices, turns the intent into actions for the devices that support it, and reports the outcome. Dashboards, timelines, automations and API clients all control devices through intents.

```json
{
  "name": "scene.recall",
  "target": { "kind": "room", "id": "hall" },
  "params": { "scene": "presentation" },
  "priority": 30
}
```

Intent names

NAME	DOES
`<capability>.<action>`	A standard capability's action on every targeted device that has it: `power.on`, `power.off`, `level.set`, `input.select`, `transport.play`, `preset.recall`. `mute.on` and `mute.off` are short for `mute.mute` and `mute.unmute`.
`scene.recall`	The actions of the scene in `params.scene`, for the targeted devices only
`action`	One raw action: `params.capability`, `params.action`, `params.params`
`release`	Drops priority holds on the targeted devices

Devices without the capability are skipped and reported, not treated as errors.

Target kinds

KIND	EXAMPLE	COVERS
`device`	`{ "kind": "device", "id": "hall-projector" }`	One device
`room`	`{ "kind": "room", "id": "hall" }`	Every device in the room

KIND	EXAMPLE	COVERS
`tag`	`{ "kind": "tag", "tag": "projectors" }`	Every device with the tag
`all`	`{ "kind": "all" }`	Every device in the project

Priority and holds

Each intent has a priority from 0 to 100. When two intents act on the same device state and overlap, the higher priority wins and the other reports the device as `overridden`. Defaults depend on the source: `system` 90, `timeline`, `schedule` and `automation` 50, everything else (users, dashboards, API) 30.

- The `holdMs` parameter keeps the intent's priority on the state it set for that long.
- An intent with priority 90 or more holds the state until a `release` intent.

An intent can also carry `at` (run at a time) and `expiresAt`.

Intent status

An intent goes through `accepted`, `scheduled`, `running`, then ends `completed`, `partial`, `failed`, `rejected` or `cancelled`. Each device gets an outcome: `ok`, `failed`, `overridden` or `skipped`. A room intent where one of three devices is offline ends `partial` and names the offline device.

Scenes

A scene is a named set of device actions recalled together: "Presentation" turns the projector on, selects HDMI 1 and dims the lights.

```JSON
{
  "id": "presentation",
  "name": "Presentation",
  "actions": [
    { "deviceId": "hall-projector", "capability": "tangle.power", "action": "on" },
    { "deviceId": "hall-projector", "capability": "tangle.input", "action": "select",
"params": { "input": "31" } }
  ]
}
```

Recall a scene with the `scene.recall` intent. Only the actions for devices inside the intent's target run, so one scene can be recalled for a whole site or for one room.

Timelines

A timeline is a show: tracks of cues, each cue at a position in milliseconds. A cue sends an intent or runs a device action, can fire early by a pre-roll to absorb device latency, and can be disabled.

SETTING	VALUES
`mode`	`timecode`: runs freely from play. `cuelist`: waits for GO at each cue.
`clock`	`internal`, `timecode` to chase a timecode input device (LTC, MTC, Art-Net timecode), `wallclock` to follow the time of day, or `ptp` to run at a PTP grandmaster's rate
`frameRate`	24, 25, 29.97df, 30, 50, 60
`timecodeOut puts`	Timecode output devices that follow the timeline
`durationMs`, `loop`	Length and looping

Transport operations are `play`, `pause`, `stop`, `seek` and `go`. A timeline's status is `stopped`, `playing`, `paused` or `waiting` (a cue list waiting for GO, or a chasing timeline waiting for timecode).

> **Note:** The older `ltc` and `mtc` clock forms are not played: use a `timecode` clock with an input device.

See Timelines.

Automations

An automation is a named list of **steps**, run when one of its **triggers** fires and its **conditions** hold, or whenever something runs it: a dashboard button, a listener, another automation, the API.

- **Triggers** (any one starts a run): time of day, device state, device event, device status, timeline (started, stopped, cue reached), project variable, dashboard control, listener message, core start, only by hand.
- **Conditions** (all must hold): device state, device status, time window, timeline status, variable, dashboard control.
- **Steps** (in order): intent, device action (optionally waiting until the device reports it is done), timeline, wait, wait for a condition, if, repeat, while, set a variable, set a control, run another automation, call a function, stop, comment, log.
- **Params** pass values in (a slider's level), and **mode** decides what a trigger does while a run is in progress: `single` (ignore), `restart`, `queued` or `parallel`.

Automations run on the primary core only. Macros, from older projects, are automations now. See Automations.

Variables and expressions

A project variable is a value core keeps for the whole project, such as `houseMode = "show"` or `visitors = 12`. Automations, listeners, dashboards and the API read and set it, and automations can start when it changes. Steps, bindings and labels read variables and params as `{{name}}`, and compute values with expressions such as `clamp({{level}} * 1.2, 0, 100)`. See Automations.

Listeners

A listener opens a port (WebSocket, HTTP, TCP, UDP, OSC) or a serial line for other systems, which send their own messages. Rules match each message and set variables, run steps and automations, and answer. See Listeners.

Dashboards

A dashboard is a control surface: pages of controls (buttons, toggles, sliders, keypads, scene lists, status, transport, GO, cameras, maps…) laid out on a grid per screen size. Controls read device state, variables, other controls or expressions through bindings, and send intents, timeline commands, run automations or set variables. Each dashboard has a skin (Console or Lobby), a control style (Classic, Studio, Skeuomorphism or Relief), a theme (light, dark or auto) and an access mode (anyone, PIN, or enrolled panels). Screens open it at `/d/<id>`. See Dashboards.

Core nodes and clusters

A core node is one running `tangle-core`. Each node has an id (default `core-1`) and a role:

ROLE	MEANING
`standalone`	A single core, no redundancy
`primary`	In a cluster, the node that drives devices, runs automations and Sentinel
`backup`	In a cluster, computes the same show in lockstep, ready to take over

For redundancy, two or more nodes run the same project and exchange heartbeats over port 7341. Every node plays timelines on a shared cluster clock; only the primary drives devices. If the primary stops, the next node takes over within a second at the same point in the show. In shared mode, devices are spread across the healthy nodes instead, while the primary still runs timelines and automations. Each failover increases an epoch counter, and the highest epoch wins, so two nodes never both stay primary. Backups compare a log of the cues they

computed with the primary's and resync if they differ. With three or more nodes, a node is primary only while it reaches a majority.

See Redundancy.

GET STARTED

Tutorial: a gallery installation

Build a complete project from scratch with the simulator - two projectors and a media player, automations that start and stop the room on a schedule, and a dashboard for the staff.

This tutorial builds a real project from an empty page: a small gallery with two projectors and a media player, which starts by itself at 10:00, stops at 18:00, and has a touch panel for the staff. It takes about 45 minutes and needs no hardware: the Tangle simulator plays the three devices, speaking their real protocols on real network ports.

Quick start shows the parts of Tangle in 15 minutes. This tutorial goes further and explains why each step is done the way it is, so you can plan your own installations the same way.

The installation

The gallery has:

DEVICE	WHAT IT DOES	IN THE SIMULATOR
Left projector	PJLink projector on the left wall	`pjlink`, TCP 4352
Right projector	PJLink projector on the right wall	`pjlink`, TCP 4353
Media player	VLC playing a playlist of three videos (Intro, Loop, Credits)	`videolan.vlc.player`, TCP 8080, password `secret`

What the client asked for:

- Every day at 10:00 the projectors turn on, and the video starts once they are warm.
- At 18:00 the video stops and the projectors turn off.
- The staff have a tablet with **Start** and **Stop** buttons, and the controls of each device for the odd problem.
- A wrong press on **Stop** must not close the room in front of visitors.

You will build it in this order, which works for any project:

1. **Devices first.** Put each device in a room, connect it and check its status. Everything else refers to devices.
2. **Automations next.** Write what the room does (start, stop) once, as automations. Schedules and buttons then call the same automation, so the room always starts the same way.

3. **The dashboard last.** Its buttons run the automations, and its device controls are generated from the drivers.

---

1. Start the simulator

The simulator comes with the release (see Simulator): the `tangle-sim` executable (on macOS, in the Tangle `.dmg` next to Tangle.app) or the `tangle-simulator` package for Node.js 20 or later.

Make an empty folder for the tutorial, open a terminal in it and start the simulator with no configuration:

```sh
./tangle-sim run                    # or: node tangle-sim.mjs run
```

It starts with no devices, creates `tangle-sim.json` in the folder and opens its web UI at `http://127.0.0.1:4380`.

Add the three devices

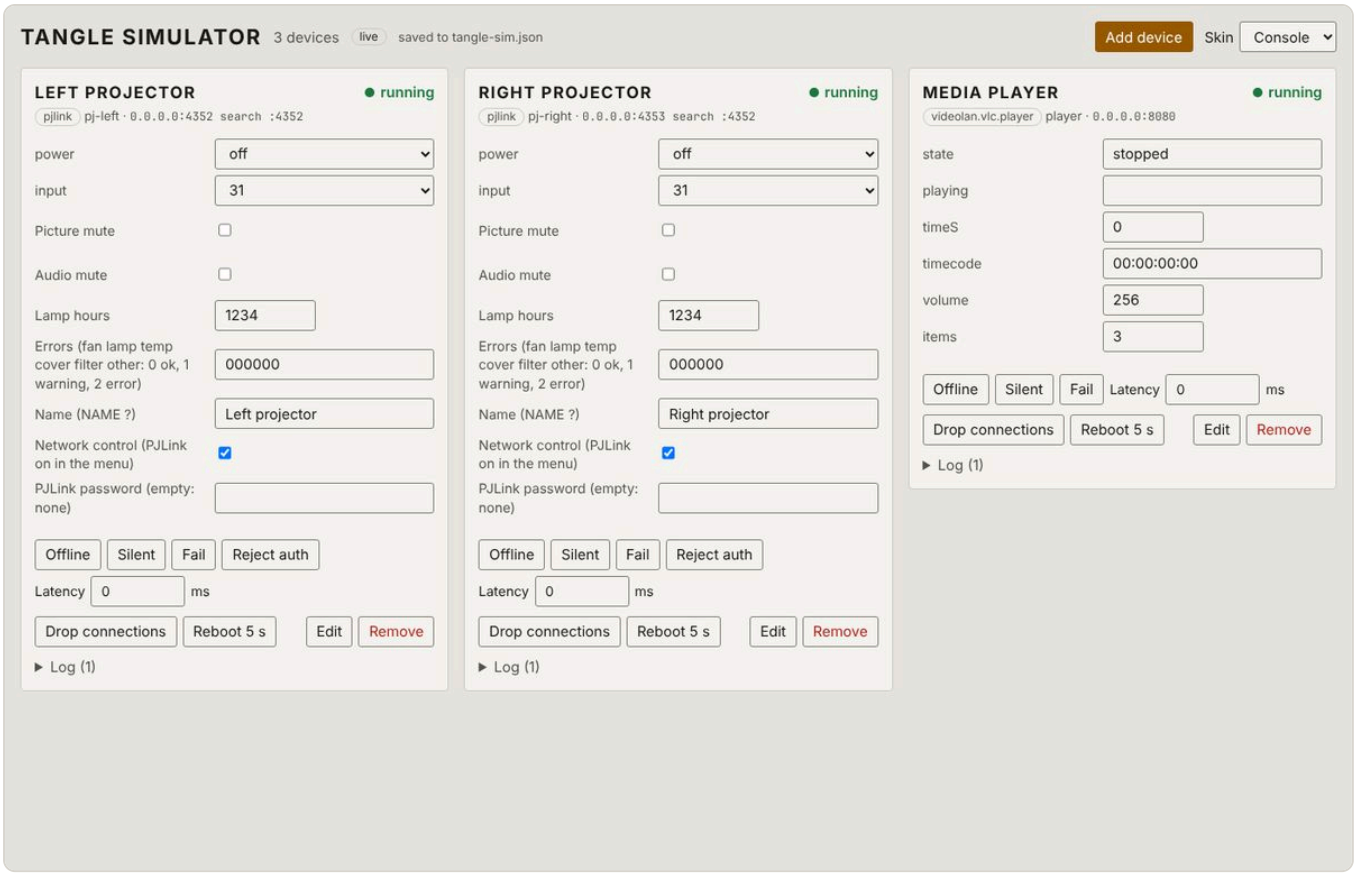
In the simulator's web UI, click **Add device** and add each device. The form asks for a **Type**, an **ID**, a **Name**, a **Port** and the settings of the type:

TYPE	ID	NAME	PORT	SETTINGS
`pjlink`	`pj-left`	Left projector	4352	Warm-up seconds `8`
`pjlink`	`pj-right`	Right projector	4353	Warm-up seconds `8`
`videolan.vlc.player`	`player`	Media player	8080	Password `secret`

Each **Add** is saved to `tangle-sim.json`. To skip the form, stop the simulator (Ctrl+C), replace the file with this and start it again:

```json
{
  "devices": [
    { "id": "pj-left", "type": "pjlink", "name": "Left projector", "port": 4352,
"warmSeconds": 8 },
    { "id": "pj-right", "type": "pjlink", "name": "Right projector", "port": 4353,
"warmSeconds": 8 },
    { "id": "player", "type": "videolan.vlc.player", "name": "Media player", "port": 8080,
"password": "secret" }
  ]
}
```

The web UI shows a card per device with its live state (the projectors' power and input, the player's state and current item) and buttons to cause faults. Keep it open in a browser tab: you will watch the devices react there.



> **Note:** A real projector takes 30 to 60 seconds to warm up. The simulator's 8 seconds keeps the tutorial short while still showing why the automation waits for it.

---

2. Start Tangle and make the project

With the desktop app

Open Tangle. It starts its own core, signs you in and opens the designer. The drivers bundled with the app are ready to install from the designer.

With the core service

Put the module packages you need (here `misc.pjlink.projector` and `videolan.vlc.player`) in a folder named `modules` next to `tangle-core`, one sub-folder per module: the designer offers them for install. Then start core and create a token for the designer:

```sh
./tangle-core run
./tangle-core token create --name designer --scopes read,control,configure   # in a second terminal
```

Open `http://127.0.0.1:7340/designer`, paste the token and click **Sign in**.

> **Note:** Building Tangle from source? `./start.sh --sim` starts core and a simulator with its own demo devices on the same ports. Use plain `./start.sh` with the tutorial's simulator instead.

Create the project

Tangle keeps a library of projects on the computer.

1. Click the project name at the top of the sidebar, then **Open or new project...**.
2. Click **New project**, type `Gallery installation` and click **Create and open**.

An empty project opens on **Start a project**, a wizard that builds a project from a template in four steps (kind, questions, devices, review). Its **Simple installation** kind makes the same kind of project as this tutorial, with one big Start and Stop. Here you will build it by hand to see each part: click **Start empty**.

There is no Save step to remember: every change is saved to the project as you make it. **Save as...** and **Export...** in the project menu make copies.

---

3. Add a room and the devices

The room

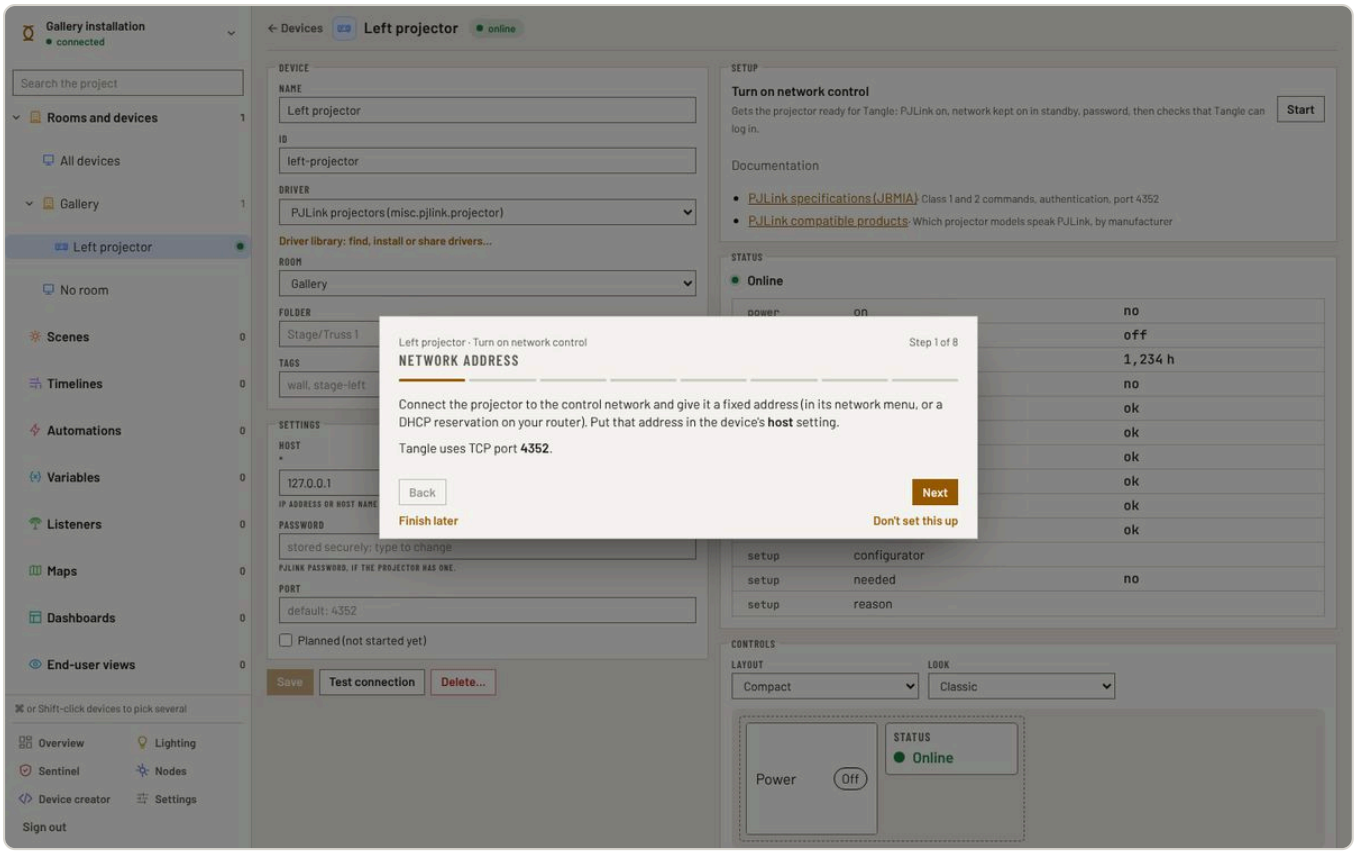
Rooms group devices the way the building does. Automations and dashboards can then say "the Gallery" instead of listing each device, so a fourth projector added later is included without changing them.

1. In the sidebar, right-click **Rooms and devices** and choose **New room...**.
2. Type `Gallery` and click **Add**.

The left projector

1. Right-click **Gallery** in the sidebar and choose **Add device**.
2. Type the **Name** `Left projector`. The **Id** `left-projector` is filled in for you: automations and the API use it, so pick names you will keep.
3. **Driver** lists only the drivers already installed. Click **Driver library: find, install or share drivers...**, search for `PJLink` and click **Use** next to PJLink projectors. Tangle asks "Install PJLink projectors? It asks for: network.": click **Install**.
4. The **Settings** part of the form now comes from the driver. Set **host** to `127.0.0.1` (or the simulator machine's address). Leave **port** empty: it defaults to 4352. This projector has no password.
5. Click **Create device**.

The device page opens with its **setup assistant**, "Turn on network control": the steps a technician follows on the real projector (fixed address, PJLink on, network kept on in standby, password). The simulated projector needs none of them, so click **Finish later**. The assistant stays on the device page under **Setup** for the day the real projector arrives.



Within a few seconds the status next to the name turns **online**. The device page shows what Tangle knows about it:

- **Status**: the live state the driver reads (power, input, lamp hours, errors).
- **Controls**: the controls the driver offers, in a **Layout** (Compact, Operator, Full) and a **Look**. Press **Power**: the simulator card shows the projector warming for 8 seconds, then on. Press it again to turn it off.
- **Log**: every action, state change and error of this device.

Test connection checks the settings in the form without saving them, which helps when a device does not come online.

The right projector

Add it the same way: **Name** `Right projector`, **Driver** PJLink projectors (now in the list, since it is installed), **host** `127.0.0.1`, **port** `4353`.

The media player

1. Add a third device in Gallery, **Name** `Media player`.
2. In the **Driver library**, search for `VLC media player` and **Use** it, then **Install**.
3. Set **host** `127.0.0.1` and **password** `secret`. **port** defaults to 8080.
4. Click **Create device**, then **Finish later** on the assistant ("Turn on VLC's web interface").

Passwords are not written in the project: they go to the computer's credential store, and exports leave them out.

Click **All devices** in the sidebar: the three devices are listed with their driver, room and status **online**. This is the moment to check every device on a real site, before writing automations that depend on them.

---

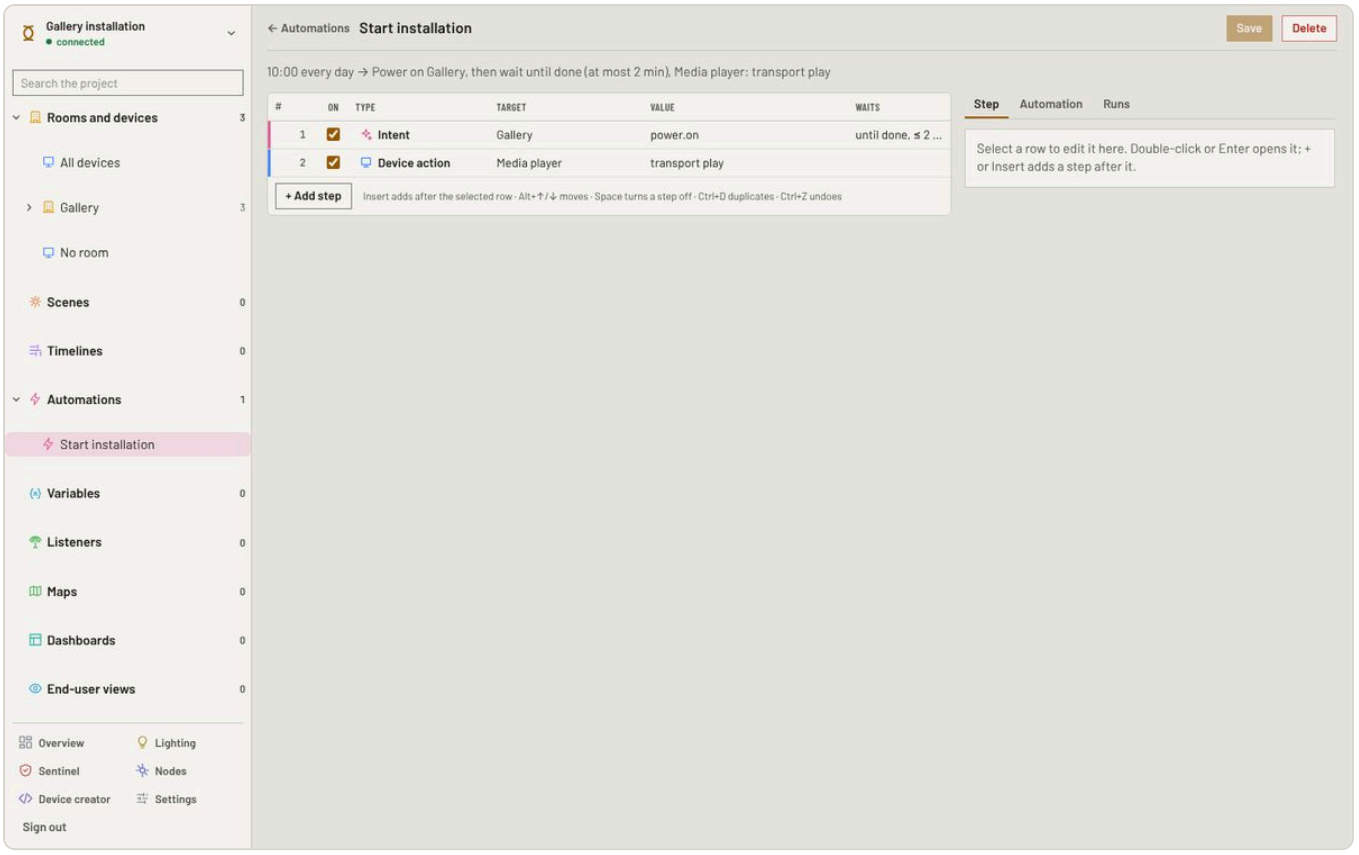
4. Automate the start and the stop

An automation is a list of steps run in order, started by triggers (a time, a device event, a button) or by hand.

Start installation

1. Right-click **Automations** in the sidebar and choose **New automation**.
2. The step table starts with one **Intent** row. Select it: the **Step 1** tab on the right edits it.
   - **Do**: Power on.
   - **Target**: Room, **Room**: Gallery.
   - Tick **Wait until the device reports it is done**. **At most** stays at 120 s, and **Then** at "go on": a projector still warming after 2 minutes does not block the video forever.
3. Click **+ Add step** and choose **Device action** in the Device group.
   - **Device**: Media player. **Action**: transport play.
4. Open the **Automation** tab.
   - **Name**: `Start installation`.
   - Under **Run when (any of)**, click **Add trigger**, choose **Time of day**, set **At** to `10:00` and keep every day ticked.
5. Click **Create automation**.

The line above the table sums it up: "10:00 every day → Power on Gallery, then wait until done (at most 2 min), Media player: transport play".



Why an intent instead of two device actions? An intent states the goal ("power on the Gallery") and each driver translates it for its device: both projectors turn on, and the media player, which has no power, is left alone. The **wait until done** option holds the automation until both projectors report they are on, so the video never starts on a dark wall.

Try it now instead of waiting for 10:00: open the **Runs** tab and click **Run now**. The run shows "Running: step 1" while the simulator's projectors warm up, then the player starts and the run ends as **completed**, after about 8 seconds.

Stop installation

Make a second automation the same way, named `Stop installation`:

1. **Device action**: Media player, transport stop.
2. **Intent**: Power off, Target Room Gallery. No need to wait here.
3. Trigger: **Time of day** at `18:00`.

Stopping the video first means visitors never see the player's desktop on a projector that is still cooling.

> **Tip: Behaviour** in the Automation tab decides what happens when an automation starts again while it is still running (here, a second Start press during the warm-up). The default, single, ignores the second start.

5. Build the staff dashboard

Create it

1. Click **Dashboards** in the sidebar, type `Gallery panel` in **New dashboard** and click **Create**. The builder opens.
2. Click **Save draft** now and then as you work. Nothing reaches screens until you **Publish**.

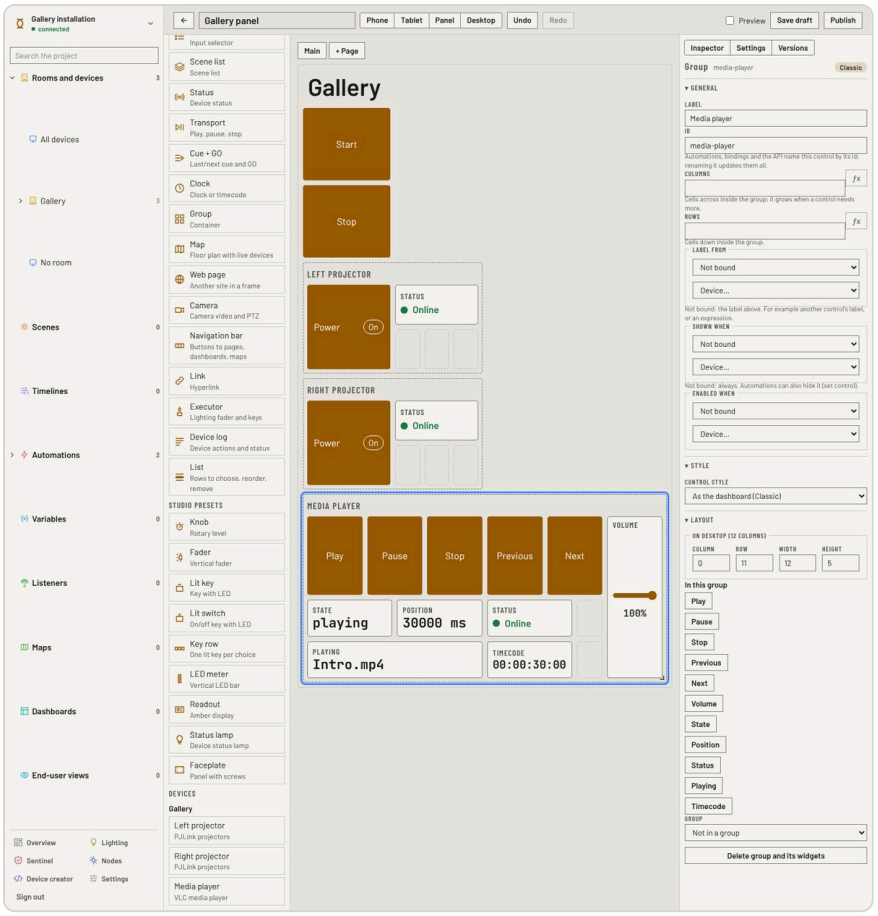
The builder has the palette on the left, the page in the middle and the **Inspector** on the right. **Phone**, **Tablet**, **Panel** and **Desktop** at the top show the page at each screen size.

Title and buttons

1. Drag a **Label** from the palette to the top of the page. In the Inspector, set **Text** to `Gallery` and **Size** to Title.
2. Drag a **Button** under it. Set **Label** to `Start`, then:
   - **Action**: Run an automation, **Automation**: Start installation.
   - **Lit when**: Device state, Media player, `tangle.transport.status`, lit while it is playing. The button lights up while the installation runs, wherever it was started from.
3. Drag a second **Button**. **Label** `Stop`, **Action** Run an automation, Stop installation, and **Confirmation** Press twice. A single wrong press only asks "Press again to confirm".

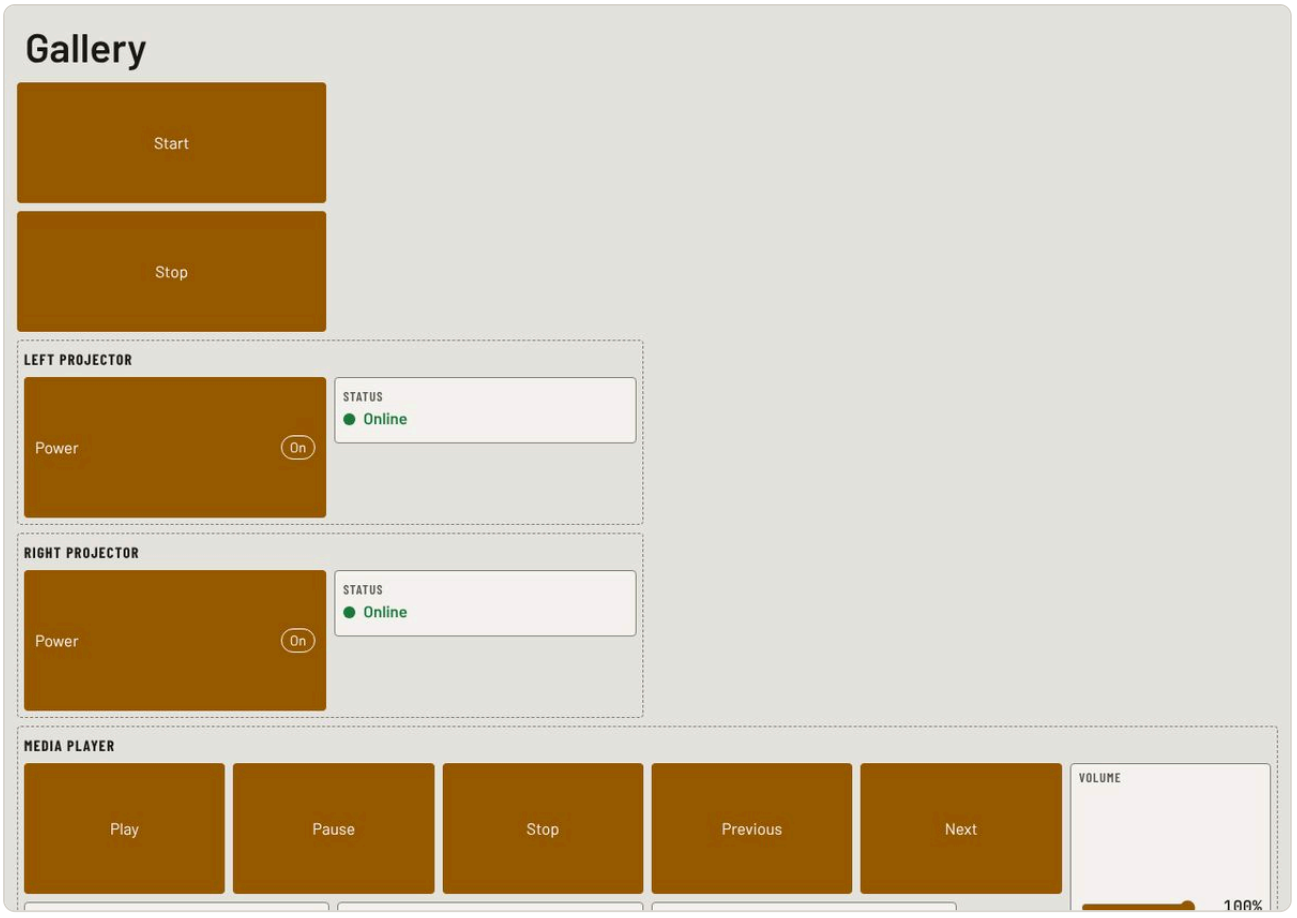
Device controls

Under **Devices** at the bottom of the palette, click **Left projector**. Tangle asks which **Layout** to use and which controls to show: choose **Compact** (power and status) and click **Add to page**. The controls arrive as a group bound to the projector. Do the same for the **Right projector**, and for the **Media player** with the **Operator** layout (play, pause, stop, previous, next, volume, and what is playing).



Publish and open it

1. Click **Save draft**, then **Publish**. The builder confirms "Published version 1". Screens showing the dashboard reload by themselves on each later publish.
2. Open `/d/gallery-panel` on your core: `http://127.0.0.1:7340/d/gallery-panel` with the core service.



Try it:

- Press **Start**: the projectors warm up in the simulator, the video starts, and **Start** lights up.
- Press **Stop** once: it asks "Press again to confirm". Press again: the video stops and the projectors cool down.

To open it on the staff tablet, core must accept other machines: start it with `--listen 0.0.0.0:7340 --allow-remote --insecure-http` and a license or `--demo`, then open `http://<core address>:7340/d/gallery-panel`. See End-user views to give the staff a menu protected by a PIN instead of a bare address.

---

6. Rehearse a failure

Projects are judged on the day something breaks. The simulator lets you rehearse it:

1. In the simulator's web UI, click **Offline** on the Right projector.
2. In the designer, the Right projector turns **offline** in the sidebar and in **All devices**. The **Overview** lists "Right projector is offline" under **Needs attention**, with a **Fix** link to the device. On the dashboard, the projector's status turns offline.
3. Press **Start** on the dashboard. The left projector warms up, but the run stops at step 1: an action sent to an offline device fails at once, so the video does not start. The automation's **Runs** tab shows the run as **failed**, with the step and the reason.

4. Click **Offline** again in the simulator to bring the projector back. Tangle reconnects by itself.

Whether a half-ready room should still start is a choice for each site. To start the video anyway, tick **Continue after a failed step** under **Behaviour** in the Automation tab. **At most** and **Then** decide the other case, a device that answers but takes too long: go on, or fail the run. **Sentinel** (Pro and Ultimate) can also send an email or a message when a device stays offline: see Sentinel.

---

7. Hand it over

- **Export...** in the project menu writes a `.tangle` file of the project, without passwords: keep it with the site documents, or **Import...** it on the site's computer from the project list.
- Replace the simulator by the real devices: change each device's **host** to the real address and run its setup assistant on site.
- Licensing explains which license the site needs. Without one, core runs as the free designer and pauses every hour.

What you learned

CONCEPT	WHERE IT WAS USED
Rooms	Automations target "the Gallery", not each device
Drivers and the Driver library	PJLink and VLC were installed when first used
Setup assistants	The steps to prepare each real device, kept for the site visit
Intents	"Power on" means the right thing for each device in the room
Wait until done	The video waits for warm projectors
Automations and triggers	One Start, run by the clock, a button or by hand
Dashboards	Buttons run automations; device groups come from the drivers
Confirmation	Stop needs two presses

Next steps

- Give the video a timeline with cues: Timelines.
- Change the opening hours without editing automations, with a project variable: Automations.
- Let another system start the room over the network: Listeners.

- Add a device Tangle has no driver for: Device creator.

GUIDES

Designer

The Tangle designer, on the web and on the desktop - projects, the project tree, signing in, token scopes, each section, and how saving and conflicts work.

The designer is where you build a Tangle project without editing JSON: rooms, devices, scenes, timelines, automations, variables, listeners, maps, dashboards, lighting and Sentinel monitoring. It is one app shipped two ways, a web app served by core and a desktop app, and both work the same once signed in.

Web and desktop

	WEB	DESKTOP APP
Where	`http://<core>:7340/designer` (core's home page `/` opens it)	Tangle app for macOS, Windows and Linux
Which core	The core that serves the page	Its own bundled core, or any core on the network
Sign-in	A token	Automatic for the bundled core; a core address and a token for others

The web designer needs nothing installed: any machine that can reach core opens it in a browser. The desktop app is the all-in-one install for a laptop: it includes core, the official device modules and Node.js, and starts its own core when it opens. See Install.

The old address `/builder` redirects to `/designer/dashboards`.

Signing in

On the web

1. Create a token on the core machine:

```sh
tangle-core token create --name designer --scopes read,control,configure
```

The secret is printed once. For a core installed as a service, run it with `sudo` (an administrator shell on Windows).

2. Open `/designer`, paste the token in **API token** and click **Sign in**.

The token is kept in the browser, so you stay signed in. It is shared with the dashboard builder: one sign-in covers both. Click **Sign out** at the bottom of the left menu to forget it.

On the desktop

When the app opens, it starts its bundled core ("Starting the core on this computer..."), creates a token for itself and signs in. You see the project that was open last.

To work on a core on the network instead:

1. Click **Sign out**.
2. Enter the **Core address**: an IP address or host name such as `10.0.0.2`, or a full address such as `http://show-core.local:7340`. Port 7340 is added when you give none.
3. Paste a token created on that core and click **Sign in**.

The app remembers the last five cores in the address field. To go back to the bundled core, sign out and click **Use the core on this computer**. The app remembers which one you chose.

> **Note:** The remote core must accept other machines (`--listen 0.0.0.0:7340 --allow-remote --insecure-http`) and must not be running as the free designer, which serves only its own computer.

Token scopes

A token carries one or more scopes. The designer checks what yours allows when you sign in.

SCOPE	LETS YOU
`read`	See the project, live state, run logs and alerts
`control`	Drive timelines (Play, Pause, Stop, GO, seek) and send intents
`configure`	Change the project: every Save, Create and Delete, run automations by hand, publish dashboards
`admin`	Tokens, licensing, node settings and pairing
`live`	For MCP clients: change things while a timeline plays

With `read` alone, the designer is read only: forms are disabled and the menu shows "Read only: this token cannot change the project (it needs the configure scope)." For day-to-day design, use `read,control,configure`. The desktop app's own token has `read,control,configure,admin`.

To revoke a token: `tangle-core token revoke designer`. It stops working at once, without a restart.

Projects

Core keeps a library of projects on its computer and runs one at a time. Click the project name at the top of the sidebar for the project menu:

ITEM	DOES
Save (Ctrl+S)	Writes the project now. Changes are saved as you make them anyway.
Save as…	A copy under a new name, which becomes the open project
Export…	A `.tangle` file (a zip with the project, dashboards and plans) to keep or move to another core. Device passwords are never exported.
New window (Ctrl+Shift+N)	Another designer window on the same project
Open or new project…	The project list: open, duplicate or delete a project, **New project**, and **Import…** a `.tangle` file
Close project	Closes it; core then refuses changes until a project is opened

Opening another project while a timeline plays asks first, since it stops the show.

A new, empty project opens on **Start a project**, a wizard that builds a project in four steps (kind, questions, devices, review). Kinds include meeting and class rooms, live shows, museums and exhibits, digital signage, building monitoring, simple installations, and rides and attractions. **Start empty** skips it; **Start from a template** on the Overview brings it back. Tutorial: a gallery installation builds a project by hand.

Layout

The **project tree** on the left holds the whole project: **Rooms and devices** (All devices, each room with its devices, No room), Scenes, Timelines, Automations, Variables, Listeners, Maps, Dashboards and End-user views, each with a count. **Search the project** at the top filters it. Under the tree are Overview, Lighting, Sentinel, Nodes, Device creator, Settings and **Sign out**.

The project name and the connection state are on top (`connected`, `backup node` when signed in to a redundant backup, or the reconnecting state). Device status, timeline position and automation runs update live over the WebSocket.

- **Right-click** anything in the tree, or on a page, for its menu: **New room…**, **Add device**, **New automation**, **Rename…**, **Copy id**, **Move to ▸**, **Open in new window**, **Delete…** and so on. On a dashboard control, **Automate ▸** makes an automation that starts when it is clicked or changes.
- **⌘-click** or **Shift-click** devices in the tree to pick several.

- **Middle-click** (or **Open in new window**) opens an item in its own window, for example a dashboard on one screen and its automations on another.

Settings > Designer > Navigation switches to the older **Section menu**, one entry per section, for people who prefer it.

Overview

The first page shows counts of rooms, devices (with how many are online), scenes, timelines, automations, dashboards and maps (click one to open it), and **Needs attention**:

- devices whose module is not installed on this core;
- devices that are `offline`, `timeout` or `error`, with the detail the module gave;
- devices that recommend a setup assistant;
- automations whose last run failed;
- devices that are not in a room.

Each line has a **Fix** or **Assign** link to the right page.

Rooms

Rooms and devices in the tree, or the **Rooms** page: a board with one column per room and a **No room** column. Drag a device between columns to assign it; the move is saved at once.

- **Add room**: right-click **Rooms and devices** and choose **New room...**, or type a name in **New room** on the Rooms page. The id is made from the name (`Meeting room 2` becomes `meeting-room-2`).
- Right-click a room to rename it, copy its id, add a device to it or delete it. **Edit** on the board also sets its tags.
- Open a room to see its devices, with **Add device**.

Devices

All devices lists every device with its driver, room, folder, tags and live status, with folders, filters and multiple selection. A device page holds its settings, setup assistants, live status, ready-made controls and log. **Add device**, the **Driver library**, **Test connection**, **Scan the network**, setup assistants and deleting devices with what uses them are covered in Devices.

Scenes

A scene is a set of device actions recalled together, for example "Presentation" turns on the projector and lowers the screen.

1. Click **Scenes** in the tree, then **New scene**, and give it a name.
2. Click **Add action** for each step: choose the **Device**, then the **Action** from that device's capabilities (`power on`, `input select`, `level set`...), then its parameters. Parameters show their type and range, for example `level (0–1)`.
3. Click **Create scene**.

Scenes are recalled with the `scene.recall` intent from dashboards, cues and automations. See Concepts.

Timelines

New timeline opens the editor. Settings at the top:

SETTING	OPTIONS
Mode	**Timecode** (runs freely from Play) or **Cue list (GO)**
Clock	**Internal**, **Chase timecode (LTC, MTC, Art-Net, media servers)** with a timecode input device or a media player or server, which of its timelines to follow, and an offset, **Time of day** with a time zone, or **PTP** with a PTP clock reference device
Frame rate	24, 25, 29.97df, 30, 50, 60, or none (cues snap to a 100 ms grid)
Duration (s), Loop	Length and looping
Generate timecode on	Timecode mode: output devices that follow the timeline, each with an optional start timecode

Editing cues

- **Timecode mode**: tracks are rows on a time ruler. Double-click a track to add a cue, drag cues to move them (they snap to frames), and use **Zoom** to change the scale.
- **Cue list mode**: cues are listed in order with **Move up** and **Move down**.
- **Add track** adds a row. Each track has a name, **Mute** and **Delete**.

Select a cue to edit it in the side panel: **Name**, **Time (s)**, **Pre-roll (ms)**, **Enabled**, and what it **Does**: a device action, or an intent (recall a scene, power, level...) on a room, device, tag or everything. A cue whose device was deleted becomes a comment cue, a note with **Point at device...** to turn it back into a cue.

Live transport

Tick **Live control** to enable the transport buttons: **Play**, **Pause**, **Stop** (or **GO**, **Pause**, **Stop** for a cue list). With live control on, clicking the ruler seeks. The header shows the status and position, plus **chasing** and **no timecode** badges for chased timelines, **following time of day** for time-of-day timelines, and **PTP not locked** for PTP timelines.

While a timeline plays, its editor is read only. Tick **Allow live editing (changes apply to the running show)** to change it anyway. A playing timeline cannot be deleted.

See Timelines.

Automations

Right-click **Automations** in the tree and choose **New automation**. The editor is a step table (one row per step, with **+ Add step**) and three tabs: the selected **Step**, the **Automation** (name, params, **Run when**, **Only if**, **Behaviour**) and its **Runs** (**Run now**, **Stop**, live run log). See Automations.

Variables

Project variables: values core keeps for the whole project, read as `{{name}}`. **Add variable** gives each a name, a label, a default value and **Keep** (kept across restarts), then **Save variables**. The current values show live. See Automations.

Listeners

Ports other systems send their own messages to, with rules, a live message log and a tester. See Listeners.

Maps

Floor plans with the devices placed on them. **Import a plan...** (or drop an SVG, DWG, DXF, PNG or JPEG plan on the page) starts a map; positions are in metres, so tracked people and tags show on the same map. Dashboards show maps with the Map widget, and end users open them from their views.

Dashboards

The dashboard builder, full width. From the dashboard list you can create, generate (room, show console, facility overview) and **Open** a published dashboard. The builder has the control palette, the grid per screen size, the

Inspector, settings (skin, control style, theme, background, units, access) and published versions. See Dashboards.

End-user views

What the people using the site open at `/view`: a menu of dashboards and maps, with its own access. See End-user views.

Lighting

The lighting engine: **Program** (pick lights on the stage, build looks, type commands like `1 thru 6 at 50`), **Run show**, **Cue lists**, **Presets**, **Playback** (executors and the grand master) and **Patch** (add lights from the fixture library).

Sentinel

Monitoring setup and live alerts, in six tabs:

TAB	WHAT IT DOES
Health	Status of devices, core nodes and probes
Alerts	Active alerts and history; **Acknowledge** (with a note) and **Silence...**
Alert rules	Conditions (status, state, metric, stale, core node), severity (info, warning, critical), delay and channels
Channels	Email, SMS, push, webhooks, Slack, Teams; send a test
Maintenance	Windows when alerts are muted
Probes	TCP and HTTP checks for things that have no device module

Sentinel is part of Tangle Pro and Ultimate. With another license you can still set up rules and channels, but nothing is checked or sent. On a backup core, Sentinel watches but only the primary sends notifications. See Sentinel.

Nodes

The **Nodes** page shows this core (CPU, memory, disk, devices, module processes, uptime) and its pairing code, the members of its redundant cluster and the cores found on the local network. When a new core appears, a banner tells you. Each found core offers **Keep separate** (its own show, not offered again) or **Add as redundant** (type its pairing code and it joins as a backup). Accepting needs the `admin` scope.

The page also sets the cluster's **Address for operators** (one `name.local` address that follows the primary), **Sharing the work** (standby or shared mode, folder rules, **Rebalance**) and the **Health limits** that mark a node "low disk", "low memory" or "overloaded". See Redundancy.

Device creator

Write a driver for a device Tangle does not know, without code where possible. See Device creator.

Settings

- Project **Name**, **Id**, **Contract** version (the project's and the designer's) and the **Core** address.
- **Export project (JSON)** downloads the project as core serves it. Secret settings are never included.
- **Designer**: the navigation (project tree or section menu), for this browser.
- **License**: what this core runs as, the license, and **Activate**, **Sign in with email**, **License by email / file...** and **Offline**. See Licensing.
- **Community**: sign in to the community driver repository. See Device creator.
- **Units**: metric or imperial for the project, this browser's choice, and overrides per quantity.
- **Dashboard controls**: the project's default control style and the highlight colours of each style.
- **Modules**: every module the core knows, installed or not, with **Install** and **Scan again**.

Saving and conflicts

Each editor saves one thing at a time (a room, a device, a scene, a timeline, an automation, a Sentinel rule). Every save works in two steps:

1. **Dry run.** The designer sends the change with `dryRun=true`. Core checks the whole project with the change applied (schema, references, settings) and changes nothing. If something is wrong, core's errors appear in a box next to the form and nothing is saved.
2. **Write.** The same change is sent for real, with the version tag (ETag) the editor read when it opened.

Saved changes apply to the running project at once and are written to the project file.

If someone else changed the same item since you opened it (another designer, a script, an AI assistant over MCP), core refuses the write and the designer shows "Someone else changed this since you opened it." with two choices:

- **Reload theirs**: discard your edits and load the current version.
- **Overwrite with mine**: save your version over theirs.

Deletes are checked the same way, so a delete that would leave the project invalid, such as removing a device a scene still uses, is refused with the reason. Changes made elsewhere appear in the designer on their own, because core announces every project change.

Dashboards work differently: the builder saves drafts and publishes versions. See Dashboards.

GUIDES

Devices

Add, configure, test and discover devices, manage the device modules that drive them, and pin hardware to one machine.

A device is one thing Tangle controls: a projector, a media server, a PC, a DMX universe, a timecode input. Each device uses a device module, which knows how to talk to that kind of device. This page covers adding and configuring devices, testing them, finding them on the network, and installing the modules they need.

Devices and modules

Every device in a project has:

FIELD	MEANING
`id`	Lowercase identifier, unique in the project (`pj-1`, `hall.projector-2`)
`name`	Display name
`moduleId`	The module that drives it, for example `misc.pjlink.projector`
`config`	Its settings (address, port, password…), checked against the module's settings schema
`room`	Optional room id
`tags`	Optional list of tags
`folder`	Optional folder in the device list, `/`-separated (`Stage/Truss 1`). One folder per device; tags are for groups that cut across folders
`planned`	Optional. `true`: the device is in the project but not started yet (not delivered, not cabled)
`node`	Optional cluster node the hardware is attached to (see Hardware attached to one machine)
`runsOn`	Optional preferred node for a networked device when the cluster runs in shared mode (see Redundancy)
`parent`	Optional device this one is a sub-device of (a light under its DMX output)

Module ids follow `<manufacturer or misc>.<product or protocol>.<category>[.<version>]`, for example `barco.pulse.projector` or `dataton.watchout.mediaserver.v7`. The Device catalog lists every

official module.

Add a device

In the designer

1. Click **Add device** on the **Devices** page, or right-click a room in the project tree and choose **Add device**.
2. Type a **Name**. The **Id** is filled in from the name; you can change it until the device is created.
3. Pick the **Driver**. The list shows the drivers installed on the core. To use another one, click **Driver library: find, install or share drivers...**: search by manufacturer, product or name, and click **Use**. A driver that is not installed yet is installed first, after you confirm the permissions it asks for (network, serial...). See Driver library.
4. Optionally pick a **Room**, a **Folder** and **Tags** (comma-separated).
5. Fill in **Settings**. This form is built from the driver's settings schema: text, number, on/off and list fields, with defaults shown as placeholders and required fields marked with `*`. Settings the form cannot show as a simple field are edited as JSON. A driver with several connections (LAN and RS-232, say) shows the connection list first and only the settings of the chosen connection.
6. Tick **Planned (not started yet)** for a device that is not on site yet: it stays in the project, its cues and controls can be programmed, and core does not try to reach it.
7. Click **Create device**.

Tangle checks the settings against the driver's schema when you save. A missing required setting, such as `host` for a PJLink projector, is refused with an error naming the setting.

When the driver has a setup assistant meant for new devices, it opens right away.

The device page

Open a device from the project tree or the device list. Beside its settings (with **Save**, **Test connection** and **Delete...**), the page shows:

- **Setup**: the driver's setup assistants and documentation links (the vendor's protocol reference, compatible models).
- **Status**: the device's live state, as the driver reports it.
- **Controls**: the controls the driver offers, ready to use, in a **Layout** (Compact, Operator, Full, and more for some drivers) and a **Look** (Classic, Studio, Skeuomorphism, Relief). The same layouts are offered when you add the device to a dashboard.
- **Log**: see Device log.

Driver can be changed on an existing device. The id and the settings the new driver also has are kept, and Tangle warns about the cues, scenes, automations and dashboard controls that use something the new driver does not offer.

The device list

Devices (or **All devices** in the project tree) lists every device with its driver, room, folder, tags and live status, and "Setup recommended" when a device answers but is not set up for Tangle yet.

- **Folders** on the left, with counts, **All devices** and **No folder**. **+ New folder**, rename, and drag devices onto a folder.
- **Filters**: search, type, driver, room, tag and status.
- **Selection**: tick devices (or ⌘ / Shift-click them in the project tree) to tag, move to a folder or delete several at once.

A device whose driver is not installed shows its driver id with "(not installed)".

Delete devices

Delete… first lists everything that uses the devices: scenes, timeline cues, automations, maps, dashboards, lighting, alert rules, maintenance windows and sub-devices. Then choose:

- **Delete everything**: their timeline cues and dashboard controls go too.
- **Keep timeline cues as comments**: each cue becomes a note ("Removed device Projector 1: power on") that **Point at device…** turns back into a cue later, and dashboard controls stay, showing the device as missing.

Either way, scene actions, map placements, lighting values and the automation triggers, conditions and steps that use the devices are removed, since they cannot run without them. An automation left without triggers or steps is deleted; one that lost a condition is turned off, so it never runs more often than intended. The API does the same with `POST /api/v1/devices/references` and `POST /api/v1/devices/remove`.

From the command line

The `tangle` command edits the project file directly:

```sh
tangle device add -p show.yaml pj-1 -m misc.pjlink.projector --room hall --set host=10.0.0.5
--set password=admin
tangle device list -p show.yaml
tangle device remove -p show.yaml pj-1
```

`--set key=value` can be repeated. Numbers and booleans are typed from the module's schema. `--name` sets the display name.

> **Note:** `tangle device` commands refuse to change a project while tangle-core is running it. Use the designer, the REST API (`/api/v1/devices`) or MCP instead: they run the same checks.

Import and export device lists

To add many devices at once, import a CSV or JSON file:

```sh
tangle device import -p show.yaml displays.csv
tangle device import -p show.yaml displays.csv --dry-run        # check only
tangle device import -p show.yaml displays.csv --skip-invalid   # add the valid rows anyway
tangle device export -p show.yaml --format csv -o displays.csv
```

CSV columns are `id`, `name`, `moduleId`, `room`, `tags` (separated by `;`) and one `config.<setting>` column per setting, for example `config.host`. Cells are typed from the module's `configSchema`. Without `--skip-invalid`, nothing is added if any row is invalid; errors give the line number. `export` writes to stdout without `-o`; `--format` is `csv` (default) or `json`.

Passwords and other secrets

Settings marked `"x-secret": true` in a module's schema (passwords, PINs, tokens, private keys) never go into the project file. They are stored in the operating system's credential store on the machine that runs the device:

MACOS	WINDOWS	LINUX
Keychain	Credential Manager	Secret Service

Entries use service `tangle` and account `<deviceId>/<setting>`. On Linux, when no credential store answers (a service without a desktop session), secrets go to `secrets.json` in the Tangle data folder, readable by its owner only. Set `TANGLE_SECRETS=file` to force that file.

In the designer, secret fields are password boxes ("stored securely; type to change"). On the command line, `--set` moves secret settings to the store, or use:

```sh
tangle device secret pj-1 password           # prompts without echo, or reads stdin when piped
tangle device secret pj-1 password --delete
```

Because secrets stay on the machine, a project file copied to another computer contains no passwords. Set them again there.

Test a connection

A connection test starts the device's module on its own, in a separate process, adds the device and waits for it to come online. It never touches the running project's modules. It gives up after 10 seconds.

- **Designer:** open the device and click **Test connection**. It tests the settings in the form, without saving. The result shows "Connected" with the status and time, or "Not connected" with the reason.

- **Command line:** `tangle device test -p show.yaml pj-1`
- **API:** `POST /api/v1/devices/{id}/test`, or the MCP tool `test_device`.

Setup assistants

Many devices need something switched on before Tangle can control them: PJLink on a projector, the web interface of VLC, Remote Login on a Mac. A **setup assistant** walks through it: what to enable on the device, step by step, then does the rest itself where it can (saves the password it was given, installs an SSH key, checks that the device answers).

- An assistant meant for new devices opens right after **Create device**. **Finish later** closes it and keeps your progress; **Don't set this up** dismisses it.
- The **Setup** card of the device page lists the driver's assistants (**Start**) and its documentation links.
- A device that answers but is not set up for Tangle yet (remote control off, no password, key refused) shows **Setup recommended** in the device list and on its page.

Assistants exist for PJLink projectors (network control, password), SSH computers (Remote Login, key, power rights), PowerShell remoting, PsExec, VLC (web interface, preview stream) and Reaper. Progress is saved with the device (`setup`), never its secrets. API: `POST /api/v1/devices/{id}/configurators/{configurator}/run`, MCP `run_device_setup`.

Device log

Core keeps the last 500 entries of each device: the actions sent to it (by whom: a dashboard, an automation, a timeline, the API) and how they ended, status changes, state changes (changes within 2 seconds are merged) and device events. Each entry has a level: info, warning or error. The log survives a core restart and is deleted with the device.

The **Log** panel of the device page follows it live, with filters (Actions, Status, State, Events, Notes; Everything, Warnings and errors, Errors), **Clear**, and **Add** to write a note ("lamp replaced"). Dashboards show it with the **Device log** widget, and automations write to it with a **Log** step.

Some actions take time: a projector warming up reports "warming up", then done or failed. The log shows each step, and automations can wait for the end (see Automations).

API: `GET /api/v1/devices/{id}/log` (`limit`, `before`, `kinds`, `level`), `GET /api/v1/devices/log?devices=a,b` (several merged), `DELETE /api/v1/devices/{id}/log`; event topic `device.log <deviceId>`; MCP `get_device_log`, `add_device_log_note`, `clear_device_log`.

Network serial ports

Serial devices (RS-232, RS-422, RS-485) are often far from the computer that runs Tangle. Every serial driver, and the device creator's serial connection, accepts a network serial port wherever it asks for a serial port:

PORT	FOR
`/dev/tty.usbserial-A10`, `COM3`	A port on this computer
`tcp://192.168.1.50:4001`	An IP-to-serial gateway in TCP server (raw socket) mode
`rfc2217://192.168.1.50:4001`	A gateway with RFC 2217 (Telnet COM port control): Tangle sets the baud rate, parity, stop bits, flow control, DTR and RTS on the gateway
`udp://192.168.1.50:4001?listen=4101`	A gateway in UDP mode, answers on the `listen` port

In the designer the serial port field is a picker: this computer, gateway TCP, RFC 2217 or UDP, with a search for gateways. **Scan the network** also finds Moxa, Lantronix and Digi gateways and suggests the generic serial driver with their address.

A gateway on the network also keeps the device reachable when the primary of a cluster changes, which a USB adapter cannot.

Find devices on the network

Discovery listens for mDNS/DNS-SD services, asks UPnP devices to answer (SSDP), and runs the search of every installed module that has one (for example the PJLink class 2 search, BrightSign players, Art-Net nodes). For each device found it lists the address and, when known, the name, manufacturer and model, plus suggested modules: official modules first, exact model matches before manufacturer-only matches. A module that found the device itself is suggested first.

- **Designer:** on the Devices page, click **Scan the network** (5 seconds). Click **Add** next to a result to open a new device with the suggested module, name and `host` filled in.
- **Command line:** `tangle discover --seconds 5` (1 to 30 s; `--json` for machine-readable output).
- **API:** `POST /api/v1/discovery`, or the MCP tool `discover_devices`.

Discovery only runs when someone starts it. tangle-core refuses a scan while a timeline is playing (show mode). `tangle discover` refuses while any project is running and points you to the core's API.

Statuses

Every device reports a connection status:

STATUS	MEANING
`connecting`	First connection in progress
`online`	Connected and answering
`degraded`	Answering, but reporting a problem (a lamp warning, an alarm, one failed poll)
`offline`	Unreachable, refused or disconnected (device off, cable pulled, wrong address)
`timeout`	Reachable or connected, but not answering in time (device hung, wrong port)
`error`	Needs the operator: wrong password, missing or invalid setting. Not retried

Every module reports at least `online`, `offline` and `timeout`. The status usually carries a detail, such as "no route to host; retrying in 4 s".

What Tangle does on its own:

- **Reconnect.** An `offline` or `timeout` device is retried after 1 s, then 2 s, 4 s and so on, up to 30 s, until it answers. An `error` device is not retried until you change its settings.
- **Polling.** Most modules ask the device for its state every 10 s. One failed poll marks it `degraded`; two in a row count as a lost connection.
- **Actions while unreachable.** An action sent to an `offline` or `timeout` device fails at once as retryable, so core can try again, instead of waiting.
- **Action limits.** Actions to the same device run one at a time, in order. An action that takes more than 10 s fails and the next one runs. tangle-core also fails any action its module has not answered within 15 s.

Driver library

The **Driver library** on the device page finds a driver and installs it in one step:

- **Tangle and this computer**: the drivers bundled with Tangle (installed or not) and the ones made on this core with the Device creator.
- **My community drivers**: the drivers you published on the community repository.
- **Also search all community drivers**: every driver shared on community.tangle.systems, by manufacturer, product, name or username. Community drivers are named after their author (`louis/acme.widget.display`) and are not reviewed by the Tangle team. A driver that misbehaves can be reported.

Settings > Modules lists every module the core knows (installed and not installed), with **Install** and **Scan again**.

Install and remove modules

The desktop app includes the official modules, ready to install from the designer. To install a module from a folder (your own, or one you were given):

```sh
tangle module add ./barco.pulse.projector
tangle module add ./barco.pulse.projector --yes    # grant the permissions it asks for without asking
tangle module list                                  # --json for details
tangle module rollback barco.pulse.projector        # back to the previous version
tangle module remove barco.pulse.projector
```

When you add a module, Tangle:

- checks its manifest, that it supports this Tangle's contract version, and that its entry file exists;
- refuses a module that declares a category but whose folder is not named exactly after its id;
- shows the permissions it asks for (`network`, `serial`, `usb`, `midi`, `filesystem`) and asks you to confirm, also on updates that add permissions;
- warns that a module installed from a folder is unsigned, and that a community module is not reviewed by the Tangle team;
- checks that its runtime is installed: Node.js or Python at the version it asks for. Set `TANGLE_NODE` or `TANGLE_PYTHON` to use a specific interpreter. A Python module with a `requirements.txt` gets its own virtual environment, built from a bundled `wheels/` folder when present, so installing works offline.

Tangle keeps the active and the previous version of each module, so `rollback` can go back one step. A module used by a running project cannot be removed, and replacing the version a running project uses is refused; projects pick up a new version when they are next loaded.

Where modules live

	MACOS	WINDOWS	LINUX
Modules	`~/Library/Application Support/Tangle/modules`	`%ProgramData%\Tangle\modules`	`~/.local/share/tangle/modules`

Each module has its own folder, `modules/<id>/`, with one sub-folder per installed version and an `install.json` recording the active and previous versions. A module folder copied by hand into `modules/` also works; it is listed as local. When tangle-core runs as a boot service it uses the machine-wide folders (`/Library/Application Support/Tangle` on macOS, `/var/lib/tangle` on Linux). `TANGLE_HOME` moves everything into one folder.

Lock module versions per project

To run a project with exactly the same module versions on another machine:

```sh
tangle module lock -p show.yaml     # writes tangle.lock next to the project
tangle module sync -p show.yaml     # on the other machine: activates the locked versions
```

`sync` reports each module as ok, activated, not installed, or installed with files that differ from the lock. tangle-core also activates the locked versions when it starts a project and reports missing or altered ones.

Hardware attached to one machine

In a redundant cluster, networked devices are driven by whichever node is primary. Hardware plugged into one computer (a USB DMX interface, a serial port, an LTC sound card output) can only be driven from that computer. Give such a device a `node`:

```yaml
devices:
  - id: dmx-usb
    name: USB DMX
    moduleId: misc.usbdmx.lighting
    node: core-b
    config: { port: /dev/tty.usbserial-EN123456 }
```

Only `core-b` runs that device's module; the primary reaches it through `core-b`. If `core-b` is down, the device is offline. For anything that must survive a machine failure, prefer network gateways (Art-Net or sACN nodes, serial-over-IP adapters). Some modules require a node of a given operating system: `microsoft.psexec.computer` only runs on a Windows node.

How modules run

Each module runs as its own process, started by tangle-core:

- **One process per module.** Ten displays that use the same module share one process, which receives one `device.add` per display. A crash in one module cannot affect another module or core.
- **JSON over stdin and stdout.** Core and module exchange JSON-RPC 2.0 messages, one JSON object per line. The module logs to stderr. The protocol is described in Module SDK.
- **Supervision.** Core pings each module every few seconds. A module that crashes or misses 3 pings is restarted, after 1 s, then with a doubling delay up to 60 s. Five crashes within a minute are reported as an unstable module.
- **Latency.** A module whose average action latency goes above 500 ms is reported as slow.
- **Native modules.** Timecode, MIDI and DMX modules are native programs rather than Node.js or Python scripts, because they need steady timing. Modules that share hardware or sockets share one process: all DMX and

Art-Net timecode modules run in `tangle-dmx`, all MIDI modules in `tangle-midi`, PTP and NTP in `tangle-clock`. LTC runs alone in `tangle-ltc`.

> **Note:** Modules run as separate processes, but restricting what a module can reach based on its declared permissions (sandboxing) is not available yet. Permissions are shown for your consent.

GUIDES

Device catalog

Every official Tangle device module by category, with what it controls, its protocol and the setup each device needs.

This page lists the official device modules, grouped by category. Each table gives the module id, what it controls, how it talks to the device, and the setup needed on the device side. To add a device with one of these modules, see Devices.

Some modules include setup steps for the device side (a script to copy, a remote-access setting to enable). The designer runs them as setup assistants on the device page; see Setup assistants. The setup notes on this page say what each one does.

The catalog has 103 modules. Settings are listed by name; secrets (passwords, PINs, keys) are stored in the credential store, never in the project. Modules that talk RS-232 also accept a network serial port (`tcp://`, `rfc2217://` or `udp://` through an IP-to-serial gateway); see Network serial ports.

Timecode and clock references

Timecode modules are native programs built for steady timing. Inputs report `tangle.timecode` state (`timecode`, `fps`, `running`, `signal`) and can be chased by a timeline. Outputs generate timecode from start, stop and locate, which core uses to output a timeline's position. See Timelines.

ID	WHAT IT DOES	PROTOCOL	NOTABLE SETTINGS
`misc.ltc.timecode`	Reads or generates SMPTE LTC on a sound card channel. 24, 25, 29.97 drop-frame and 30 fps; input rate detected automatically	Audio (SMPTE 12M)	`mode` (`input` / `output`), `device` (sound card name or part of it; empty for the default), `channel`, `fps`, `levelDb` (output, default -12 dBFS), `dropoutMs`
`misc.mtc.timecode`	Reads (chases) or generates MIDI Time Code on a MIDI port of the computer	OS MIDI (CoreMIDI, Windows MIDI, ALSA)	`mode`, `port` (required, name or part of it), `fps`, `dropoutMs`
`misc.rtpmidi.timecode`	Reads or generates MTC over the network with RTP-MIDI (AppleMIDI sessions). No driver needed	UDP 5004/5005	`mode`, `session` (`initiator` or `listen`), `host`, `port`, `localPort`, `name`

ID	WHAT IT DOES	PROTOCOL	NOTABLE SETTINGS
`kissbox.midi.timecode`	Reads or generates MTC through a Kissbox MIDI interface in MIDI-over-UDP mode	Kissbox UDP (default 9813)	`mode`, `host` (required), `port`, `localPort`
`misc.artnet.timecode`	Receives ArtTimeCode from a desk or media server, or generates it	Art-Net UDP 6454	`direction`, `fps`, `host` (output; empty for broadcast), `timeoutMs`, `streamId`
`misc.ptp.timecode`	Listens to a PTPv2 (IEEE 1588) grandmaster and reports its identity, lock state and offset to this computer's clock	UDP 319/320 multicast	`domain`, `interface`, `lockThresholdUs`, `timeoutMs`
`misc.ntp.timecode`	Queries an NTP server and reports clock offset, round-trip delay and stratum	NTP UDP 123	`host` (required), `intervalMs` (16000; keep 16 s or more for public servers)

Setup notes:

- **LTC:** connect the LTC cable to a line input (reader) or line output (generator). The `audio-devices` setup step lists the sound card inputs and outputs. If no sound card matches, the device is `offline` with the list of devices that exist.
- **MTC and MSC:** plug in the MIDI interface first. The `midi-ports` setup step lists the ports. For virtual ports, enable the IAC Driver in Audio MIDI Setup on macOS, or use loopMIDI on Windows. The port list is checked every 10 s, so an unplugged interface goes `offline`.
- **RTP-MIDI:** as `initiator`, Tangle invites the peer at `host`:`port`; as `listen`, Tangle waits for invitations on `localPort`. Windows may ask to allow the UDP ports through the firewall on first use. Kissbox interfaces running RTP-MIDI firmware use `misc.rtpmidi.timecode`.
- **Kissbox MIDI:** the default ports and the datagram format are assumptions; check them in the Kissbox configuration tool before relying on this module.
- **PTP:** ports 319 and 320 need administrator rights: root on macOS and Linux (or `CAP_NET_BIND_SERVICE` on Linux); Windows usually allows them but may ask about the firewall. Without them the device shows `error` with that explanation. Accuracy is in the tens to hundreds of microseconds.
- **Status:** a timecode input goes `timeout` when no timecode arrives for `dropoutMs` (default 1000) and back `online` when it returns. Art-Net timecode inputs stay `online` and set `signal` to false instead.
- **Edition:** chasing or generating timecode, and PTP clock sync, require Tangle Pro or Ultimate. See Licensing.

Projectors

ID	WHAT IT CONTROLS	PROTOCOL	NOTABLE SETTINGS
`misc.pjlink.projector`	Projectors that speak PJLink (most Epson, NEC/Sharp, Panasonic, Sony, Canon, Hitachi, Optoma, ViewSonic): power, input, picture mute, lamp hours, errors	TCP 4352	`host`, `port`, `password` (secret)
`barco.pulse.projector`	Barco Pulse platform (UDX, UDM, F70/F80/F90, G62, HDX 4K, Threshold…): power, source, shutter, laser power, lens zoom/focus/shift, laser runtime, alarms	Pulse JSON-RPC, TCP 9090	`host`, `port`
`christie.native.projector`	Christie Boxer, Griffyn, M, HS, Crimson and D/HD/WU series: power, input, shutter, lamp hours, alarms	Christie ASCII protocol, TCP 3002	`host`, `port`
`panasonic.native.projector`	Panasonic PT-RZ, PT-RQ, PT-DZ, PT-MZ, PT-VMZ, PT-REQ/REZ and most networked models: power, input, shutter, freeze, light hours, errors	NTCONTROL, TCP 1024	`host`, `port`, `user` (default `admin1`), `password` (secret)

Setup notes:

- **PJLink:** turn on PJLink in the projector's network menu and note its password. Class 2 projectors are found by discovery; class 1 projectors do not answer the search.
- **Barco Pulse:** the JSON-RPC API is on by default; no password is needed for the functions this module uses.
- **Christie:** projectors with RS-232 only need an RS-232-to-IP adapter (a serial device server in TCP server mode); set `host` and `port` to the adapter. Projectors or adapters that ask for a login on connect are not supported yet.
- **Panasonic:** turn on network control ("Command control" / "NETWORK CONTROL") in the projector's network menu. Both protected (password) and unprotected modes work.
- Input, shutter and mute while the projector is off or warming up fail as retryable, so core tries again. Projector warnings and alarms show the device as `degraded`.

Media servers

All media server modules report `tangle.transport` (play, pause, stop, seek, load) for a main timeline or playback, plus a module-specific capability for any timeline by name.

ID	WHAT IT CONTROLS	PROTOCOL	NOTABLE SETTINGS
`dataton.watchout.mediaserver.v6`	WATCHOUT 6 (and 5) production or display computer: main and auxiliary timelines, control cues, load show, standby, online	Watchout TCP control (3040 production, 3039 display)	`host`, `mode` (`production` / `display`), `password` (secret), `timelines`
`dataton.watchout.mediaserver.v7`	WATCHOUT 7: play, pause, stop, jump to time or cue on any timeline	HTTP API on the Director, port 3019	`host`, `mainTimeline`
`disguise.d3.mediaserver`	disguise (d3 / Designer r17.4+): transports, cues, tracks, sections, machine health	disguise HTTP API on the director, port 80	`host` (the director), `transport`
`christie.pandorasbox.mediaserver`	Pandoras Box servers and Manager: sequence transport, cues, numeric device parameters	PB automation TCP, port 6211	`host`, `domain`, `sequence`, `fps`
`ioversal.vertex.mediaserver`	Vertex: playbacks, cues, fade to cue, go to time, any Vertex script line; timecode and cue feedback	External control TCP, port 50009	`host`, `playback`, `feedback`, `frameRate`
`hive.player.mediaserver`	Hive players (Beeblade family): playlists, clips, seek, output brightness	HTTP, port 80, base path `/api/v1`	`host`, `basePath`, `username`, `password` (secret)
`brightsign.player.mediaserver`	BrightSign players running the Tangle player script: play files or the playlist from the SD card, seek, volume	Text commands over TCP/UDP, port 5000	`host`, `port`, `playlist`, `loop`, `autoplay`, `volume`, `networkMode`
`modulopi.kinetic.mediaserver`	Modulo Kinetic: timeline transport and selection, tasks, any JSON-RPC method	JSON-RPC over WebSocket, port 8080	`host`, `port`, `path`, `timeline`
`resolume.arena.mediaserver`	Resolume Arena and Avenue 7+: clips, columns, clear, composition master, layer opacity and bypass, BPM, tap, resync	REST (8080) and OSC (7000)	`host`, `port`, `oscPort`, `trigger` (`osc` / `rest`)

ID	WHAT IT CONTROLS	PROTOCOL	NOTABLE SETTINGS
`vyv.photon.mediaserver`	VYV Photon (and Xenon): timeline play, pause, stop, seek, speed, free-run, LTC chase, cues, system commands	External TCP control, port 55503	`host`, `frameRate`, `playSpeed`
`avstumpfl.pixera.mediaserver`	PIXERA: play, pause, stop, locate, cues on any timeline. Reports every timeline's time as a named playhead (`tangle.playhead`)	JSON-RPC over TCP, port 1400	`host`, `mainTimeline`
`greenhippo.hippotizer.mediaserver`	Green Hippo Hippotizer V4: media and presets on layers, mix presets and levels with fades, timelines and cues, any pin, OSC to Multicontroller	REST API component, port 40512; optional OSC	`host`, `mix`, `timeline`, `oscPort`, `pollIntervalMs`

Setup notes:

- **Watchout 6:** allow control in Watchout (production: Preferences > Control; the display cluster always listens). Auxiliary timelines cannot be queried: their state is what Tangle last commanded.
- **Watchout 7:** `load` is not supported; load the show in Producer.
- **Pandoras Box:** the wire format was written without access to a real server. Check it on your Pandoras Box before using it in a show. Positions are converted to frames with the `fps` setting.
- **Hive:** the HTTP endpoints are an assumption made without Hive's API reference. Check them on a player before use.
- **BrightSign:** the player must run the Tangle player script. Run the `sd-card` configurator: it builds `autorun.brs` (the script) and `tangle-settings.json` (this player's port, playlist, volume and network) from the device's settings. Copy both to the root of the SD card next to the media files and start the player. With `networkMode: static`, the script sets the address at boot; use it as `host`. Discovery finds players running the script.
- **Modulo Kinetic:** add a "Web socket server" device in Kinetic Designer and use its port.
- **Resolume:** turn on the webserver (Preferences > Webserver, port 8080) and OSC input (Preferences > OSC, port 7000).
- **VYV Photon:** Photon does not report the timeline position; `positionMs` is the last position sought to.
- **Pixera:** in Settings > API, enable the TCP API with the JSON/TCP protocol, port 1400.

Game engines

ID	WHAT IT CONTROLS	PROTOCOL	SETUP
`epic.unreal.engine`	Unreal Engine 5 (and 4.26+), editor or packaged game: call Blueprint functions, set and watch properties, Remote Control presets	Remote Control API: HTTP 30010, WebSocket 30020	No Tangle plugin. Enable Epic's Remote Control API plugin. Configurator `remote-control`
`misc.godot.engine`	Godot 4 games and apps: fire named triggers, set named properties, receive game events	Tangle engine protocol, TCP 9420	Tangle Godot addon in the game. Configurator `plugin`
`unity.unity.engine`	Unity games and real-time apps: fire named triggers, set named properties, receive game events	Tangle engine protocol, TCP 9420	Tangle Unity package in the game. Configurator `plugin`

Each game event becomes a device event `game.<name>` that automations can react to. See Engine plugins for setting up each engine.

Players

ID	WHAT IT CONTROLS	PROTOCOL	NOTABLE SETTINGS
`video.lan.vlc.player`	VLC media player: play, pause, stop, seek, playlist (add, remove, move, sort), media from a folder on the VLC computer, open a file or URL, volume, loop, random, fullscreen. Reports its time as a playhead (`tangle.playhead`) and an optional preview stream (`tangle.preview`)	Lua HTTP interface (8080) or RC / telnet console	`host`, `protocol` (`http` / `rc`), `port`, `password` (secret), `mediaFolder`, `previewPort`, `fps`
`blackmagic.hyperdeck.player`	Blackmagic HyperDeck recorders: play (speed, loop, single clip), pause, stop, record named clips, clip and timecode locate, slot select; transport, timecode, clips and slots	HyperDeck Ethernet Protocol, TCP 9993	`host`, `port`
`qsc.core.player`	Audio Player components in a Q-SYS design: play, pause, stop, file selection, loop, status and position. One device can drive several players	QRC, TCP 1710	`host`, `user`, `pin` (secret), `players` (required), `mainPlayer`, `controlNames`

ID	WHAT IT CONTROLS	PROTOCOL	NOTABLE SETTINGS
`cockos.reaper.player`	Reaper as a playback or recording machine: transport, record, loop, markers and regions as cues, master volume	OSC (8000 in, 9000 feedback), optional web remote	`host`, `oscPort`, `feedbackPort`, `webPort`, `webUser`, `webPassword` (secret), `cues`, `allowedActions`

Setup notes:

- **VLC:** turn on the web interface (Preferences > All > Interface > Main interfaces: Web, with a Lua HTTP password), or run the setup assistant, which gives the command line for macOS, Windows and Linux. The **preview** assistant starts VLC with a small Motion JPEG stream on `previewPort` (8081), which a Camera control can show; it costs some CPU on the VLC computer.
- **Q-SYS players:** each player component needs Script Access set to External or All in Designer. `seek` is not available: the Audio Player has no control for it.
- **Reaper:** in Preferences > Control/OSC/web, add an **OSC (Open Sound Control)** surface with pattern config **Default**: Device IP is the Tangle computer, Device port is `feedbackPort` (9000), Local listen port is `oscPort` (8000); tick "Allow binding messages". Adding the **Web browser interface** (port 8080, set `webPort: 8080`) is recommended: the module then sees a Reaper that quit as `offline` and lists markers and regions as cues. Reaper actions run only if their ids are listed in `allowedActions`.

Computer control

Computer modules only run named commands written in the device's settings, never free text from a dashboard or cue. Each command can declare typed parameters (`string`, `number`, `integer`, `boolean`, `enum`), which are checked before anything is sent and are never pasted into the script.

ID	WHAT IT CONTROLS	PROTOCOL	NOTABLE SETTINGS
`misc.wol.computer`	Wakes computers with a Wake-on-LAN magic packet; optional check that the computer is up	UDP magic packet (port 9)	`mac` (required), `broadcast`, `password` (SecureOn, secret), `checkHost`, `checkPort`
`misc.ssh.computer`	macOS, Linux and Windows (OpenSSH) computers: shut down, restart, sleep, hibernate, start and stop apps, named commands; uptime and load	SSH	`host`, `username`, `os`, `password` / `privateKey` / `passphrase` (secrets), `keyFile`, `hostKeyCheck`, `sudo`, `commands`, `apps`

ID	WHAT IT CONTROLS	PROTOCOL	NOTABLE SETTINGS
`microsoft.powershell.computer`	Windows PCs over PowerShell Remoting: shut down, restart, sleep, hibernate, apps, named PowerShell commands; uptime, CPU, Windows version	WinRM, 5985 (5986 with HTTPS)	`host`, `username`, `password` (secret), `https`, `auth` (`ntlm`, `kerberos`, `basic`), `certValidation`, `commands`, `apps`
`microsoft.psexec.computer`	Windows PCs through Sysinternals PsExec: same functions as the PowerShell module, without remoting setup	SMB (445) and the `ADMIN$` share	`host`, `username`, `password` (secret), `psexecPath`, `statusCheck` (`tcp` / `psexec`), `commands`, `apps`

Setup notes:

- **Wake-on-LAN:** turn on Wake-on-LAN in the computer's BIOS/UEFI and network card settings (Windows: Device Manager > network card > "Wake on Magic Packet"; macOS: "Wake for network access"; Linux: `ethtool -s eth0 wol g`). Use the subnet's broadcast address when the computer is on another interface; routers usually do not forward broadcasts. Without `checkHost` the module cannot tell whether the computer is up. An asleep computer shows as `degraded` (not `offline`) so it can still receive the wake action. Turning a computer off is done with the SSH or PowerShell module.
- **SSH:** Windows needs the built-in OpenSSH Server feature. With `sudo: true`, give the account a sudoers rule without password for exactly the power commands. Host keys are trusted on first use by default and stored in the module's data folder; after reinstalling a computer, delete its entry or set `hostKey`. Requires Python 3.11+ and installs `asyncssh` in the module's environment.
- **PowerShell Remoting:** run the `enable-remoting` configurator. It generates a script for that PC (enable remoting, firewall rules, `LocalAccountTokenFilterPolicy` for local administrator accounts, Basic authentication when `auth: basic`, and a self-signed HTTPS listener when `https: true`). Run it in Windows PowerShell as administrator on the PC. The account must be an administrator. Over plain HTTP use `ntlm`; use `basic` with `https` only. Requires Python 3.11+ and installs `pywinrm`; for Kerberos (domain PCs) install `pywinrm[kerberos]`, which needs the system Kerberos libraries on macOS and Linux.
- **PsExec:** only runs on a Windows core node with PsExec installed; on macOS or Linux the device shows `error`. In a cluster, pin the device to a Windows node with `node` (see Devices). The password is passed on PsExec's command line, so other users of that machine can see it in the process list: run the core under its own account, or leave `username` empty to use the core's own Windows account.
- `tangle.power` `on` fails on the SSH, PowerShell and PsExec modules with a hint to use `misc.wol.computer`.

I/O

ID	WHAT IT CONTROLS	PROTOCOL	NOTABLE SETTINGS
`advantech.adam.io`	Advantech ADAM-6000 Ethernet I/O (6050, 6052, 6060, 6066, 6017, 6024, or a custom map for other Modbus TCP I/O): digital inputs with change events, digital and relay outputs, pulse, analog in and out	Modbus TCP 502	`host`, `unitId`, `model`, `map`, `aiRange`, `aoRange`, `registers`, `allowRawWrites`
`kissbox.io.io`	Kissbox I/O units, card by card: digital inputs with change events, relay and digital outputs, analog in and out	Kissbox UDP (9812)	`host`, `cards` (required, `{ slot, type }` with `di8`, `do8`, `ai8`, `ao8`)

Setup notes:

- Digital input changes send an `input.changed` device event for automations.
- **ADAM:** set `aiRange` and `aoRange` to the ranges configured in Advantech's utility (default 0 to 10 V). Raw Modbus writes are refused unless `allowRawWrites` is on.
- **Kissbox I/O:** the UDP wire format is provisional and has not been matched to Kissbox's protocol documentation. It will not talk to a real Kissbox until that is done.

Audio DSP

ID	WHAT IT CONTROLS	PROTOCOL	NOTABLE SETTINGS
`qsc.core.dsp`	Q-SYS Cores (and Designer emulation): the gains, mutes, selectors and triggers you list, snapshots, engine status. Changes made elsewhere are pushed back	QRC, TCP 1710	`host`, `user`, `pin` (secret), `controls`, `levelControl`, `muteControl`, `snapshots`
`biamp.tesira.dsp`	Biamp Tesira: the level, mute and other block controls you list (instance tag, attribute, index), presets, faults, raw TTP commands. Changes made elsewhere come back through subscriptions	Tesira Text Protocol, Telnet TCP 23 or RS-232	`host`, `serialPort`, `user`, `password` (secret), `controls`, `levelControl`, `muteControl`, `presets`

ID	WHAT IT CONTROLS	PROTOCOL	NOTABLE SETTINGS
`symetrix.composer.dsp`	Symetrix Prism, Edge, Radius NX and Jupiter: faders, mutes and other controls by controller number, presets, raw commands	Composer protocol, TCP or UDP 48631, or RS-232	`host`, `transport`, `serialPort`, `controls`, `levelControl`, `muteControl`, `presets`
`yamaha.rcp.dsp`	Yamaha DME7, MTX/MRX and CL/QL/TF/DM7/RIVAGE PM: faders, channel on/mutes, names and other parameters by address, scene and preset recall, device status	Yamaha remote control protocol, TCP 49280 (RS-232C on MTX/MRX)	`host`, `serialPort`, `controls`, `levelControl`, `muteControl`, `sceneCategory`, `presets`
`allenheath.ahm.dsp`	Allen & Heath AHM-64/32/16 and dLive: input, zone, control group, room, group, aux, matrix, main and DCA levels and mutes you list, presets and scenes, raw MIDI	TCP MIDI, port 51325	`host`, `model` (`ahm` / `dlive`), `midiChannel`, `controls`, `levelControl`, `muteControl`, `presets`

Only the controls listed in the settings can be changed from Tangle. Named controls come from Designer's Named Controls pane; component controls need the component's Script Access set to External or All. `tangle.level` maps 0..1 linearly onto the gain's `min` .. `max` dB (default -60..0), which is not the fader taper Q-SYS shows. A Core in Standby or Idle shows as `degraded`.

The other DSP modules work the same way: list the controls you want in `controls`, and pick the one that `tangle.level` and `tangle.mute` drive with `levelControl` and `muteControl`.

Lighting

The DMX modules are native and share one DMX engine (`tangle-dmx`): lighting devices, outputs and inputs on the same Tangle universe work together.

ID	WHAT IT DOES	PROTOCOL	NOTABLE SETTINGS
`misc.dmx.lighting`	Lighting devices (dimmers, LED pars, moving heads) described by lighting device profiles and patched to a universe and address. Also a recorder	DMX engine	`role` (`light` / `recorder`), `universe`, `address`, `profile`, `mode`, `recordSource`, `refreshHz`

ID	WHAT IT DOES	PROTOCOL	NOTABLE SETTINGS
`misc.artnet.lighting`	Sends a universe to an Art-Net node, or receives one from a lighting desk. Finds nodes with ArtPoll	Art-Net UDP 6454	`direction`, `universe`, `host` (empty for broadcast), `net` / `subnet` / `artnetUniverse`, `refreshHz` (40), `sync`, `nodeCheck`, `merge`
`misc.sacn.lighting`	Sends a universe as sACN (multicast or unicast), or receives one from desks, highest priority first	sACN (E1.31) UDP 5568	`direction`, `universe`, `sacnUniverse`, `host` (empty for multicast), `priority`, `sourceName`, `syncUniverse`, `merge`
`misc.usbdmx.lighting`	Sends a universe through a USB interface: Enttec DMX USB Pro and compatibles, or FTDI-based Open DMX style interfaces	USB serial	`port` (required), `interface` (`enttec-pro` / `open-dmx`), `universe`, `refreshHz`, `breakUs`, `mabUs`
`misc.dali.lighting`	DALI lighting through a Lunatone DALI-2 IoT gateway: lights by short address, group or broadcast switched, dimmed and sent to scenes; lamp and gear failures show as `degraded`	Gateway REST and WebSocket, port 80	`gateway`, `host`, `lights`, `mainLight`, `scenes`
`lutron.integration.lighting`	Lutron RadioRA 2/3, HomeWorks QS/QSX, Quantum and Athena: zones (dimmers, switches, shades) set and followed, keypad buttons as events, phantom buttons for scenes	Lutron Integration Protocol, Telnet TCP 23 with login, or RS-232	`host`, `user`, `password` (secret), `zones`, `mainZone`, `keypads`, `scenes`
`pharos.designer.lighting`	Pharos LPC, TPC, LPC X, VLC/VLC+ and EXT controllers: start, release, toggle, pause and resume timelines and scenes, group masters, triggers, with their states and temperatures	Controller HTTP API	`host`, `secure`, `user`, `password` (secret), `levelGroup`, `temperatureLimit`

A lighting device sends nothing by itself: add an output (Art-Net, sACN or USB) on the same Tangle universe. Several outputs may send the same universe.

Lighting device profiles

A lighting device profile says what each channel of a model means, per mode. Built-in profiles:

PROFILE	MODES
`generic-dimmer`	`8-bit` (1 ch), `16-bit` (2 ch)

PROFILE	MODES
`generic-rgb`	`RGB` (3 ch), `Dimmer RGB` (4 ch)
`generic-rgbw`	`RGBW` (4 ch), `Dimmer RGBW` (5 ch)
`generic-moving-head`	`Basic` (8 ch), `Compact` (6 ch)

The generic profiles are not real products: check the channel order against the device's manual.

For any other model, download its definition from open-fixture-library.org (the "OFL JSON" export, one file per model). Put it in the module's `profiles` folder and set `profile` to the file name without `.json`, or give an absolute path in `profile`. `mode` picks a mode by name or short name (`8ch`); the default is the first mode. Matrix models (pixel templates) are refused.

	PROFILES FOLDER
macOS	`~/Library/Application Support/Tangle/module-data/misc.dmx.lighting/data/profiles/`
Windows	`%ProgramData%\Tangle\module-data\misc.dmx.lighting\data\profiles\`
Linux	`~/.local/share/tangle/module-data/misc.dmx.lighting/data/profiles/`

Lighting devices accept `tangle.level.set` (0 to 1), `light.color.set` (`#rrggbb`, a colour name, or `red` / `green` / `blue` / `white` / `amber` / `uv`), `light.position.set` (`pan`, `tilt`) and `light.channel.set` (any channel by number or name, 0 to 255). Every action takes `fadeMs`.

How the patch and merge work:

- **Patch.** A device reserves `address` to `address + channels - 1` of its universe. A device that overlaps another gets status `error` naming the channels and the other device, and is not retried until you change its address.
- **Merge.** Every source (each light, raw channels, desk input with `merge`, take playback) is merged per channel. Intensity channels are HTP (highest wins); other patched channels such as pan, tilt and wheels are LTP (last change wins). On devices without a dimmer channel, `tangle.level` scales the colour.
- A light is `online` once patched, whether or not an output sends its universe yet.

Recording a lighting sequence

A `misc.dmx.lighting` device with `role: recorder` records one universe into named takes and plays them back:

- `recordSource: output` records what Tangle sends; `recordSource: input` records what a lighting desk sends to an Art-Net or sACN input device on the same universe.
- Actions: `dmx.recorder.record` (`take`), `stop`, `play` (`take`, `offsetMs`, `loop`), `select`, `list`, `delete`. A timeline cue or a dashboard button can start recording or playback.

- Playback merges with live control (HTP/LTP as above). When a take ends, its last look holds until `stop`.
- Takes are stored as files in the DMX process's data folder, keeping only the channels that change.

> **Note:** Recording into a DMX track on a timeline is not available yet. Use the recorder's takes and start them from cues.

Output status and setup

- **Art-Net:** a node never acknowledges DMX, so the module checks it with ArtPoll. With `nodeCheck: auto`, a unicast node that answers is `online` and goes `timeout` when it stops answering; a node that never answers stays `online` with a note that its status is not checked. On macOS, broadcasts to 255.255.255.255 only go out on the primary interface; a directed broadcast (such as `2.255.255.255`) is more reliable.
- **sACN:** receivers never answer, so the device is `online` with a note saying where it sends, and `offline` only when the network refuses to send.
- **USB DMX:** find the port name with `ls /dev/tty.*` on macOS or in Device Manager on Windows (`/dev/ttyUSB0` style on Linux). Some interfaces need the FTDI driver on macOS. An unplugged interface goes `offline` and comes back when plugged in again. Open DMX interfaces depend on the computer's timing; use 30 Hz if a receiver flickers. When the interface is plugged into one cluster node, set the device's `node`.
- **Edition:** the number of DMX universes (distinct `universe` values) follows your edition: Lite allows 2. See Licensing.

Lighting consoles

ID	WHAT IT CONTROLS	PROTOCOL	NOTABLE SETTINGS
`malighting.grandma.console.ma2`	grandMA2 consoles, onPC and NPUs: named commands, Go / Go back / Pause / Off / Goto Cue on executors, fader levels, macros, with the console's answer	Telnet, TCP 30000	`host`, `user`, `password` (secret), `commands`, `mainExecutor`, `levelExecutor`, `templates`, `allowRaw`
`malighting.grandma.console.ma3`	grandMA3 consoles, onPC and processing units: named commands, Go+ / Go- / Pause / Off / Goto Cue on sequences, executor faders and keys, macros	OSC (8000 in, 8001 feedback)	`host`, `port`, `localPort`, `prefix` (`gma3`), `commands`, `mainSequence`, `levelFader`, `heartbeat`, `allowRaw`

ID	WHAT IT CONTROLS	PROTOCOL	NOTABLE SETTINGS
`etc.eos.console`	ETC Eos family (Ti, Gio, Ion Xe, Element 2, Nomad, Eos software): Go and Stop/Back on cue lists, fire a cue, macros, channel intensity, submasters, named command lines, with the active and pending cue	OSC on TCP 3032 (length-prefixed or SLIP) or UDP	`host`, `transport`, `framing`, `localPort`, `user`, `mainList`, `levelSub`, `commands`, `allowRaw`
`chamsys.magicq.console`	ChamSys MagicQ consoles and MagicQ PC/Mac: playback go, stop, pause, release, level and jump to cue; page, channel intensity, execute items, named remote commands	OSC (with playback feedback) or ChamSys Remote Ethernet Protocol (UDP 6553)	`host`, `protocol`, `port`, `localPort`, `mainPlayback`, `levelPlayback`, `commands`
`behringer.x32.console`	Behringer X32 and Midas M32 mixers: faders, mutes, names and other parameters by strip or OSC address, scene recall, raw OSC. Console changes come back through `/xremote`	OSC, UDP 10023	`host`, `controls`, `levelControl`, `muteControl`, `scenes`

The grandMA, Eos and MagicQ modules only run the commands written in the settings and a few templates with checked parameters. Free text commands are refused unless `allowRaw` is on.

Setup notes:

- **grandMA2:** Setup > Console > Global Settings: enable Telnet "Login Enabled". Create a user for Tangle with the rights it needs, or use administrator.
- **grandMA3:** Menu > In & Out > OSC: add a line with Destination IP = the Tangle computer, Port = `localPort` (8001), the console's receive port = `port` (8000), Prefix = `prefix`, and turn on Receive, Receive Command, Send and Send Command. Enable input and output. With `heartbeat` on, a console that is off or not echoing shows as `timeout`, never `offline` (UDP cannot tell the difference).

Show control

ID	WHAT IT DOES	PROTOCOL	NOTABLE SETTINGS
`misc.mscontrol`	Sends MIDI Show Control commands (GO, STOP, RESUME, LOAD, SET, FIRE, ALL_OFF…) to lighting consoles, sound and show controllers (ETC Eos, QLab, grandMA…), and reports the MSC commands it receives as device events	MIDI SysEx over a MIDI port or RTP-MIDI	`transport` (`midi` / `rtpmidi`), `outputPort`, `inputPort`, `deviceId` (127 = all-call), `commandFormat`, `listenDeviceId`

Cue numbers, lists and paths are digits and dots (`12.5`). Received commands become `msc.received` device events. The `midi-ports` setup step lists the MIDI ports.

Displays

ID	WHAT IT CONTROLS	PROTOCOL	NOTABLE SETTINGS
`samsung.mdc.display`	Samsung commercial displays and LED walls: power, input, volume, mute, panel on/off, backlight, temperature, errors	MDC, TCP 1515 or RS-232	`host`, `serialPort`, `displayId`
`lg.signage.display`	LG commercial and signage displays: power, input, volume, mute, screen off, backlight, temperature, screen fault	LG RS-232C command set, TCP 9761 or RS-232	`host`, `serialPort`, `setId`
`sharp.nec.display`	Sharp/NEC MultiSync large-format displays: power, input, volume, mute, screen mute, backlight, temperature, self-diagnosis	NEC external control, TCP 7142 or RS-232	`host`, `serialPort`, `monitorId`
`philips.sicp.display`	Philips professional displays: power, input, volume, backlight, temperature, operating hours	SICP, TCP 5000 or RS-232	`host`, `serialPort`, `monitorId`
`sony.bravia.display`	Sony BRAVIA Professional displays: power, input, volume, mute, picture mute	Simple IP Control, TCP 20060 (with notifications), or RS-232C on BZ models	`host`, `protocol`, `serialPort`

More projectors: `epson.escvp.projector` (ESC/VP21 over ESC/VP.net TCP 3629 or RS-232), `sony.adcp.projector` (ADCP, TCP 53595 with or without the password challenge, or RS-232) and `sharp.nec.projector` (NEC binary commands, TCP 7142 or RS-232) give power with warm-up and cool-down, input, picture mute, freeze, light hours and errors. Like PJLink, their power actions report "warming up" until the projector is on, so an automation step can wait for it.

Video switchers, routers and AV over IP

ID	WHAT IT CONTROLS	PROTOCOL	NOTABLE SETTINGS
`extron.sis.switcher`	Extron switchers, matrices and scalers: routes (follow or breakaway), presets, video mute, volume and audio mute; front-panel changes reported	SIS, Telnet TCP 23 or RS-232	`host`, `password` (secret), `inputs`, `outputs`, `inputNames`, `outputNames`, `mainOutput`, `presets`
`kramer.p3000.switcher`	Kramer switchers and matrices: routes per layer, presets, video mute, level and mute	Protocol 3000, TCP 5000 or RS-232	`host`, `machineNumber`, `user`, `password` (secret), `inputs`, `outputs`, `mainOutput`, `presets`
`lightware.lw3.switcher`	Lightware matrices and switchers: video and audio routes, video mute, analog output volume and mute, with subscriptions	LW3, TCP 6107 or RS-232	`host`, `inputs`, `outputs`, `mainOutput`, `dialect`, `audioPort`
`blackmagic.videohub.switcher`	Blackmagic Videohub routers: routes, prepared routes taken together, locks, labels	Videohub Ethernet Protocol, TCP 9990	`host`, `mainOutput`
`aja.kumo.switcher`	AJA KUMO SDI routers: routes, names, locks, salvos as presets	KUMO HTTP API, port 80	`host`, `password` (secret), `mainDestination`
`blackmagic.atem.switcher`	Blackmagic ATEM production switchers: program and preview per M/E, cut and auto, aux outputs, macros	ATEM UDP protocol, port 9910	`host`, `mixEffect`
`analogway.livepremier.switcher`	Analog Way LivePremier (Aquilon, Zenith, Ascender, Pulse): screen, aux and master memories, TAKE, layer and aux sources	REST API, HTTP	`host`, `user`, `password` (secret), `screens`, `layers`, `mainScreen`, `memories`
`barco.eventmaster.switcher`	Barco Event Master (E2, S3, E3, EX, Encore 3): presets to preview or program, transitions, cues, layer and aux sources, freeze	JSON-RPC, HTTP port 9999	`host`, `operatorId`, `password` (secret), `mainDestination`, `mainLayer`

ID	WHAT IT CONTROLS	PROTOCOL	NOTABLE SETTINGS
`obs.obs.switcher`	OBS Studio 28+: program and preview scenes, transitions, streaming, recording, audio mute, any obs-websocket request	obs-websocket 5, port 4455	`host`, `password` (secret), `audioInput`
`vmix.vmix.switcher`	vMix: transitions, preview and program inputs, overlays, streaming, recording, outputs, master volume, any shortcut function, with tally	TCP API 8099 or HTTP 8088	`host`, `protocol`, `user`, `password` (secret)
`crestron.nvx.avoip`	Crestron DM NVX endpoints: route a decoder to a stream, local inputs, start and stop streams, HDMI output mute, stream and sync status	On-device REST API, HTTPS	`host`, `username`, `password` (secret)
`atlona.omnistream.avoip`	Atlona OmniStream encoders and decoders: route a decoder to an encoder stream, video and audio mute, link status, alarms	OmniStream JSON API over WebSocket	`host`, `username`, `password` (secret), `output`, `sources`
`justaddpower.jap.avoip`	Just Add Power 2G, 3G, Omega, Ultra, MaxColor: receivers switch channel (unified or per video, audio, USB), mute or turn off video; transmitters turn their stream on and off	Telnet TCP 23 or RS-232	`host`, `role`, `channelMode`, `channels`
`audinate.dante.routing`	Dante subscriptions on any brand's Dante device: subscribe receive channels to `channel@device`, unsubscribe, subscription status	Dante control protocol (ARC, UDP 4440); found by name over mDNS	`name`, `host`, `sources`

Setup notes:

- **ATEM:** the protocol was written after the open-source atem-connection library, not from Blackmagic documentation.
- **Dante:** uses the reverse-engineered control protocol documented by the netaudio project. Check it with your devices before relying on it in a show.

Cameras

ID	WHAT IT CONTROLS	PROTOCOL	NOTABLE SETTINGS
`misc.visca.camera`	PTZ cameras over VISCA: Sony VISCA over IP, plain VISCA over TCP or UDP (PTZOptics, Lumens, Marshall, AVer, BirdDog) or RS-232. Power, pan/tilt, zoom, focus, presets	UDP 52381, TCP/UDP or RS-232	`transport`, `host`, `port`, `address`, `presets`, `streamUrl`, `snapshotUrl`
`panasonic.aw.camera`	Panasonic AW-UE and AW-HE PTZ cameras: power, pan/tilt, zoom, focus, presets, tally, head errors	HTTP CGI, port 80	`host`, `user`, `password` (secret), `presets`, `streamUrl`, `snapshotUrl`

The modules control the camera; the picture comes from the camera's own MJPEG, snapshot or HLS output, set in `streamUrl` or `snapshotUrl` for the dashboard Camera control. Panasonic preset numbers start at 1 in Tangle (preset 1 is the camera's code 00).

Conference rooms and presentation

ID	WHAT IT CONTROLS	PROTOCOL	NOTABLE SETTINGS
`cisco.roomos.codec`	Cisco Room, Board and Desk devices: dial, hang up, answer, microphone mute, volume, standby, presentation, selfview, camera presets	xAPI over WebSocket	`host`, `user`, `password` (secret), `presentationSource`
`poly.g7500.codec`	Poly G7500, Studio X30/X50/X52/X70/X72, G62: microphone and camera mute, volume, dial, hang up, reboot; sleep and wake	Poly REST API (HTTPS) and command line (telnet 24)	`host`, `user`, `password` (secret), `dialType`, `callRate`
`zoom.rooms.codec`	Zoom Rooms: join, start, leave and end meetings, answer or decline calls, microphone and camera mute, volume, HDMI sharing	Zoom Rooms Control System API, SSH port 2244	`host`, `user`, `password` (secret)
`barco.clickshare.presentation`	Barco ClickShare base units: sharing status, buttons, screens, standby, health, wake, reboot	ClickShare REST API v2, HTTPS 4003	`host`, `user`, `password` (secret)

ID	WHAT IT CONTROLS	PROTOCOL	NOTABLE SETTINGS
`mersive.solstice.presentation`	Mersive Solstice Pods: users, posts, sharing, end session, clear posts, bulletin and emergency messages, restart	OpenControl API v1, HTTP or HTTPS	`host`, `password` (secret)
`misc.calendar.scheduling`	Room calendars in Microsoft 365 or Google Workspace: meeting in progress, next meeting, free/busy, with `meeting.starting`, `meeting.started` and `meeting.ended` events for automations	Microsoft Graph or Google Calendar	`provider`, `room`, `tenantId`, `clientId`, `clientSecret` (secret), `serviceAccountKey` (secret), `leadMinutes`

Microphones and amplifiers

ID	WHAT IT CONTROLS	PROTOCOL	NOTABLE SETTINGS
`shure.mxa.microphone`	Shure MXA ceiling and table arrays, P300, ANI: device mute and LED, channel gains and mutes, presets, identify, meters	Shure command strings, TCP 2202	`host`, `levelChannel`, `presetNames`, `meterRateMs`
`shure.wireless.microphone`	Shure ULX-D, QLX-D, SLX-D, Axient Digital: per-channel name, mute, gain, battery, RF; low battery as events	Shure command strings, TCP 2202	`host`, `family`, `channels`, `lowBatteryBars`, `lowBatteryMinutes`
`sennheiser.ssc.microphone`	Sennheiser EW-DX, SpeechLine, TeamConnect Ceiling: mute, gain or level, identify, names, battery, RF	SSCv2 (HTTPS with Server-Sent Events) or SSC v1 (UDP 45)	`host`, `protocol`, `password` (secret), `channels`
`crown.hiqnet.amplifier`	Crown DCi, CDi DriveCore, XLC, XTi 2, I-Tech HD: power, mute, level and fault parameters you list	HiQnet, TCP 3804	`host`, `node`, `controls`, `powerControl`, `muteControl`, `levelControl`
`powersoft.armonia.amplifier`	Powersoft X and T series, Quattrocanali, Duecanali, Ottocanali: standby, channel gain and mute, presets, alarms	Powersoft UDP protocol, port 1234	`host`, `channelNames`, `presets`

Power

ID	WHAT IT CONTROLS	PROTOCOL	NOTABLE SETTINGS
`middleatlantic.racklink.power`	Middle Atlantic RackLink PDUs: outlets on, off, cycle; sequencing; voltage, current, power, temperature; EPO and surge faults as `degraded`	RackLink protocol, TCP 60000 with login, or RS-232	`host`, `user`, `password` (secret, required), `powerOutlets`
`wattbox.ovrc.power`	SnapAV WattBox: outlets on, off, toggle, reset; voltage, current, power; UPS battery (`degraded` on battery)	WattBox Integration Protocol, Telnet 23	`host`, `user`, `password` (secret), `powerOutlets`

Building and network

ID	WHAT IT CONTROLS	PROTOCOL	NOTABLE SETTINGS
`misc.bacnet.building`	BACnet/IP controllers: the analog, binary and multi-state points you list, followed with COV (or polled) and commanded at a priority; Who-Is discovery	BACnet/IP, UDP 47808	`host`, `deviceInstance`, `priority`, `points`, `powerPoint`, `levelPoint`
`misc.knx.building`	KNX through a KNXnet/IP interface or router: group addresses with their DPT read, written and followed	KNXnet/IP tunnelling (UDP 3671) or routing (224.0.23.12)	`mode`, `host`, `points`, `powerPoint`, `levelPoint`
`misc.modbus.building`	Modbus TCP or RTU devices: coils, inputs and registers you list (bool, 16/32-bit, float32, word order, scale), polled and written	Modbus TCP 502, or RTU on a serial port or gateway	`host`, `serialPort`, `unitId`, `points`, `powerPoint`, `levelPoint`
`misc.snmp.network`	Any SNMP v1/v2c agent: UPSes (UPS-MIB), PDUs, switches; reads OIDs into state, switches outlets, traps as events	SNMP, UDP 161 (traps on `trapPort`)	`host`, `version`, `community` (secret), `values`, `outlets`, `ups`
`misc.ping.network`	Anything with an IP address: answers to ICMP ping or a TCP port, round-trip time, packet loss, last seen. Puts it on dashboards and under Sentinel alerts	ICMP or TCP	`host`, `method`, `tcpPort`, `intervalMs`, `failures`

ID	WHAT IT CONTROLS	PROTOCOL	NOTABLE SETTINGS
`global cache.i tach.ga teway`	Global Caché iTach and Flex: IR codes to IR-only devices, relays, contact sensors; standard power, mute and input for IR-only TVs and players	iTach API, TCP 4998	`host`, `irPort`, `irCodes`, `relays`, `sensors`, `inputs`

For a Modbus device you use often, make a driver with named registers in the device creator instead of listing points on every device.

Generic devices

When no module exists for a device, use one of these and write its commands in the device settings, or make a driver in the device creator.

ID	WHAT IT CONTROLS	PROTOCOL	NOTABLE SETTINGS
`misc.tc p.generi c`	Any device that takes text or binary commands over TCP	TCP	`host`, `port`, `commands`, `feedback`, `bindings`, `poll`
`misc.ud p.generi c`	Any device that takes UDP datagrams. Set a `poll` command that gets an answer, so an absent device shows `timeout`	UDP	`host`, `port`, `localPort`, `broadcast`, `commands`, `feedback`, `poll`
`misc.se rial.gen eric`	Any RS-232, RS-422 or RS-485 device, on a local port or an IP-to-serial gateway	Serial	`path`, `baudRate`, `parity`, `commands`, `feedback`, `bindings`
`misc.ht tp.gener ic`	Any device with an HTTP or REST API: each command is a request, feedback rules read the answers	HTTP, HTTPS	`baseUrl`, `username`, `password` (secret), `headers`, `commands`, `feedback`

`commands` are named, with placeholders for parameters; `bindings` map standard capabilities (power, mute, input, level, preset) to commands; `feedback` rules read answers into state.

Listeners and libraries

Two modules are not devices:

ID	WHAT IT DOES
`tangle.builtin.listener`	The built-in listeners: WebSocket, HTTP, TCP, UDP (text, hex, JSON, OSC) and serial, so other systems can trigger Tangle
`tangle.image.library`	The `image` function library: resize, crop, rotate, adjust and combine images, read their colours

GUIDES

Timelines

Build timecode timelines and cue lists, run the transport, chase incoming timecode and generate timecode outputs.

A timeline fires cues at set moments. Each cue sends a device action or an intent. Tangle core plays timelines on a dedicated timing thread, so cues fire on time even when the network is busy, and the same timeline runs in lockstep on every node of a redundant cluster.

Modes

Every timeline has a `mode`:

MODE	WHAT IT DOES
`timecode`	Free-running. After `play`, cues fire at their position (`atMs`) on the timeline.
`cuelist`	Waits for GO. Each `go` fires the next cue, in time order, and the timeline waits again.

Use `timecode` for shows that run to a clock: a pre-programmed sequence, a show synced to video or audio, a looping attract loop. Use `cuelist` when an operator calls each step, as in theatre.

In both modes the state reports the last cue fired (`lastCueId`) and the next one (`nextCueId`).

Tracks and cues

A timeline holds tracks, and each track holds cues. Tracks group cues for editing (lights, video, sound). A muted track does not fire its cues.

```json
{
  "id": "show",
  "name": "Main show",
  "mode": "timecode",
  "frameRate": "25",
  "durationMs": 120000,
  "loop": false,
  "tracks": [
    {
      "id": "video",
      "name": "Video",
      "cues": [
        {
          "id": "c1",
          "name": "Projectors on",
          "atMs": 0,
          "payload": { "kind": "intent", "intent": { "name": "power.on", "target": { "kind":
"room", "id": "room-a" } } }
        },
        {
          "id": "c2",
          "name": "Start clip",
          "atMs": 2520,
          "preRollMs": 200,
          "payload": { "kind": "action", "deviceId": "media-1", "capability":
"tangle.transport", "action": "play" }
        }
      ]
    }
  ]
}
```

Timeline fields:

FIELD	REQUIRED	MEANING
`id`, `name`	yes	Identifier and display name.
`mode`	yes	`timecode` or `cuelist`.
`tracks`	yes	Tracks and their cues.
`clock`	no	What drives the timeline. Default `{ "kind": "internal" }`. See Clock sources.
`frameRate`	no	`24`, `25`, `29.97df`, `30`, `50` or `60`. Used for frame snapping in the designer and for timecode.
`durationMs`	no	Length of the timeline. At the end it stops, or starts again from 0 if `loop` is true.
`loop`	no	Start again from 0 at `durationMs`.

FIELD	REQUIRED	MEANING
`timecodeOutputs`	no	Devices that generate timecode following this timeline. See Timecode generation.

Cue fields:

FIELD	REQUIRED	MEANING
`id`	yes	Unique within the timeline.
`atMs`	yes	Position from the start of the timeline, in milliseconds.
`payload`	yes	`{ "kind": "intent", "intent": {...} }`, `{ "kind": "action", "deviceId", "capability", "action", "params" }`, or `{ "kind": "comment", "text" }`: a note that plays nothing. A cue whose device was deleted with **Keep timeline cues as comments** becomes a comment cue that remembers the device and the action, so **Point at device...** can turn it back into a cue.
`name`	no	Display name.
`preRollMs`	no	Send the cue this many milliseconds early to absorb device latency. The device still gets the exact intended time as its execution time.
`enabled`	no	`false` skips the cue. Default `true`.

Cues fire as intents with source `timeline`, so intent priorities and holds apply. Timelines use priority 50.

> **Note:** A timeline with no `durationMs` keeps playing after its last cue until you stop it.

Clock sources

`CLOCK.KIND`	STATUS
`internal`	Default. Core's own clock.
`timecode`	Chase a timecode input device, or the playhead of a media player or server. See Timecode chase.
`wallclock`	Follow the time of day. See Time of day.
`ptp`	Run at the rate of a PTP grandmaster. See PTP.
`ltc`, `mtc`	Older forms, not played. Use `timecode` with an LTC or MTC input device instead.

Transport

The transport operations are:

OPERATION	TIMECODE MODE	CUE LIST MODE
`play`	Starts playing from the current position.	Arms the list: status `waiting` until GO.
`pause`	Holds the current position.	Pauses the list.
`stop`	Returns to 0 and forgets which cues fired.	Returns to the first cue.
`seek`	Jumps to `positionMs`. Cues skipped over do not fire. A stopped timeline becomes paused at the new position.	Moves to `positionMs`; cues before it count as played.
`go`	Not used (returns `unsupported`).	Fires the next cue. After the last cue the list stops.
loop	Set `loop: true`. At `durationMs` the timeline starts again from 0, re-anchored on the exact end time so loops never drift.	Not used.

After a pause, `play` resumes from the same position and no cue before it fires again. A cue that sits exactly at the new position of a seek fires right away.

Timeline status is one of `stopped`, `playing`, `paused` or `waiting` (a cue list waiting for GO, or a chasing timeline waiting for timecode).

Sending transport commands

Transport needs a token with the `control` scope.

REST:

```sh
curl -X POST http://127.0.0.1:7340/api/v1/timelines/show/transport \
  -H "Authorization: Bearer $TOKEN" -H "content-type: application/json" \
  -d '{"op":"seek","positionMs":30000}'
```

The answer is the new timeline state. `GET /api/v1/state/timelines` returns the state of every timeline.

WebSocket:

```json
{ "v": 1, "id": "42", "type": "timeline", "control": { "op": "go", "timelineId": "cues" } }
```

MCP: `control_timeline` and `get_timeline_states`. An assistant whose token lacks the `live` scope cannot change anything while a timeline plays. See MCP.

Automations can also drive the transport with a `timeline` action. See Automations.

Timeline state

Core publishes a `timeline.state` event (topic `timeline.state:<timelineId>`) on every change and 10 times per second while playing:

```JSON
{
  "timelineId": "show",
  "status": "playing",
  "positionMs": 31250.0,
  "sampledAt": 1760362800000,
  "lastCueId": "c7",
  "nextCueId": "c8"
}
```

`sampledAt` is the wall-clock time at which `positionMs` was measured, so a client can move its playhead smoothly between events. Chasing timelines also report `chasing` and `clockLost`, time-of-day timelines `chasing`, and PTP timelines `clockLost`.

Editing in the designer

Open **Timelines** in the designer (`/designer/timelines`) and choose **New timeline**.

1. Enter a name and choose the mode: **Timecode** or **Cue list (GO)**.
2. Set the clock, frame rate, duration and loop as needed.
3. Add cues:
   - Timecode mode shows tracks as rows on a zoomable time ruler. Double-click a track to add a cue, and drag a cue to move it. Cues snap to frames, or to 100 ms when no frame rate is set.
   - Cue list mode shows cues as an ordered list in GO order. Use the up and down arrows to reorder them.
4. Select a cue to edit its name, time, pre-roll, enabled flag and what it does: **A device action** or **An intent**.
5. Choose **Create timeline** or **Save**.

Use **Add track** to add tracks. Each track has a name, a mute switch and a delete button.

Live control

The transport buttons (Play, Pause, Stop, or GO for a cue list) and seeking by clicking the ruler stay disabled until you tick **Live control** for the session. This prevents an accidental click from starting or moving a running show.

Editing while a timeline plays

While a timeline plays, the editor is read-only. Tick **Allow live editing** to change it anyway: changes apply to the running show. Through the REST API the same rule applies: a change to a playing timeline is refused with `409`

`timeline_running` unless the request adds `?allowLive=true`.

When a timeline is edited during playback, cues keep their "already fired" flag by id. A new cue placed before the current position counts as played.

Timecode chase

A timeline can follow incoming timecode. The input is any device with the `tangle.timecode` capability (LTC, MTC, RTP-MIDI, Kissbox or Art-Net timecode modules), or a media player or server with the `tangle.playhead` capability: VLC follows the clip it plays, Pixera has one playhead per timeline. Chasing and generating timecode need Tangle Pro or Ultimate (see Licensing).

```json
{
  "id": "show", "name": "Show", "mode": "timecode", "frameRate": "25",
  "clock": { "kind": "timecode", "deviceId": "ltc-in", "offset": "01:00:00:00", "freewheelMs":
500 },
  "tracks": []
}
```

CLOCK FIELD	MEANING
`deviceId`	The timecode input device, or the media player or server.
`playhead`	For a device with several playheads: which one to follow, for example the Pixera timeline `Pre-show`. Empty: the device's main playhead.
`offset`	The timecode at which the timeline is at 0. Default `00:00:00:00`. Format `HH:MM:SS:FF` (`;` before the frames for drop-frame).
`freewheelMs`	How long the timeline keeps playing when frames stop arriving, before it waits. Default 500, maximum 10000.

How chasing behaves:

- **Arming.** `play` arms the timeline (`chasing: true`). Its status is `waiting` until timecode runs. `pause` and `stop` disarm it.
- **Following.** Cues fire on the source's time. Core measures from the earliest-arriving frames, so pipe and network jitter do not move cues. Small drift moves the timeline smoothly.
- **Jumps.** When the source jumps (two frames that agree on a new position), the timeline relocates there. Cues before the new position count as played.
- **Parked source.** A source that is not running positions the timeline at its timecode.
- **Before the start.** Timecode earlier than `offset` holds the timeline at 0.

- **Dropout.** Without frames the timeline keeps playing for `freewheelMs`, then waits where the timecode stopped (status `paused`, still armed).
- **Lost signal.** When the input reports no signal, or goes offline or times out, the state shows `clockLost: true`. The designer shows a "no timecode" badge.
- **End.** At `durationMs` a chased timeline waits instead of stopping or looping. The source decides what comes next.

To set this up in the designer, choose **Clock: Chase timecode (LTC, MTC, Art-Net, media servers)**, then pick the input device (and, for a media server, which of its timelines), the **Start timecode** and **Freewheel (ms)**.

Automations and expressions read playheads too: `device.timecode("pixera", "Pre-show")`, `device.position("vlc")`. Dashboards show them with the Clock control.

Time of day

A timeline with a `wallclock` clock uses the time of day as its position: a cue at `atMs` 32400000 (09:00:00) fires at 9 am every day. Use it for building and room schedules.

```JSON
{
  "id": "building-day",
  "name": "Building day",
  "mode": "timecode",
  "clock": { "kind": "wallclock", "timezone": "America/Montreal" },
  "tracks": []
}
```

`timezone` is an IANA zone name. Leave it empty to use the zone of the computer running core. An unknown zone makes `play` return an `unsupported` error naming it.

How it behaves:

- **Arming.** `play` arms the timeline (`chasing: true`) and moves it to the current time of day. Cues earlier in the day do not fire. `pause` and `stop` disarm it. `seek` returns `unsupported`.
- **Every day.** At midnight the timeline starts again from 0, and the day's cues fire again. `durationMs` and `loop` are ignored.
- **Clock changes.** Core checks the timeline against the clock every second and moves it when they differ by more than 5 ms. When the clock moves forward by up to 65 minutes (a daylight-saving change), the cues in between fire at once, in order. A larger jump skips them. When the clock moves back, cues that already fired do not fire again.
- **After a restart.** A time-of-day timeline does not arm by itself. To have it run whenever core starts, add an automation with the **Core starts** trigger that plays it.

In the designer, choose **Clock: Time of day** and set the **Time zone**.

PTP

A timeline with a `ptp` clock plays like the internal clock (play, pause, seek, loop), but at the rate of a PTP grandmaster. Over a long show it stays in step with media servers and other systems locked to the same grandmaster, instead of drifting with the computer's own clock (typically up to a few ms per minute).

```json
"clock": { "kind": "ptp", "deviceId": "ptp-gm" }
```

CLOCK FIELD	MEANING
`deviceId`	The PTP clock reference device (`misc.ptp.timecode`) to follow.
`domain`	When `deviceId` is not set: follow the first PTP clock reference device on this domain. Default 0.

How it behaves:

- **Following.** Core reads the device's offset to the grandmaster (a few times a second). When the timeline drifts more than 1 ms from PTP time, it moves by the drift. Cues fire on PTP time.
- **Steps.** A change of 20 ms or more at once is a new reference (another grandmaster, or the computer's clock set by hand), not drift: the timeline keeps its position.
- **Not locked.** When the device is not locked, offline or timed out, the timeline keeps playing on the computer's clock and the state shows `clockLost: true`. The designer shows a "PTP not locked" badge.
- **Redundancy.** Leave the PTP device unpinned (no `node`), so it runs on the primary core.

In the designer, choose **Clock: PTP** and pick the **PTP reference**. PTP timelines need Tangle Pro or Ultimate.

Timecode generation

`timecodeOutputs` lists timecode output devices (`tangle.timecode`) that follow the timeline:

```json
"timecodeOutputs": [
  { "deviceId": "mtc-out", "offset": "01:00:00:00" },
  { "deviceId": "artnet-tc" }
]
```

`offset` is the timecode sent when the timeline is at 0 (default `00:00:00:00`).

Every transport change (play, pause, seek, loop, chase jump) is sent to each output:

- While playing: `start` with the timecode of the next frame boundary about 40 ms ahead, and the exact time to start.
- Otherwise: `stop`, parked on the current position.

Outputs report their own timecode. An output that drifts more than 2 frames (for example a sound card clock) is sent the timecode again.

A chased timeline with outputs works as a converter, for example LTC in, MTC and Art-Net timecode out.

In the designer, timecode outputs appear under **Generate timecode on** for timecode-mode timelines. Tick each device and enter its start timecode.

Frame rates

Timecode is counted at the rate the device reports (its `fps` state). When a device does not report one, core uses the timeline's `frameRate`, then 25. `29.97df` is drop-frame: frame numbers 0 and 1 are skipped every minute except every tenth, so the count stays in step with real time.

Behaviour under redundancy

In a redundant cluster every node plays every timeline in lockstep on a shared cluster clock. Only the primary sends cues to devices; backups fire the same cues in shadow and record them in their cue log.

- Transport commands go to the primary. A backup answers `409 not_primary`.
- If the primary fails, the new primary continues from its own position. It sends cues that the old primary may not have sent, if they were due in the last 2 seconds. No cue fires twice.
- Only the primary chases timecode and drives timecode outputs. Backups replay its decisions, so their cue logs match. After a failover, the new primary sends the transport to every output again and starts its own chase.

Limits by edition

	LITE	PRO	ULTIMATE
Timelines and cue lists	2 in total	unlimited	unlimited
Timecode chase and generation	no	yes	yes

A project that goes over its edition is refused at load, with a message naming the limit. See Licensing.

GUIDES

Automations

Step lists that run on triggers, from buttons, listeners or by hand - with conditions, control flow, variables, expressions, library functions, run modes and a run log.

An automation is a named list of steps. It runs when one of its triggers fires and all of its conditions hold, and also whenever something runs it: a dashboard button, a listener rule, another automation, the API or MCP. Automations are stored in the project (`project.automations`) and run inside tangle-core, on the primary node of a cluster or on a standalone node.

Since contract 0.2.30, automations also hold what macros used to (params, steps without triggers): macros in older projects are moved into automations when the project loads.

Example

At 08:00 on weekdays, if the projector is online, recall the presentation scene in Room A:

```json
{
  "id": "morning",
  "name": "Morning",
  "triggers": [{ "kind": "time", "at": "08:00", "days": ["mon", "tue", "wed", "thu", "fri"]
}],
  "conditions": [{ "kind": "status", "deviceId": "projector-1", "status": "online" }],
  "steps": [
    {
      "kind": "intent",
      "intent": { "name": "scene.recall", "target": { "kind": "room", "id": "room-a" },
"params": { "scene": "presentation" } }
    }
  ],
  "mode": "single"
}
```

Automation fields:

FIELD	REQUIRED	MEANING
`id`, `name`	yes	Identifier and display name.
`steps`	no	What it does, in order. `actions`, the name before contract 0.2.30, is still read.
`triggers`	no	Any one starts a run. Without triggers it runs only when something runs it.
`conditions`	no	All must hold for the run to go ahead. None: always runs.

FIELD	REQUIRED	MEANING
`params`	no	Inputs, read as `{{name}}` in steps: `name`, optional `label` and `default`.
`mode`	no	What a trigger does while a run is in progress. Default `single`.
`continueOnError`	no	Keep running the next steps after one fails. The run ends as `partial`.
`priority`	no	0 to 100, applied to the intents the automation sends. Higher wins on conflict. Default 50, the same as timelines.
`enabled`	no	Default `true`. `false`: its triggers do not start it, but it still runs by hand, from buttons and from other steps.
`folder`	no	Folder in the automation list, for example `Lobby/Evening`.
`description`	no	Free text.

Editing in the designer

Right-click **Automations** in the project tree and choose **New automation**, or open one from the tree.

The editor has the **step table** on the left: one row per step with its number, an **On** checkbox (a step turned off is skipped, with the steps inside it), its type, target, value and what it waits for. **+ Add step** opens the step picker, grouped as Device, Flow, Data, Run and Notes. Keyboard: Insert adds a step after the selected row, Alt+↑/↓ moves it, Space turns it off, Ctrl+D duplicates and Ctrl+Z undoes.

On the right:

- **Step N** edits the selected step.
- **Automation** holds the name, id, description, folder, **Params**, **Run when (any of)** (triggers), **Only if (all of)** (conditions) and **Behaviour**: the mode ("When it starts again while running"), **Continue after a failed step** and **Priority (0–100)**.
- **Runs** has **Run now** (with **Ignore the conditions** when there are some), **Stop**, and the run log, which updates live.

The line above the table sums the automation up, for example "10:00 every day → Power on Gallery, then wait until done (at most 2 min), Media player: transport play". Click **Create automation**, then **Save** for later changes. Runs use the saved version: save first to try your changes.

Tutorial: a gallery installation builds two automations step by step.

Triggers (Run when)

KIND	IN THE DESIGNER	FIELDS	FIRES WHEN
`time`	Time of day	`at` (`HH:MM`, 24 h), optional `days`	The node's local clock reaches that minute. Every day when `days` is absent.
`deviceState`	Device state changes	`deviceId`, `capability`, `key`, optional `to`, `above`, `below`, `unit`	A state value changes. With `to`, `above` or `below`, only when the new value starts matching. `unit` converts the thresholds (`above: 26, unit: "°C"`).
`deviceEvent`	Device event (button…)	`deviceId`, `name`	The device emits the event, for example `button.pressed`.
`deviceStatus`	Device goes online/offline	`deviceId`, `status`	The device's status becomes this one (`online`, `offline`, `timeout`, `degraded`, `error`, `connecting`).
`timeline`	Timeline starts/stops/reaches a cue	`timelineId`, `on`, `cueId` for `on: "cue"`	`started`, `stopped` (end reached or stop) or `cue` (reaches `cueId`).
`variable`	A project variable changes	optional `variable`, `to`, `above`, `below`	A project variable changes. Without `variable`: any of them. The run gets `{{variable}}`, `{{value}}` and `{{previous}}`.
`control`	A dashboard control is clicked or changes	optional `controlId`, `on`, `to`, `above`, `below`	Someone operates a dashboard control, or a step or the API changes its value. `on`: `change` (default), `click`, `press`, `release`, `hold` or `any`. A button fires on click. The run gets `{{control}}`, `{{value}}`, `{{previous}}` and `{{event}}`.
`listener`	A listener message matches	optional `listener`, `rule`	A rule of an inbound listener matches a message. The run gets the rule's captures as `{{name}}`. See Listeners.
`startup`	Core starts	none	This node starts running automations: core starts, or a backup is promoted to primary.
`manual`	Only by hand	none	Never fires by itself. Marks an automation meant to be run by hand. Any automation can be run by hand.

Weekdays are `mon`, `tue`, `wed`, `thu`, `fri`, `sat`, `sun`. In the designer each trigger has a **Triggers on** checkbox to switch it off without deleting it.

> **Note:** A minute that passes while core is down, or while the node is a backup, is not replayed. A `time` trigger at 08:00 does not fire at 08:05 because core restarted at 08:03.

Conditions (Only if)

KIND	FIELDS	HOLDS WHEN
`state`	`deviceId`, `capability`, `key`, optional `equals`, `above`, `below`, `unit`	The current value passes every test given.
`status`	`deviceId`, `status`	The device's current status is this one.
`time`	optional `after`, `before`, `days`	Local time is in the window. `after` later than `before` spans midnight (for example 19:00 to 07:00).
`timeline`	`timelineId`, `status`	The timeline's status is `stopped`, `playing`, `paused` or `waiting`.
`variable`	`variable`, optional `equals`, `notEquals`, `above`, `below`	A project variable, param or run variable passes every test. A missing one is `null`.
`control`	`controlId`, optional `property`, `equals`, `notEquals`, `above`, `below`	A dashboard control's value (or another property: `label`, `visible`, `enabled`, `pressed` or a prop name) passes every test.

A state that core has not seen yet counts as not holding. A run whose conditions fail is logged as `skipped` with reason `conditions not met`. Conditions are also checked when the automation is run by hand, from a button or the API (unless forced), but not when another automation's step runs it.

Steps

KIND	IN THE DESIGNER	FIELDS	WHAT IT DOES
`intent`	Intent	`intent`, optional `wait`, `timeoutMs`, `onTimeout`	Sends an intent (scene recall, power, level and so on) to a device, a room, devices with a tag, or everything.

KIND	IN THE DESIGNER	FIELDS	WHAT IT DOES
`action`	Device action	`deviceId`, `capability`, `action`, optional `params`, `wait`, `timeoutMs`, `onTimeout`	Sends one device action.
`if`	If	`conditions`, optional `match` (`all` or `any`), `steps`, `otherwise`	Runs `steps` when the conditions hold, `otherwise` when they do not.
`repeat`	Repeat	`count`, `steps`	Runs the steps `count` times. `{{index}}` counts from 0.
`while`	While	`conditions`, optional `match`, `steps`, `maxIterations` (default 1000)	Runs the steps again while the conditions hold, checked before each pass. Fails with `loop limit reached` past the limit.
`delay`	Wait	`ms`	Waits, up to 86400000 ms (one day).
`waitUntil`	Wait for a condition	`conditions`, optional `match`, `timeoutMs` (default 30000), `onTimeout`	Waits until the conditions hold. At the timeout: `continue` (default) or `fail`.
`stop`	Stop	optional `result` (`ok` or `fail`), `reason`	Ends the run here, completed or failed.
`set`	Set a variable, Expression	`variable`, `value` or `add`	Gives a variable a value, or adds to it (a missing one counts as 0). The designer's **Expression** is a `set` whose value is an expression.
`control`	Set a control	`controlId`, optional `value`, `add`, `label`, `visible`, `enabled`, `props`	Sets a dashboard control on every screen showing it. `null` clears an override.
`automation`	Run an automation	`automationId`, optional `params`, `wait` (default `true`)	Runs another automation, inside this run (`wait: true`, its failure fails the step) or as its own run (`wait: false`). Calls go at most 8 deep and an automation cannot call itself.

KIND	IN THE DESIGNER	FIELDS	WHAT IT DOES
`call`	Function	`function`, optional `args`, `into`, `timeoutMs`, `onError`	Calls a library function and keeps its result in `into`. See Function libraries.
`timeline`	Timeline	`timelineId`, `op`, optional `positionMs`, `cue`	Transport: `play`, `pause`, `stop`, `seek` or `go`. `cue` jumps to a cue by id or `{{name}}`. See Timelines.
`comment`	Comment	`text`	A note in the list. Does nothing.
`log`	Log	`message`, optional `level` (`info`, `warning`, `error`), `deviceId`	Writes a line in the run's log and the core log, and in the device's log when `deviceId` is given.

Every step takes `disabled: true` to be skipped (the designer's **On** checkbox).

Intents and device actions are sent with source `automation` and the automation's id, so the intent engine's priorities and holds apply. A `delay` in one automation never holds up another.

Waiting until devices are done

By default an intent or device action step waits until every device accepted it (`wait: "sent"`), then the next step runs. Tick **Wait until the device reports it is done** (`wait: "done"`) to wait until each device reports that it finished: a projector is on, not warming up; a screen is down. The step waits at most `timeoutMs` (default 120000, **At most** in the designer), then goes on (`onTimeout: "continue"`, the default) or fails the run (`"fail"`).

```json
{
  "kind": "intent",
  "intent": { "name": "power.on", "target": { "kind": "room", "id": "gallery" } },
  "wait": "done",
  "timeoutMs": 120000
}
```

Failures

When a step is rejected or fails, the run ends as `failed`, and later steps are not run. An action sent to an offline device fails at once. With `continueOnError: true` (**Continue after a failed step**), the remaining steps run and the run ends as `partial`.

The run records the failed step by its position (`step`, for example `2` or `3.1` inside an `if`) and a `reason`, for example `power.on partial: right-projector device offline`.

Variables and expressions

Variables

Three kinds of values are read as `{{name}}` in step values, templates and expressions:

- **Params** of the automation, filled by whatever runs it. A dashboard control passes its value as the param named `value`. A listener rule passes its captures.
- **Run variables**: any name a `set` or `call` step writes that is not a project variable. They live for one run.
- **Project variables**, declared under **Variables** in the project tree (`project.variables`): values core keeps for the whole project, for example `houseMode = "show"` or `visitors = 12`. Everyone sees them: dashboards show and change them, the API reads and writes them, and `variable` triggers start automations when they change. They are kept across restarts unless **Keep** is unticked (`persist: false`).

Triggers add their own values: `{{value}}` and `{{previous}}` for variable and control triggers, captures for listener triggers, `{{index}}` inside a `repeat`.

A value that is only `"{{name}}"` keeps the variable's type (a number stays a number). `"Room {{room}} is ready"` fills it in as text.

Expressions

Wherever a step takes a value, it can take an expression instead, computed when the step runs. In the designer, click **ƒx** next to a value.

```JSON
{ "kind": "set", "variable": "level", "value": { "$expr": "clamp({{level}} * 1.2, 0, 100)" } }
```

An expression has:

- `{{name}}` for params and variables, numbers, `"strings"`, `true`, `false`, `null`.
- `+ - * / %`, `== != < <= > >=`, `&& || !`, `a ? b : c` and parentheses.
- `state(device, capability, key)` to read device state.
- Built-in functions: `abs`, `min`, `max`, `round`, `floor`, `ceil`, `clamp`, `len`, `upper`, `lower`, `str`, `num`, `now`.
- Dashboard controls: `control.value(id)`, `control.label(id)`, `control.visible(id)`, `control.enabled(id)`, `control.pressed(id)`, `control.min(id)`, `control.max(id)`, `control.prop(id, name)`. `control.click(id)`, `control.press(id)`, `control.release(id)` and `control.hold(id)` are true when this run was started by that event of that control, and `control.event(id)` gives the event name.
- Device playheads: `device.timecode(id, playhead?)`, `device.position(id, playhead?)`, `device.duration(id, playhead?)` and `device.running(id, playhead?)`, for devices with the `tangle.playhead` capability (media servers, players). Without a playhead name they read the main one.
- Library functions such as `math.scale(...)` or `time.format(now())`.

Function libraries

Function libraries add named functions to expressions and to the **Function** step (`call`), written `namespace.name(args)`. Core comes with five libraries:

LIBRARY	FUNCTIONS
`math`	`pow`, `sqrt`, `exp`, `log`, `log10`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `atan2`, `hypot`, `degrees`, `radians`, `sign`, `pi`, `mod`, `scale`, `lerp`, `clamp01`, `roundTo`, `dbToGain`, `gainToDb`
`text`	`trim`, `pad`, `replace`, `split`, `join`, `contains`, `startsWith`, `endsWith`, `indexOf`, `slice`, `repeat`, `format`, `hex`, `fromHex`
`time`	`format`, `parse`, `add`, `diff`, `ms`, `duration`, `hour`, `minute`, `weekday`, `date`
`list`	`at`, `first`, `last`, `sum`, `avg`, `min`, `max`, `sort`, `reverse`, `unique`, `includes`, `slice`, `range`
`color`	`hex`, `rgb`, `hsv`, `fromHsv`, `kelvin`, `mix`, `toRgbw`, `brightness`

For example `math.scale(128, 0, 255, 0, 100)` is 50.2, `time.add(now(), "1h30m")` is in an hour and a half, and `color.kelvin(3200)` is the colour of warm white light.

Modules can add libraries too. The **Image processing** library (`tangle.image.library`, namespace `image`: `info`, `resize`, `thumbnail`, `crop`, `rotate`, `flip`, `grayscale`, `adjust`, `compose`, `convert`, `averageColor`, `dominantColors`, `pixel`) needs the IMAGE entitlement of Pro and Ultimate licenses; the free designer and demo mode can call every library. Install it from **Settings > Modules**.

Only pure functions (same arguments, same result, no side effects) can be used inside expressions and conditions. Any function can be used in a **Function** step, which waits for it (default 5 s, at most 120 s) and fails the run on an error, or goes on with `{{error}}` set when **On error** is continue.

`GET /api/v1/functions` lists every function with its signature, and `POST /api/v1/functions/{name}/call` tries one. Writing a library module is covered in Module SDK.

Run modes

`mode` decides what happens when a trigger fires during a run:

MODE	BEHAVIOUR
`single`	Ignore the new trigger. The ignored run is logged as `skipped` (`already running`).

MODE	BEHAVIOUR
`restart`	Cancel the run in progress (logged as `cancelled`, `restarted`) and start again.
`queued`	Run again after the current run. Up to 10 runs can wait; more are skipped (`queue full`).
`parallel`	Run alongside the current run, up to 10 at once; more are skipped (`too many runs`).

Loop guard

An automation can trigger itself, for example by changing the state that its trigger watches. Core starts at most 10 runs of one automation per second. Further runs are skipped with reason `rate_limited` and a warning is logged. The show and other automations keep running.

Timing

From the triggering event to the first step takes under 20 ms on the primary. Time triggers fire within a second of the minute.

Running an automation by hand

Running by hand needs a token with the `control` scope. Conditions are checked unless you force the run.

REST:

```sh
curl -X POST http://127.0.0.1:7340/api/v1/automations/morning/run \
  -H "Authorization: Bearer $TOKEN" -H "content-type: application/json" \
  -d '{"force": true, "params": {}}'
```

`force: true` skips the conditions; `params` fills the automation's params. The answer is the run, with `state` `started` or `skipped`. `POST /api/v1/automations/{id}/stop` cancels its runs in progress.

WebSocket (for example from a dashboard button):

```json
{ "v": 1, "id": "7", "type": "automation.run", "automationId": "close-room" }
```

The acknowledgement carries the run (`runId`, `state`). The run's trigger is recorded as `manual`.

MCP: `run_automation`.

A manual run is refused with `409 not_primary` on a backup node.

Run history

Core keeps the last 50 runs of each automation in memory (they are not kept across restarts).

- Designer: the **Runs** tab of the automation.
- REST: `GET /api/v1/automations/{id}/runs`, newest first.
- MCP: `get_automation_runs`.
- Events: subscribe to `automation.run:<id>` (or `automation.run:*`). Core publishes an event when a run starts and when it ends.

Each run looks like this:

```JSON
{
  "runId": "4f0c6d1e-...",
  "automationId": "morning",
  "state": "completed",
  "trigger": "time 08:00",
  "params": {},
  "log": [],
  "startedAt": 1760356800012,
  "endedAt": 1760356800190
}
```

Run states: `started`, `completed`, `partial`, `failed`, `skipped`, `cancelled`. `reason` explains a failed, partial, skipped or cancelled run, and `step` names the step in progress or the one that failed. `log` holds the lines of log steps (at most 100). `trigger` describes what started it, for example `manual`, `startup`, `deviceEvent keypad-1 button.pressed` or `timeline show cue c4`.

Automations and redundancy

Only the primary node (or a standalone node) runs automations, also in shared mode. A backup stays idle and starts running them when it is promoted, which also fires its `startup` triggers. Runs in progress on a failed primary are not resumed. When a node steps down to backup, its runs in progress are cancelled. See Redundancy.

More examples

Recall a scene when a keypad button is pressed:

```json
{
  "id": "present",
  "name": "Presentation button",
  "triggers": [{ "kind": "deviceEvent", "deviceId": "keypad-1", "name": "button.pressed" }],
  "steps": [
    { "kind": "intent", "intent": { "name": "scene.recall", "target": { "kind": "room", "id":
"room-a" }, "params": { "scene": "presentation" } } }
  ]
}
```

Start the attract loop at night only, when the show timeline has stopped:

```json
{
  "id": "attract",
  "name": "Attract loop after the show",
  "triggers": [{ "kind": "timeline", "timelineId": "show", "on": "stopped" }],
  "conditions": [{ "kind": "time", "after": "19:00", "before": "23:00" }],
  "steps": [
    { "kind": "delay", "ms": 5000 },
    { "kind": "timeline", "timelineId": "attract", "op": "play" }
  ]
}
```

Count visitors with a door contact on an Ethernet I/O module, and switch the house to "full" at 120:

```json
{
  "id": "count-visitors",
  "name": "Count visitors",
  "triggers": [{ "kind": "deviceEvent", "deviceId": "door-sensor", "name": "input.changed" }],
  "mode": "queued",
  "steps": [
    { "kind": "set", "variable": "visitors", "add": 1 },
    {
      "kind": "if",
      "conditions": [{ "kind": "variable", "variable": "visitors", "above": 119 }],
      "steps": [{ "kind": "set", "variable": "houseMode", "value": "full" }]
    }
  ]
}
```

Set a room level from a dashboard slider (0 to 100) with a param. `level.set` takes a level from 0 to 1:

```json
{
  "id": "set-level",
  "name": "Set the room level",
  "params": [{ "name": "value", "label": "Level (%)", "default": 50 }],
  "steps": [
    {
      "kind": "intent",
      "intent": { "name": "level.set", "target": { "kind": "room", "id": "room-a" }, "params":
{ "level": { "$expr": "clamp({{value}} / 100, 0, 1)" } } }
    }
  ]
}
```

Shut a room down by hand from a dashboard, continuing past devices that do not answer:

```json
{
  "id": "close-room",
  "name": "Close room",
  "triggers": [{ "kind": "manual" }],
  "steps": [
    { "kind": "timeline", "timelineId": "show", "op": "stop" },
    { "kind": "intent", "intent": { "name": "power.off", "target": { "kind": "room", "id":
"room-a" } } }
  ],
  "continueOnError": true,
  "mode": "single"
}
```

GUIDES

Dashboards

Build control panels for operators, rooms and wall panels with tangle-interact: widgets, device layouts, screen sizes, access, kiosks and skins.

Dashboards are the control surfaces people touch: an operator console in the booth, a room panel by the door, a status wall for the building team. You build them in the designer and screens open them in a browser at `/d/<id>`. Core serves both; nothing else is installed on the screens.

A dashboard has one or more pages of widgets on a grid, with a layout per screen size, a skin, a theme and an access mode. Dashboards are stored in the project.

Open a dashboard

Every published dashboard has an address on the core:

```http
http://<core address>:7340/d/<dashboard id>
```

For example `http://10.0.0.2:7340/d/room-hall`. In the builder, **Open** next to a dashboard opens it in a new tab.

> **Note:** For tablets and panels to reach the dashboard, core must accept other machines (`--listen 0.0.0.0:7340 --allow-remote --insecure-http`) and must have a license or run with `--demo`. The free designer serves only its own computer. See Install.

Address options

Add these to the address after `?`, separated by `&`:

```http
/d/room-template?room=room-3&kiosk=1&cores=10.0.0.2:7340,10.0.0.3:7340&panel=<token>
```

PARAMETER	MEANING
`room`	The room for a dashboard made for many rooms (see room templates)
`kiosk=1`	Wall panel mode: stays on this dashboard, keeps the screen awake, and is remembered on the device, so the panel comes back to it after a reboot even if its browser only opens `/d/`
`reset=1`	Forget the remembered kiosk settings

PARAMETER	MEANING
`cores`	Core nodes to try first. Optional with redundancy: the dashboard learns the other nodes from the one it reaches and follows the primary. See One address for operators.
`panel`	An enrolled panel token. Stored on the device and removed from the address bar.
`awake=0`	Kiosk: let the screen sleep
`dim=<minutes>`	Kiosk: dim the screen after this many minutes without a touch. A touch wakes it.

What screens see

- Live state: values, status and timeline positions update as they change. A dashboard subscribes only to what its widgets use.
- When the connection to core drops, a banner says "Disconnected from Tangle. Reconnecting..." and controls are disabled until it is back.
- When a control is pressed, it shows a spinner while core works, and "Failed:" with the reason if the action fails.
- When you publish a new version, screens showing the dashboard reload it on their own.
- A dashboard with several pages shows page tabs at the top.

The builder

Click **Dashboards** in the designer's project tree. The dashboard list shows each dashboard with its published version and an **Open** link, and three ways to start:

- **New dashboard**: type a name and click **Create**. The builder opens on an empty page named Main. The dashboard is stored when you first click **Save draft**.
- **Generate**: make a dashboard from the project (see generated dashboards).
- Click a dashboard's name to edit it.

The editor

AREA	WHAT IT HOLDS
Top bar	Dashboard name, screen size (**Phone**, **Tablet**, **Panel**, **Desktop**), **Undo** and **Redo** (Cmd+Z and Shift+Cmd+Z, or Ctrl on Windows and Linux), **Preview**, **Regenerate** (generated dashboards), **Save draft**, **Publish**
Palette (left)	**Basic controls**, **Input controls**, **Widgets**, **Studio presets**, and **Devices** grouped by room

AREA	WHAT IT HOLDS
Page (centre)	Page tabs, **+ Page**, and the grid for the chosen screen size
Side panel (right)	**Inspector** of the selected widget, dashboard **Settings**, published **Versions**
Footer	Problems that block saving, broken bindings, and the preview log

To build a page:

1. Drag a widget from the palette onto the page, or click it to add it at the bottom.
2. Move and resize widgets on the grid.
3. Select a widget to set its label, bindings and action in the **Inspector**.
4. Press Delete or Backspace to remove the selected widget.

Broken bindings in the footer lists widgets that point at a device, state key or timeline that no longer exists. Click one to select it.

Preview

Tick **Preview** to see the page as screens will, at the chosen screen size. Presses are only logged ("Would send power.on to device hall-projector"). Tick **Live control** to send them to core for real. For a room template, pick the room to preview with.

Widgets

WIDGET	SHOWS	DOES
Basic controls		
Button	Its label, lit while **Lit when** holds	Sends its action; can send another when held
Toggle	On, Off or – from its value	Sends `action` to turn on, `offAction` to turn off
LED	An indicator lamp, lit from its value	Display only
Label	Text or a heading (**Size** Normal, Small, Large or Title)	Display only
Picture	A picture or a logo, uploaded or from an address	Display only

WIDGET	SHOWS	DOES
Shape	A box, a circle or a line	Display only
Value	A value with an optional unit	Display only
Slider	A level from its value, between `min` and `max` (default 0 to 1), as a percentage	Sends its action with the new value while dragging (at most every 200 ms) and on release
Input controls		
Select list	A drop-down list	Sends or sets the chosen option
Text field	A text or number field	Sends or sets what was typed
Keypad	A numeric keypad	Sends or sets the number entered
Stepper	A value with – and + keys	Steps the value
Colour	A colour picker	Sends or sets the colour
Direction pad	Up, down, left, right and centre keys	Sends an action per key (PTZ, menus)
XY pad	A two-axis pad	Sends both values
Widgets		
Meter	A level bar from its value, `min`, `max`	Display only
Source selector	One button per entry in `options`, the current one lit from its value	Sends its action with the chosen value
Scene list	One button per scene (from `options`, or every scene in the project)	Recalls the scene, by default in the dashboard's room
Status	A device's status (online, offline…) with its detail	Display only
Transport	Play, Pause, Stop (GO, Pause, Stop for a cue list), status and position of a timeline	Drives the timeline
Cue + GO	Last and next cue of a timeline, and a large GO button	Sends GO
Clock	Time of day, a timeline position, or a device's timecode	Display only

WIDGET	SHOWS	DOES
Group	A titled container for other widgets, with its own grid	Holds device control groups and status groups
Map	A map from the project, with live device status and tracked positions	Tap a device to see its status
Web page	Another site or page in a frame (`url`, http or https)	Optional reload every `refresh` seconds
Camera	A camera's video: MJPEG, snapshots, HLS or WebRTC, or a device's preview stream	With a camera device, a pan, tilt and zoom pad
Navigation bar	A row (or column) of buttons to pages, dashboards, maps or addresses	Opens the destination, the current one lit
Link	Text or a button to a page, dashboard, map or address	Opens it, optionally in a new tab
Executor	A lighting playback fader and its keys	Drives a lighting executor
Device log	The log of a device, several devices or a room's devices, filtered by kind and level	Display only
List	Rows to choose, reorder or remove, such as a media player's playlist	Sends the chosen row, the new order or the removal

Studio presets are ready-made controls in the Studio style: Knob, Fader, Lit key, Lit switch, Key row, LED meter, Readout, Status lamp and Faceplate.

See Embedded content for the settings of Map, Web page, Camera, Navigation bar and Link.

Control ids

Every widget has an **ID**, unique in the whole project. Automations, bindings and the API name a control by its id, and renaming it in the Inspector updates every reference. A control without a binding keeps its value in core, so every screen showing it agrees. Automations set controls with the **Set a control** step and start on them with the trigger **A dashboard control is clicked or changes** (see Automations). REST: `/api/v1/controls`; MCP: `list_controls`, `set_control`, `press_control`, `rename_control`.

Bindings

A control's properties follow bindings: **Value** (or **Lit when** for a button), **Label from**, **Shown when** and **Enabled when**. Each binding has a source:

SOURCE	READS
Not bound	The value set in the Inspector (always shown and enabled)
Device state	A state value of a device, for example Media player `tangle.transport.status`, with a test such as "is playing"
Project variable	A project variable
Another control	Another control's value or property
Expression	An expression, for example `{{visitors}} > 100` or `control.value("level") * 100`
Fixed value	A constant, for example a source list or a unit

Timeline widgets bind to a timeline's `status`, `positionMs`, `lastCue` or `nextCue`, and the system values `connection`, `role`, `nodeId`, `license` and `time` are available too. **ƒx** next to other properties of a control (colours, ranges, text) makes them follow an expression or another control. An expression is live on every screen only once the dashboard is published.

The builder only offers bindings that fit the property (a slider only gets numbers).

Actions

A control's **Action** is one of:

- **Send an intent**: a name such as `power.on`, `input.select` or `scene.recall`, a target (a device, a room, a tag, or everything) and parameters. A parameter set to `"{{value}}"` is replaced by the control's value: the slider level, the chosen input, the scene picked. See intents.
- **Control a timeline**: Play, Pause, Stop or GO on a timeline.
- **Run an automation**: runs an automation, with **Parameters** (JSON) where `"{{value}}"` is the control's value. The automation's conditions are checked.
- **Set a variable**: sets a project variable to a value, or adds to it.

A button can also do something else when held: **When held** takes any of these actions, sent instead of the normal action once the button is held for the hold time (default 0.8 s). A thin line fills along the button's bottom edge while it is held.

Dashboard intents have priority 30, so a running timeline or automation (50) wins over a panel press on the same device. Older dashboards that use `"$value"` still work.

Confirmation

For critical actions, set **Confirmation** on the widget:

SETTING	BEHAVIOUR
None (`none`)	Acts on the first press
Press twice (`confirm`)	The first press arms it ("Press again to confirm"); a second press within 3 seconds sends it
Press and hold (`hold`)	Press and hold for the hold time (default 0.8 seconds); releasing early cancels

Device controls and layouts

Modules can describe ready-made controls for their devices and group them into named layouts. The PJLink module, for example, offers:

LAYOUT	CONTROLS
Compact	Power and status
Operator	Power, input, picture mute, status, current input
Full	Everything, for technicians: adds power state (warming, cooling) and lamp hours

To place a device:

1. Drag the device from **Devices** in the palette onto the page (or click it). Devices are listed by room.
2. Choose a **Layout**.
3. Tick or untick **Controls to show**. Ticking a control that is not in the layout switches to an automatic arrangement of the ticked controls.
4. Pick a **Control style** if the group should not use the dashboard's.
5. Click **Add to page**.

The controls are placed as a group bound to that device, already wired: the power toggle reads `tangle.power on` and sends `power.on` and `power.off`, with the confirmation the module asks for.

A module that declares no controls gets them from its standard capabilities, in one "All controls" layout: power toggle, level slider, mute toggle, current input, current preset and transport status, plus status. The dialog says "This module declares no controls: these come from its capabilities." Module authors define controls in the manifest; see Module SDK.

Pages and screen sizes

A dashboard has one or more pages; screens show them as tabs. Add one with **+ Page**.

Each page has a layout per screen class. Pick the class in the top bar and arrange the grid for it:

CLASS	GRID COLUMNS	USED WHEN
Phone	4	The window is under 600 px wide
Tablet	12	600 px to under 1100 px
Desktop	12	1100 px and wider
Panel	12	Kiosk screens (`kiosk=1`) 600 px and wider

You do not have to lay out every class. A class without its own layout uses the nearest class that has one; on a phone, widgets are stacked in one column in reading order (top to bottom, then left to right). Widgets with no position in a layout are added at the bottom. New widgets are added to every layout the page already has.

Inside a group, children share one arrangement across classes and stack on phones when the group is wider than the phone grid.

One dashboard for many rooms

Twenty identical meeting rooms need one dashboard, not twenty. In **Settings**, tick **One dashboard for many rooms (open with ?room=...)**. Device references then select a device by room, capability or tag (for example "the first device with `tangle.power` in this room") instead of naming a fixed device, and room targets use the room the dashboard is opened with. Open it with the room in the address:

```http
/d/meeting-room-template?room=meeting-3
```

Without this option, you can set the dashboard's **Room**, which scene lists and new widgets use by default.

Access

In **Settings**, **Who can open it**:

MODE	WHO	NOTES
Anyone on the network	Every browser that reaches core	The default
People with the PIN	Anyone who types the PIN	4 to 12 digits. After 5 wrong PINs in a minute, the dashboard asks to wait a minute. Core stores only a hash of the PIN.

MODE	WHO	NOTES
Enrolled panels only	Devices opened with a panel token	Other browsers see "This panel is not enrolled".

A browser that opens a dashboard gets a session that lasts 12 hours, then asks again (for a PIN dashboard, the PIN). With redundancy, sessions survive a failover.

Enrol a wall panel

1. Create a token for the panel on the core:

```sh
tangle-core token create --name lobby-panel --scopes read,control
```

2. On the panel, open the dashboard once with the token and kiosk mode:

```http
http://10.0.0.2:7340/d/lobby?kiosk=1&panel=<token>
```

The panel stores the token, removes it from the address bar and keeps showing the dashboard after reboots. To unenrol it, revoke the token: `tangle-core token revoke lobby-panel`.

> **Note:** Panel tokens belong to one core. With redundancy, create the panel's token on every node.

> **Warning:** A dashboard session can read the whole project, device settings included (passwords and keys never are). Do not rely on dashboard access modes to hide project details from people on the network.

Generated dashboards

The builder can make a dashboard from what the project already knows. In the dashboard list, under **Generate**:

BUTTON	MAKES	SKIN
Generate room dashboard	A title, a scene list with the room's scenes (scenes whose actions all use devices of the room), then each device's controls (power, level, mute, input, status)	Lobby, theme follows the device
Generate show console	For a timeline. Cue list: last and next cue with a large GO, transport, position. Timecode: transport, timecode display, cues.	Console, dark
Generate facility overview	The status of every device, grouped by room, plus a "No room" group	Console

For a room, tick **For every room like this one** to make a room template instead.

A generated dashboard is a normal dashboard: edit it freely. When the project changes (new devices, a removed scene), click **Regenerate**. Widgets for new devices and scenes are added at the bottom, widgets whose source is gone are marked as removed, and everything you changed or added stays as it is.

Skins and themes

Each dashboard has a skin and a theme, set in **Settings**.

SKIN	LOOK	DEFAULT FOR
Console	Dark lighting-desk look, amber accent, condensed uppercase labels, monospace timecode, dense square controls (40 px)	Operators, show consoles, overviews, the builder and the designer
Lobby	Stone and sage tones, pill buttons, large rounding, taller controls (44 px)	Room and building panels used by non-technical people

Theme: **Follow the device** (light or dark from the screen's setting), **Light** or **Dark**. Both skins have both modes, and every text and control colour meets WCAG 2.1 AA contrast.

Fonts are bundled with the UI, so dashboards render the same at venues with no internet. A dashboard with no skin, or an unknown one, renders as Console.

Control styles

The **control style** draws the controls themselves, inside the skin:

STYLE	LOOK
Classic	The skin's flat controls
Studio	Knobs, lit keys, faders, LED meters, readouts
Skeuomorphism	Brushed metal, rubber keys, pointer knobs, LCD readouts
Relief	Soft raised and pressed shapes in flat colours

The project's default is set in the designer's **Settings > Dashboard controls**, with each style's **highlight colours** (Accent: value arcs, fader fills and focus; Display: readout digits). A dashboard, a group or a single control can pick another style in the builder.

Background

Background in **Settings** takes a colour (with its opacity), an uploaded image or a web address. **Widget cards** below 100 % lets the background show through the controls' cards.

Units

Devices report in their own units (°C, W, h…). Dashboards show values in the project's system (Metric or Imperial, set in the designer's **Settings > Units**). A dashboard can pick another in **Settings**, and a viewer can add `?units=imperial` to the address.

Drafts and publishing

Screens always show the published version. While you edit:

- **Save draft** keeps your work on the core without changing what screens show ("Draft saved. Screens still show the published version.").
- **Publish** makes the current state the new published version. Screens showing it reload at once.

Before saving or publishing, the builder checks the dashboard and lists problems in the footer under "Cannot save:". Core checks it again.

Versions in the side panel lists every published version with its time and who published it. **Restore** publishes an old version again as a new version; the others are kept. Drafts and versions are stored in the core's data folder, under `dashboards/<project id>/`.

Edit without the designer

Dashboards are part of the project, so the REST API and MCP can create and publish them too. The format is the `Dashboard` schema of the contract: `pages`, each with `widgets` and `layouts` per screen class, plus `skin`, `theme`, `access`, `room` or `roomParam`. See Project file.

GUIDES

Redundancy

Run the same show on two or more tangle-core machines in lockstep, with automatic failover in under a second.

A redundant cluster runs the same project on two or more machines. Every node computes the show, so timelines run on all of them at the same moment, but only the primary drives devices. If the primary stops, the next node takes over within a second, already at the same point of the show.

Redundancy needs Tangle Ultimate, or Tangle Pro with the redundancy add-on. See Edition requirement.

Concepts

Primary and backups. One node is the primary: it sends commands to devices, runs automations and Sentinel, and accepts changes. The other nodes are backups. A node without redundancy is `standalone`.

Lockstep. Each backup keeps a connection to the primary and receives the project, device state, and every transport operation with the cluster time at which it happened. Each backup then plays every timeline itself, in shadow: cues go to its cue log, not to devices. When the primary fails, a backup is already where the show is.

Cluster clock. All nodes share one time scale, measured against the primary's clock. Every node computes the same execution time for every cue. A new primary keeps its clock offset, so cluster time never jumps.

Epochs. The epoch is a counter that goes up on each failover. If two nodes ever both claim to be primary, the higher epoch wins (on a tie, the smaller node id), and the other steps down.

Quorum. With 3 or more nodes, a node can only be or become primary while it reaches a majority of the cluster (itself included). A network split therefore never gives two primaries. With 2 nodes no majority can survive a failure, so the quorum is 1.

Cue fingerprints. Each node keeps a cue log: the count of cues fired (or shadow-fired) and a digest of that sequence. Equal digests at equal counts prove two nodes computed the same show. Heartbeats carry this position, and backups compare theirs with the primary's.

Diverged. A backup whose cue log differs from the primary's reports `diverged`, resyncs from the primary, and cannot take over until it is in sync again.

Failover timing. Nodes send heartbeats every 100 ms. When the primary is silent for 500 ms, the first candidate takes over. Each next candidate in election order waits 200 ms longer, so only one takes over.

Who is primary. The node started with `--role primary` goes first, then the others by node id. Only healthy, synced backups are candidates. Every node starts as a backup and listens before becoming primary, so a node that comes back after a failover never fights the one that took over.

Requirements

- The same project on every node. By default a backup adopts the primary's project and saves it to its own project file, so you edit the project on the primary only.
- A cluster key shared by every node. When you add a core from the designer, core creates it for you. For a cluster set up by hand, choose one of at least 16 characters.
- A unique node id per node.
- UDP and TCP port 7341 open between all nodes (heartbeats over UDP, state mirroring over TCP). `--cluster-listen` changes the address and port.
- For automatic discovery and pairing: UDP and TCP port 7342 open, and the nodes on the same local network (discovery uses multicast, which does not cross routers).
- The same device modules installed on every node.
- A license activated on every node. See Edition requirement.

Heartbeats and mirror messages are encrypted and authenticated with the cluster key (XChaCha20-Poly1305). Nothing else on the network can pose as a node or read the project, device state or device passwords that cross between nodes. A message that fails authentication is dropped.

Device passwords and keys, and notification channel secrets, follow the primary: it copies them to every backup over the encrypted channel when the backup joins and whenever they change, and each backup stores them in its own credential store. Cores kept separate get nothing. A backup that takes over logs in to password-protected devices with the same passwords.

Add a core from the designer

Every core announces itself on the local network. The main core (the primary, or a standalone core) lists the cores it hears and asks you what to do with each new one. This is the easiest way to build a cluster: no node ids, peers or keys to type.

1. Install tangle-core, a license and the same device modules on the new machine, and start it. Its project does not matter: it will take the main core's.
2. Find the new core's 6-digit pairing code. It is in the new core's log (`pairing code of this core code=482913`) and on the **Nodes** page of its own designer. A new code is made at each start, and after 5 wrong tries.
3. In the designer of the main core, a banner tells you a new core was found. Open **Nodes**: the new core is listed with its name, host and address.
4. Choose:
   - **Keep separate**: the new core runs its own, independent show. The main core remembers your choice and does not ask again. Nothing changes on the new core.
   - **Add as redundant**: type the pairing code. The two cores pair, and the new core joins this core's cluster as a backup.

Accepting a core needs a token with the `admin` scope, and a license that allows the new cluster size. If it does not, core refuses with the reason.

What happens when you pair:

- The two cores prove they know the code without sending it (a password-authenticated key exchange, SPAKE2). A wrong code fails with `wrong_pairing_code` and changes nothing on either core.
- The main core creates a random cluster key and hands it to the new core inside that exchange. From then on all traffic between the cores is encrypted.
- The main core becomes the primary without stopping its devices or timelines. The new core stops its own devices, takes the primary's project (its previous project file is kept as `<name>.before-cluster.<ext>`), syncs, and receives the device passwords.
- Every member learns about the new node and saves the membership. Quorum follows the new size: a third node makes the quorum 2.
- Each core saves its membership and key in `cluster.json` in its data folder, readable by the Tangle account only. At the next start it rejoins its cluster without any option.

`tangle-core cluster show` prints this machine's saved cluster. To take a machine out of its cluster, stop core on it and run `tangle-core cluster leave`. This forgets the saved membership on that machine only; the other members keep it in their peers, which is harmless while it is off.

> **Note:** A designer sign-in made on the primary works on a newly paired backup only after both cores restart.

Cores in another cluster are listed as `clustered` and cannot be paired. A backup does not list discovered cores; open the designer on the primary. To stop a core from announcing itself, start it with `--no-discovery`. Across routed networks, where multicast does not reach, set up the cluster by hand as below.

Set up a 3-node cluster

You can also build a cluster by hand, with options on each node. Three nodes are the recommended setup: a network split can never produce two primaries.

1. Install tangle-core, the same modules and the same project file on the three machines. In this example they are 10.0.0.1, 10.0.0.2 and 10.0.0.3.
2. Choose a shared key and set it on every machine:

   ```sh
   export TANGLE_CLUSTER_KEY=<same secret on every node, 16+ characters>
   ```

3. Start each node with its own `--node-id` and the cluster addresses of the other nodes in `--peer`:

```sh
# machine A (10.0.0.1): goes first
tangle-core run -p show.yaml --node-id core-a --peer 10.0.0.2:7341,10.0.0.3:7341
# machine B (10.0.0.2)
tangle-core run -p show.yaml --node-id core-b --peer 10.0.0.1:7341,10.0.0.3:7341 --role
backup
# machine C (10.0.0.3)
tangle-core run -p show.yaml --node-id core-c --peer 10.0.0.1:7341,10.0.0.2:7341 --role
backup
```

4. Check the cluster (see Check the cluster): one primary, two backups with `synced: true`.

To accept designer and dashboard connections from other machines, add the remote access options to each node: `--listen 0.0.0.0:7340 --allow-remote --insecure-http`. See Operations.

Set up a 2-node cluster

```sh
export TANGLE_CLUSTER_KEY=<same secret on both nodes>
# machine A (10.0.0.1)
tangle-core run -p show.yaml --node-id core-a --peer 10.0.0.2:7341
# machine B (10.0.0.2)
tangle-core run -p show.yaml --node-id core-b --peer 10.0.0.1:7341 --role backup
```

> **Warning:** With 2 nodes, if the link between them drops, both become primary and both drive devices until the link is back. Then the higher epoch wins and the other steps down. Use 3 nodes, or connect the two machines with a dedicated link.

Cluster options

OPTION	MEANING
`--node-id <id>`	Name of this node. Required with `--peer`, and unique per node.
`--peer <host:port>`	Another node's cluster address. Repeat it, or separate addresses with commas, for each other node. Turns redundancy on.
`--role primary` or `--role backup`	`primary` (the default) makes this node go first when choosing a primary. Give `--role backup` to every other node.
`--cluster-listen <addr:port>`	Address for heartbeats (UDP) and mirroring (TCP). Default `0.0.0.0:7341`.

OPTION	MEANING
`--cluster-key <key>`	The shared key for a cluster set up by hand. Prefer the `TANGLE_CLUSTER_KEY` environment variable. Traffic is encrypted with a key derived from it.
`--cluster-key-file <path>`	File holding the shared key.
`--no-discovery`	Do not announce this core on the network and do not listen for discovered cores.
`--no-project-sync`	Do not copy the primary's project to the backup. You must update each node's project by hand; a backup with a different project reports `degraded` and will not take over.

Running as a service

Set `TANGLE_CLUSTER_KEY` when you install the service. The key goes to a file only the service can read (`cluster.key` in the data folder), not into the service definition:

```sh
sudo TANGLE_CLUSTER_KEY=<shared key> tangle-core service install -- \
  --project /shows/main.yaml --node-id core-a --peer 10.0.0.2:7341,10.0.0.3:7341
```

On Windows, set the key in an administrator PowerShell first (`$env:TANGLE_CLUSTER_KEY = "<shared key>"`), then run `tangle-core service install -- ...`.

See Operations.

Sharing the work

By default a cluster runs in **standby** mode: the primary drives every networked device and the other nodes wait to take over. A large site can instead spread the devices across the nodes, in **shared** mode, to lighten the load on one machine. The primary stays the one "brain": it runs timelines, automations, intents, variables and the lighting programmer; the other nodes run device modules and keep shadowing the show, so any of them can still take over.

Set it on the **Nodes** page, under **Sharing the work** (stored in the project as `cluster`, so every node agrees):

SETTING	MEANING
Mode	**Standby: the primary runs every device** (default) or **Shared: spread devices across the nodes**
Move devices back when their node returns	Off by default: a device that moved stays where it is until **Rebalance**

SETTING	MEANING
Folders on a node	Devices in a folder of the device list run on the chosen node, unless the device picks one itself (**Runs on** on the device page, `runsOn`)
Rebalance	Spreads the devices evenly again and moves devices back to their usual node. Only the devices that change node restart. **Preview rebalance** shows the moves first.
Health limits	When a node shows "low disk", "low memory" or "overloaded" (defaults: 10 % disk free, 10 % memory free, CPU above 85 % for 30 s). Sentinel's core node alert rules use the same limits.

How devices are placed:

- **Hardware attached** to a node (`node`) always runs there and never moves.
- **Runs on** (`runsOn`) or a folder rule gives a networked device a preferred node. It moves to another node while that one is down.
- Everything else is placed automatically, balanced by device weight across the healthy nodes. A device does not move again unless its node fails or you rebalance.
- When a node stops, the primary moves its devices to the least loaded survivors in about 2 seconds. If the primary itself fails, the new primary keeps the placement and only re-places the old primary's devices.
- A node only runs the devices of the latest placement from the current primary, and a node that loses the majority stops its devices first, so two nodes never drive the same device in a cluster of three or more.

Intents and client connections still go to the primary, which forwards actions for a device placed elsewhere over the cluster link (one network hop, typically under 2 ms). Device state comes back the same way and is mirrored to every node.

API: `GET /api/v1/cluster/placement`, `POST /api/v1/cluster/rebalance`, `PUT /api/v1/project/cluster`; MCP `get_cluster_placement`, `rebalance_cluster`, `set_cluster_settings`; event `cluster.placement`. Shared mode needs no extra license: every node is a host, as in standby mode.

Hardware attached to one machine

Networked devices are driven by whichever node is primary. A device wired to one machine (serial port, USB DMX interface, LTC output) can only be driven from that machine. Give it a `node` in the project:

```JSON
{ "id": "dmx-usb", "name": "USB DMX", "moduleId": "misc.usbdmx.lighting", "node": "core-b",
"config": {} }
```

Only core-b runs that device's module, whatever its role. The primary sends actions for it to core-b over the cluster, and its state comes back the same way, so every node sees it.

If core-b is down, that device is offline and its actions fail with an error naming the node. Everything else continues. For anything that must survive a machine failure, prefer network gateways: Art-Net or sACN nodes, serial-over-IP converters.

On a standalone node the `node` field is ignored.

Check the cluster

`GET /api/v1/cluster` (scope `read`) or the MCP tool `get_cluster_status` returns this node's view:

```json
{
  "nodeId": "core-b",
  "role": "backup",
  "epoch": 3,
  "health": "ok",
  "synced": true,
  "diverged": false,
  "primary": "core-a",
  "size": 3,
  "quorum": 2,
  "clockOffsetMs": -0.42,
  "nodes": [
    { "nodeId": "core-a", "role": "primary", "epoch": 3, "health": "ok", "synced": true,
      "cueLog": { "count": 128, "digest": "..." }, "address": "10.0.0.1:7341", "ageMs": 40,
"alive": true },
    { "nodeId": "core-c", "role": "backup", "epoch": 3, "health": "ok", "synced": true,
      "cueLog": { "count": 128, "digest": "..." }, "address": "10.0.0.3:7341", "ageMs": 65,
"alive": true }
  ]
}
```

FIELD	MEANING
`role`	`primary`, `backup` or `standalone`.
`health`	`ok`, or `degraded` for a backup that cannot take over safely (different project, or out of sync).
`synced`	This backup holds a full, current copy of the primary's state.
`diverged`	This backup's cue log differed from the primary's; it is resyncing.
`primary`	Node id of the primary, or `null` if there is none.
`size`, `quorum`	Configured number of nodes, and how many a node must reach to be primary.
`nodes`	Every other node as last heard: role, epoch, health, sync state, cue log, address, heartbeat age and `alive`.

Before a show, check that every backup is `synced` and that cue log counts and digests match the primary's.

To be alerted when a node goes down, falls out of step or the cluster has no primary, add a `coreNode` alert rule in Sentinel.

Clients and the primary

Changes go to the primary. A backup refuses configuration changes, transport commands, intents, manual automation runs and Sentinel actions with `409 not_primary`. The answer includes the primary's address:

```JSON
{
  "type": "https://tangle.dev/problems/not_primary",
  "title": "Conflict",
  "status": 409,
  "detail": "this node is a backup; send changes to the primary",
  "code": "not_primary",
  "primary": "http://10.0.0.1:7340"
}
```

Reading (project, state, cluster status) works on every node.

Every node serves the dashboards (`/d`), end-user views (`/view`) and their live connection. A dashboard follows the primary by itself: see One address for operators.

The primary copies API tokens, including enrolled panel tokens, to every backup, so a client keeps working after a failover. Dashboard and view PIN sessions are signed with the cluster key and are accepted by every node.

One address for operators

Operators should not need to know which machine is the primary. Tangle gives you three layers. The first is always on. Add the others when you need them.

1. Any node's address works

Open a dashboard on any node, for example `http://10.0.0.2:7340/d/lobby`. The page asks that node for the other members of the cluster and remembers them on the device. Then:

- it connects to the primary, even when you opened a backup's address;
- when the primary fails, it reconnects to the new primary within a few seconds, without a reload and without asking for the PIN again;
- when the node you opened goes down, the page keeps working through the others, and after a reload it starts from the nodes it remembers.

You can still list the nodes yourself with the `cores` parameter, for example `/d/lobby?cores=10.0.0.1:7340,10.0.0.2:7340`. This helps a panel that has never reached the cluster before. See Dashboards.

The one limit: the browser must load the page from a node that is up. A bookmark to a machine that is switched off does not open. The cluster address fixes this.

2. The cluster address

Give the cluster a name in the designer, under **Nodes > Cluster address**, for example `riverside`. The current primary then answers for `riverside.local` on the local network, and operators bookmark `http://riverside.local:7340/d/lobby`.

- Only the primary answers. When a backup takes over, it announces the name within a second, and the old primary sends a goodbye if it is still running. A standalone core answers for the name too, so you can set it up before adding a backup.
- If another machine already uses the name, the Nodes page shows a conflict and Tangle does not announce it. Pick another name.
- `.local` names use mDNS (Bonjour). macOS, iOS, iPadOS and Linux with Avahi resolve them. Windows 10 and 11 resolve them in most browsers. Android support depends on the device, so give Android panels a node's IP address and let layer 1 do the failover.
- mDNS stays on one network segment. It does not cross routers or VLANs.

3. A floating IP or a reverse proxy

For sites with a network administrator, or panels that cannot resolve `.local`:

Floating IP (Linux, keepalived). One virtual IP moves to whichever machine holds it. Make keepalived follow the Tangle primary with a check script, so the IP and the primary stay on the same machine:

```sh
#!/bin/sh
# /usr/local/bin/tangle-primary: succeeds when this node is the Tangle primary.
# TOKEN is an API token with the read scope.
curl -sf -H "Authorization: Bearer $TOKEN" http://127.0.0.1:7340/api/v1/cluster | grep -q
'"role":"primary"'
```

```conf
# /etc/keepalived/keepalived.conf on every node
vrrp_script tangle_primary {
  script "/usr/local/bin/tangle-primary"
  interval 1
  fall 1
  rise 1
}
vrrp_instance TANGLE {
  state BACKUP
  interface eth0
  virtual_router_id 73
  priority 100
  advert_int 1
  virtual_ipaddress { 10.0.0.100/24 }
  track_script { tangle_primary }
}
```

Panels then use `http://10.0.0.100:7340`. keepalived needs root and runs on Linux only. On macOS and Windows, use a reverse proxy.

Reverse proxy on another machine (Caddy). The proxy sends requests to whichever node answers, and the dashboard then finds the primary by itself:

```caddyfile
:80 {
  reverse_proxy 10.0.0.1:7340 10.0.0.2:7340 10.0.0.3:7340 {
    lb_policy first
    health_uri /d
    health_interval 1s
    fail_duration 2s
  }
}
```

The proxy machine is then a single point of failure for the first page load, so keep it simple and on a UPS. Open dashboards are not affected if it fails, because they talk to the nodes directly.

Which one to use

SITE	USE
Small venue, Apple devices or Linux panels	Cluster address (`riverside.local`)
Mixed or Android panels	A node's IP address on each panel. Failover is automatic.
Venue with IT, VLANs or a strict network	Floating IP or reverse proxy, plus the cluster address for the crew

Failure scenarios

FAILURE	WHAT HAPPENS
Primary loses power or crashes	The first healthy, synced backup takes over in under a second with a higher epoch, at the right position. It sends cues the old primary may not have sent, if they were due in the last 2 seconds. No cue fires twice.
Network split, 3 nodes (2 and 1)	The side with the majority keeps or elects a primary. The single node has no quorum and stays backup; if it was primary, it steps down.
Network split, 2 nodes	Both become primary until the link is back. Then the higher epoch wins.
A backup computes different cues	It reports `diverged` and `degraded`, resyncs from the primary, and cannot take over until it is synced.
A backup runs a different project (with `--no-project-sync`)	It reports `degraded` and will not take over until the projects match.
Node with an attached device fails	That device is offline. Everything else continues.
A node comes back after a failover	It starts as backup, follows the current primary and syncs. It does not take the primary role back.

What a new primary does on promotion:

- Timelines continue from their position. Chasing timelines start their own chase, and timecode outputs are sent the transport again.
- Automations start running and `startup` triggers fire. Runs that were in progress on the old primary are not resumed, and time triggers missed during the outage are not replayed.
- Sentinel starts evaluating. Firing notifications in the first two minutes say "after failover".

Edition requirement

EDITION	REDUNDANCY
Lite	Not available.
Pro	With the redundancy add-on: the license sets how many redundant hosts are allowed.
Ultimate	Included, any number of redundant hosts.

EDITION	REDUNDANCY
Free designer, demo mode	Not available.

One license covers one installation: a main host plus its redundant hosts. Activate the same license key on each machine. The first machine becomes the main host, the next ones join as redundant hosts, up to the number the license allows. Each host keeps its own signed license file, so a backup takes over licensed with no network call.

If the license does not allow redundancy, or the cluster lists more other nodes than the license allows, core logs the reason and runs standalone. A machine activated as a redundant host runs licensed only as part of its cluster (with `--peer`).

See Licensing.

Recommendations

- Use 3 nodes. With 2 nodes, put them on a dedicated link.
- Give the machine you prefer as primary `--role primary` and every other node `--role backup`.
- Keep all nodes on the same version of tangle-core and the same module versions.
- Keep hardware that must survive a failure off single machines: use network gateways instead of `node`-attached devices.
- Before the show, fail over on purpose and check that every device comes online on each node.
- Install each node as a service so it restarts after a crash.
- Check `GET /api/v1/cluster` before each show, and add Sentinel `coreNode` rules so the crew hears about a lost backup.
- Give the cluster an address, and test a failover from an operator's tablet before the show.

GUIDES

Sentinel

Monitor devices, core nodes, network assets and custom metrics, raise alerts and notify people by email, SMS, push, webhooks, Slack or Teams.

Sentinel is the monitoring built into tangle-core. It watches devices, core nodes, probes and metrics that any script pushes, raises alerts from rules you define, and notifies people. There is nothing extra to install. Sentinel is part of the Pro and Ultimate editions.

What Sentinel watches

SOURCE	WHAT IT SEES
Devices	Each device's status (`online`, `offline`, `timeout`, `degraded`, `error`, `connecting`) and every state value its module reports, such as lamp hours or temperature.
Core nodes	In a redundant cluster: nodes that are down, nodes out of step, and a cluster with no primary.
Probes	TCP and HTTP checks of network assets that have no device module: a web server, a media server's control port, a switch.
Metrics	Numbers that any client pushes through the API, with labels, for example a room temperature from a sensor script.

Sentinel runs on the primary node of a cluster, or on a standalone node. Backups keep their view of the system up to date but do not evaluate rules or send anything until they become primary. Probes and notifications run in separate tasks with timeouts, so Sentinel never slows the show.

Where you configure it

Sentinel uses four kinds of project resources:

PROJECT KEY	WHAT IT HOLDS
`alertRules`	What to watch and when to raise an alert.
`notificationChannels`	Where notifications go.
`maintenanceWindows`	When notifications are muted.

PROJECT KEY	WHAT IT HOLDS
`probes`	TCP and HTTP checks.

Edit them in the designer's **Sentinel** section (`/designer/sentinel`), or through REST (`/api/v1/alertRules`, `/api/v1/notificationChannels`, `/api/v1/maintenanceWindows`, `/api/v1/probes`) and MCP (`create_alertRule`, `update_alertRule` and so on), like any other project resource.

Alert rules

JSON

```
{
  "id": "server-room-hot",
  "name": "Server room hot",
  "severity": "critical",
  "condition": { "kind": "metric", "name": "room.temperature", "labels": { "room": "server" },
"above": 30, "clear": 28 },
  "forSeconds": 60,
  "channels": ["ops-email"]
}
```

FIELD	REQUIRED	MEANING
`id`, `name`	yes	Identifier and display name.
`severity`	yes	`info`, `warning` or `critical`.
`condition`	yes	What to watch. See Alert conditions.
`forSeconds`	no	How long the condition must hold before the alert fires. Default 0.
`clearForSeconds`	no	How long the condition must be false before the alert resolves. Default 0.
`schedule`	no	Active hours: `{ "days": [...], "from": "HH:MM", "to": "HH:MM" }` in the node's local time. `from` later than `to` spans midnight. Outside the schedule, alerts of this rule resolve with reason `outside schedule`.
`channels`	no	Channel ids to notify. Empty or absent: every channel whose `minSeverity` allows this severity.

FIELD	REQUIRED	MEANING
`repeatMinutes`	no	Notify again every N minutes while the alert fires and nobody has acknowledged it.
`notifyOnResolve`	no	Send a notification when the alert resolves. Default `true`.
`message`	no	Message template with `{name}`, `{subject}`, `{value}` and `{severity}`.
`enabled`	no	Default `true`.
`description`	no	Free text.

A rule produces one alert per subject. A status rule over all devices raises one alert for each device that goes offline, not one alert for the whole rule. Alert ids look like `<ruleId>/<subject kind>:<subject id>`, for example `projectors-offline/device:projector-1`.

Alert conditions

`status`: devices whose status is in `statuses`. Default: `offline`, `timeout` and `error`.

```json
{ "kind": "status", "rooms": ["lobby"], "statuses": ["offline", "timeout"] }
```

`state`: a device state value compared to a threshold, with `above`, `below`, `equals` or `notEquals`. `clear` is the value that resolves the alert (hysteresis).

```json
{ "kind": "state", "rooms": ["room-a"], "capability": "lamp", "key": "hours", "above": 1800 }
```

`metric`: a pushed or probe metric compared to a threshold, per label set. Same comparisons and `clear` as `state`.

```json
{ "kind": "metric", "name": "probe.latencyMs", "labels": { "probe": "web" }, "above": 500 }
```

`stale`: no update for `seconds`. Watch a device state key (`capability` and `key`) or a metric (`metric` and `labels`).

```json
{ "kind": "stale", "seconds": 300, "metric": "room.temperature" }
```

A device that never reported the key is only watched when you name it in `devices`.

`coreNode`: problems in a redundant cluster, with `on`:

ON	FIRES WHEN
`down`	A core node is not heard.
`unsynced`	A node is out of step: degraded, diverged or not synced.
`noPrimary`	The cluster has no primary.

```json
{ "kind": "coreNode", "on": "down", "nodes": ["core-b", "core-c"] }
```

Without `nodes`, all nodes are watched. On a standalone node, `coreNode` rules never fire.

`status`, `state` and `stale` conditions select devices with `devices`, `rooms` or `modules` (module ids). With none of them, every device is watched.

Severities

SEVERITY	TYPICAL USE
`info`	Worth knowing, no action needed.
`warning`	Needs attention soon, such as lamp hours or a slow probe.
`critical`	Needs action now, such as a device offline during opening hours.

Channels have a `minSeverity`, so you can send everything to email and only critical alerts to SMS.

Alert lifecycle

An alert moves through these states:

1. **pending**: the condition holds, and `forSeconds` has not passed yet. If the condition clears now, the alert is dropped and nothing is sent.
2. **firing**: the condition held for `forSeconds`. A notification goes out.
3. **resolved**: the condition was false for `clearForSeconds` (and, with `clear`, the value passed the clear threshold). A resolve notification goes out if `notifyOnResolve` is on.

With `"above": 30, "clear": 28`, an alert fires at 31, stays firing at 29.5 and resolves at 27.9. This stops an alert from flapping around one value.

An alert also ends when its rule is disabled or removed, or its subject is removed from the project. These endings are not notified.

Notification channels

Every channel has these fields:

FIELD	MEANING
`id`, `name`	Identifier and display name.
`kind`	`email`, `sms`, `push`, `webhook`, `slack` or `teams`.
`settings`	Kind-specific, below.
`minSeverity`	Lowest severity this channel receives when a rule does not list channels.
`quietHours`	`{ "days", "from", "to", "includeCritical" }`. Non-critical notifications are dropped in this window; critical ones too when `includeCritical` is true.
`enabled`	Default `true`.

> **Note:** Passwords, tokens and webhook URLs marked secret are stored in the OS credential store of the machine, not in the project file. The project keeps `********` in their place.

Email

SETTING	MEANING
`host`	SMTP server. Required.
`port`	SMTP port.
`security`	`starttls` (default, port 587), `tls` (port 465) or `none` (port 25, trusted networks only).
`username`, `password`	SMTP login. `password` is secret.
`from`	Sender address. Required.
`to`	List of recipients. Required.

```json
{
  "id": "ops-email", "name": "Ops email", "kind": "email", "minSeverity": "warning",
  "settings": { "host": "smtp.example.com", "username": "alerts@example.com", "password":
"...", "from": "alerts@example.com", "to": ["ops@example.com"] }
}
```

The subject reads like `[Tangle Lobby] FIRING critical: Server room hot on room.temperature{room=server}`, where `Tangle Lobby` is the project name.

SMS (Twilio)

SETTING	MEANING
`provider`	`twilio`. Required.
`accountSid`	Twilio account SID. Required.
`authToken`	Twilio auth token. Secret.
`from`	Sender number (E.164, for example `+15145550100`) or messaging service id. Required.
`to`	List of phone numbers. Required.

SMS and push messages are shortened to 160 characters.

Push (ntfy, Pushover)

SETTING	MEANING
`provider`	`ntfy` or `pushover`. Required.
`server`	ntfy server. Default `https://ntfy.sh`.
`topic`	ntfy topic.
`token`	ntfy access token, optional. Secret.
`userKey`	Pushover user key.
`appToken`	Pushover application token. Secret.

Critical alerts are sent with a higher priority than warnings and info.

Webhook

SETTING	MEANING
`url`	Where to POST. Required.

SETTING	MEANING
`signingKey`	Optional. Secret. Signs each request body.
`headers`	Extra HTTP headers, as an object.

Core POSTs JSON:

```json
{
  "event": "firing",
  "site": "Tangle Lobby",
  "title": "[Tangle Lobby] FIRING critical: Server room hot on room.temperature{room=server}",
  "text": "room.temperature{room=server} is 31.5\n\nRule: Server room hot\nSeverity:
critical\n...",
  "severity": "critical",
  "alert": { "id": "server-room-hot/metric:room.temperature{room=server}", "state": "firing",
"...": "..." },
  "at": 1760362800000
}
```

`event` is `firing`, `repeat`, `resolved`, `summary` or `test`. With a `signingKey`, the request carries `X-Tangle-Signature: sha256=<hex>`, where `<hex>` is the HMAC-SHA256 of the raw body with the signing key. To verify, compute the same HMAC over the body you received and compare:

```js
import crypto from "node:crypto";

function verify(rawBody, header, key) {
  const expected = Buffer.from("sha256=" + crypto.createHmac("sha256",
key).update(rawBody).digest("hex"));
  const received = Buffer.from(header ?? "");
  return expected.length === received.length && crypto.timingSafeEqual(expected, received);
}
```

Slack

SETTING	MEANING
`webhookUrl`	A Slack incoming webhook URL. Secret: anyone with it can post.

Microsoft Teams

SETTING	MEANING
`webhookUrl`	A Teams workflow (Power Automate) webhook URL. Secret.

Teams receives an adaptive card coloured by severity.

Delivery, retries and rate limits

- Notifications go into a queue and are sent in the background. Each attempt has a 10-second timeout.
- A failed delivery is retried after 5 seconds, 30 seconds and 2 minutes, then recorded as failed in the history.
- Each channel sends at most 20 messages per minute. The rest are counted and summarised in one message once the minute is over ("20 more alerts were not sent to this channel in the last minute").

Test a channel

In the designer, open the channel and choose **Test**. Through the API:

```sh
curl -X POST http://127.0.0.1:7340/api/v1/sentinel/channels/ops-email/test \
  -H "Authorization: Bearer $TOKEN"
```

The test is sent at once, without queue or retry, and the answer reports success or the provider's error, for example an SMTP authentication failure.

Maintenance windows

A maintenance window mutes notifications. Alerts still fire and show in the designer, marked `muted`.

One-off window (times in milliseconds since the epoch):

```json
{ "id": "lamp-swap", "name": "Lamp swap", "devices": ["projector-1"], "start": 1760346000000,
"end": 1760353200000 }
```

Weekly window:

```json
{ "id": "nightly", "name": "Nightly shutdown", "rooms": ["lobby"], "days": ["mon", "tue",
"wed", "thu", "fri"], "from": "23:00", "to": "07:00" }
```

Scope a window with `rules`, `devices` and `rooms`. With none of them, the window mutes everything.

If a problem is still there when the window ends, Sentinel sends one firing notification for it.

Probes

Probes check network assets that have no device module.

```json
[
  { "id": "web", "name": "Booking site", "kind": "http", "url": "https://example.com/health",
"expectStatus": [200], "intervalSeconds": 30 },
  { "id": "nas", "name": "Media NAS", "kind": "tcp", "host": "10.0.0.20", "port": 445, "room":
"control" }
]
```

FIELD	MEANING
`kind`	`tcp` (connect to `host` and `port`) or `http` (request `url`).
`expectStatus`	HTTP only. Accepted status codes. Default 200 to 399.
`intervalSeconds`	Default 30, minimum 5.
`timeoutMs`	Default 5000, from 100 to 60000.
`room`	Optional room, for grouping on the health board and in maintenance windows.
`enabled`	Default `true`.

Each probe produces metrics labelled `probe=<id>`: `probe.up` (1 or 0), `probe.latencyMs`, and for HTTP `probe.httpStatus`. Alert on them with a `metric` condition:

```json
{ "id": "web-down", "name": "Booking site down", "severity": "critical", "forSeconds": 60,
  "condition": { "kind": "metric", "name": "probe.up", "labels": { "probe": "web" }, "below":
1 } }
```

Pushing metrics

Any script can push metric samples (scope `control`):

```sh
curl -X POST http://localhost:7340/api/v1/sentinel/metrics \
  -H "Authorization: Bearer $TOKEN" -H "content-type: application/json" \
  -d '[{"name":"room.temperature","value":31.5,"labels":{"room":"server"}}]'
```

Each sample has `name` (letters, digits, `.`, `_` and `-`, up to 128 characters), a numeric `value`, optional `labels` (string values) and an optional `at` timestamp in milliseconds. A label `room` that matches a room id ties the metric to that room.

Push to the primary node. Core keeps one sample per minute for 24 hours, and forgets a metric that has not been updated for a day. `GET /api/v1/sentinel/metrics` returns the current values; add `?samples=true` for the last 24 hours of minute samples.

Pushing also works over WebSocket (`sentinel.metrics` with `samples`) and MCP (`push_metrics`).

Acknowledging and silencing

Both need the `control` scope and are refused on a backup node.

Acknowledge an alert to stop its repeat notifications. The alert keeps firing until the condition clears and then resolves normally.

```sh
curl -X POST http://127.0.0.1:7340/api/v1/sentinel/ack \
  -H "Authorization: Bearer $TOKEN" -H "content-type: application/json" \
  -d '{"alertId":"lamp-hours/device:projector-1","note":"lamp ordered"}'
```

Silence one alert, or every alert of a rule (current and future), until a time in milliseconds since the epoch. Silenced alerts send no notifications.

```sh
curl -X POST http://127.0.0.1:7340/api/v1/sentinel/silence \
  -H "Authorization: Bearer $TOKEN" -H "content-type: application/json" \
  -d '{"ruleId":"projectors-offline","until":1760400000000}'
```

In the designer's **Alerts** tab, acknowledge with a note, or silence for 15 minutes, 1 hour, 4 hours or 1 day.

History

Every alert transition is recorded: `pending`, `firing`, `resolved`, `dropped`, `repeat`, `acknowledged`, `silenced`, `notified` and `notifyFailed` (with the channel and the error).

- `GET /api/v1/sentinel/alerts?history=true` returns current alerts and the latest transitions, newest first. `?limit=` sets how many (up to 500).
- Events: subscribe to `sentinel.alert:<ruleId>` (or `sentinel.alert:*`).
- On disk, history is a JSON Lines file at `<data>/sentinel/<project id>/history.jsonl` in the core data folder (see Operations). It rotates at 5 MB, keeping one old file, and survives restarts.

The designer's Sentinel section

Open **Sentinel** in the designer. It has these tabs:

TAB	WHAT IT SHOWS
Health	Every core node, every room with its devices, and every probe, coloured by health, updated live.

TAB	WHAT IT SHOWS
Alerts	Current (firing and pending) alerts with acknowledge and silence actions, and the recent history with notification results.
Alert rules	Rule editor with device, room, capability and key pickers.
Channels	Channel editor with masked secrets and a **Test** button.
Maintenance	Maintenance windows.
Probes	TCP and HTTP probes.

On a backup node the section says the core is a backup and that the primary sends notifications. Without a monitoring license it says so; you can still set up rules and channels.

API and MCP reference

ACTION	REST	MCP	WEBSOCKET	SCOPE
Status: counts by severity, firing alerts, health of devices, core nodes and probes	`GET /api/v1/sentinel`	`get_sentinel_status`	`sentinel.status`	read
Current alerts and history	`GET /api/v1/sentinel/alerts`	`list_alerts`	`sentinel.alerts`	read
Acknowledge	`POST /api/v1/sentinel/ack`	`ack_alert`	`sentinel.ack`	control
Silence	`POST /api/v1/sentinel/silence`	`silence_alerts`	`sentinel.silence`	control
Push metrics	`POST /api/v1/sentinel/metrics`	`push_metrics`	`sentinel.metrics`	control
Read metrics	`GET /api/v1/sentinel/metrics`	`get_metrics`		read

ACTION	REST	MCP	WEBSOCKET	SCOPE
Test a channel	`POST /api/v1/sentinel/channels/{id}/test`	`test_channel`		configure

See REST API, WebSocket and MCP.

Edition

Sentinel needs the Pro or Ultimate edition (the `MONITORING` entitlement). It also runs in demo mode. On Lite and in the free designer:

- No rules are evaluated, no probes run and nothing is sent.
- `GET /api/v1/sentinel` reports `licensed: false`.
- Acknowledge, silence, metric pushes and channel tests answer `403 not_licensed`.
- Alert rules, channels, maintenance windows and probes stay editable, so you can set them up before upgrading.

Turning monitoring on or off never affects devices, timelines, automations or cues. See Licensing.

Coming later: Tangle Cloud

Tangle Cloud, a hosted remote monitoring service, is not available yet. It is planned to receive heartbeats, alerts and metrics from each site, alert when a whole site goes silent, and give a view across many sites. Local Sentinel works on its own and does not need it.

GUIDES

Licensing

Tangle editions and what each unlocks, activating and moving a license, offline use, free designer and demo modes, and expiry.

A license unlocks a Tangle edition on a computer that runs tangle-core. Without a license, core runs as the free designer or in demo mode. Licensing is offline first: each machine keeps a signed license file and trusts it with no network call, and a license problem never interrupts a running show.

Editions

	LITE	PRO	ULTIMATE
Controlled devices	20	100	unlimited
DMX universes	2	unlimited	unlimited
Timelines and cue lists	2 in total	unlimited	unlimited
Timecode chase and generation (LTC, MTC, Art-Net timecode)	no	yes	yes
Clock sync (PTP)	no	yes	yes
Monitoring (Sentinel)	no	yes	yes
External API (REST and MCP for third-party software)	no	yes	yes
Redundancy	no	add-on, per redundant host	included, any number of hosts
Remote clients (dashboards and designer from other machines)	yes	yes	yes

DMX universes are counted as the distinct `universe` numbers in device settings (Art-Net, sACN and DMX modules put the universe there).

"External API" means REST and MCP calls made with your own API tokens by other software. These always work on every edition:

- The built-in designer and dashboards served by core.

- The WebSocket client protocol.
- The `tangle-core` command line on the machine.

On Lite, an external REST or MCP call is refused with `403 not_licensed`.

License models

A license is one of these models. The model sets how long it runs and how it moves; the edition sets what it unlocks.

MODEL	MEANING
Perpetual	Node-locked. Runs for good, and includes updates released until the date shown as `updates until`.
Floating	Used by one installation at a time. Can move to another installation once every 7 days.
Subscription	Runs until its end date, with 14 days of grace.
Education	Like a subscription, for teaching.

Free designer and demo modes

Without a license, core runs in one of two modes. You choose when you start core.

	FREE DESIGNER (DEFAULT)	DEMO (`--DEMO`)
Use	Offline programming	Trying the full product
Devices	unlimited	up to 10
Features	Timecode and sync; no monitoring, no external API, no redundancy	Everything except redundancy
Network	This computer only: core listens on 127.0.0.1, and remote access options are turned off	Remote clients allowed
Sessions	Pauses every hour	Pauses every hour

```sh
tangle-core run --project show.json          # free designer
tangle-core run --project show.json --demo   # demo mode
```

At the end of each hour, core pauses playing timelines, disconnects devices and logs that the session hour is over. To continue for another hour, resume the session:

- REST: `POST /api/v1/license/resume` (scope `control`)
- MCP: `resume_session`
- WebSocket: `license.resume`

These session limits apply only to unlicensed starts. A licensed core never falls into them, even if its license lapses while it runs.

Activate a license

Activation needs internet access on the machine, unless you use a license request by email or offline activation.

In the designer

Open **Settings > License**. It shows what core runs as, the license (with **Licensed to**: the customer it was issued to), this computer's role, the limits, the features and the dates, and whether the license server can be reached. Then:

- **License key**: paste the key and click **Activate**.
- **Sign in with email**: sign in with the email and password of your Tangle license account, pick the license if you have several, and Tangle activates it. Neither the password nor the session is kept. **Forgot password?** emails a reset link.
- **License by email / file...**: for a computer that cannot reach the license server (see License by email).
- **Refresh now**, **Deactivate**, and **Offline (never contact the license server)**.

These need a token with the `admin` scope; the desktop app's own token has it.

On the command line

```sh
tangle-core license activate <key>
tangle-core license activate <key> --name "Lobby rack"
```

`--name` is the name shown on the license (default: the computer's name). The license applies from the next start of core: restart core or its service.

The first machine you activate creates the installation and becomes its main host. With redundancy, activate the same key on each other node: they join as redundant hosts, up to the number the license allows. If the license has no room left, activation fails and says how to free a place.

> **Note:** License files live in the core data folder (see Operations). A core installed as a boot service on macOS or Linux uses the machine-wide folder, so run the license commands with `sudo` there, for example `sudo tangle-core license activate <key>`.

Check the license

```sh
tangle-core license status
tangle-core license status --json
```

The status shows the state, edition, model, the last characters of the key, host role (main or redundant), device and redundant host limits, features, and the dates that matter: `expires`, `updates until`, and `refresh before`. `--json` also shows what the next start will run as (`nextStart`).

License states:

STATE	MEANING
`valid`	Licensed.
`expiring`	Licensed, but it ends soon, is in its grace period, or its file must be refreshed soon.
`expired`	The license or its file ended: unlicensed from the next start.
`invalid`	The file is missing, tampered with, for another computer, or the license is suspended.
`none`	No license on this computer.

Refresh a license

When online, core fetches a fresh license file a minute after it starts and then every 24 hours (and sends the heartbeats floating licenses need), retrying every hour after a failure. A failure never stops the show: the signed file on disk keeps working until it expires. To refresh by hand, for example after a renewal or an upgrade:

```sh
tangle-core license refresh
```

The new file takes effect at the next start.

How long a license file stays valid without going online:

MODEL	LICENSE FILE VALID FOR
Perpetual	No limit
Floating	7 days
Subscription, education	30 days

Two days before the file expires, the state becomes `expiring` with the date to connect before.

Move a license

To move a node-locked license to another computer, deactivate it on the old one, then activate it on the new one:

```sh
tangle-core license deactivate       # on the old computer
tangle-core license activate <key>   # on the new computer
```

`deactivate` removes this computer from its installation and deletes the local license file. The computer runs as the free designer (or `--demo`) from its next start.

To move a floating license, run this on the new computer:

```sh
tangle-core license move <key>
```

A floating license can move once every 7 days. A second move within a week is refused. The old installation keeps running its current project; it becomes unlicensed at its next start once its license file expires.

With an email and password

```sh
tangle-core license sign-in owner@example.com              # password typed on standard input
tangle-core license sign-in owner@example.com --license <id>   # when the account has several licenses
```

The password can also come from `TANGLE_LICENSE_PASSWORD`.

License by email

A computer that cannot reach the license server but can send email (from another machine) can ask for its license file by email:

1. In **Settings > License**, click **License by email / file...** and export a request, or on the command line:

   ```sh
   tangle-core license request --email owner@example.com --out lobby.request    # or --key <key>
   ```

2. Email the request file to licensing@tangle.systems from the address the license is registered to. The answer is a license file.
3. Install the file with **License by email / file...**, or `tangle-core license import <file>`.

Offline mode

Tick **Offline (never contact the license server)** in **Settings > License** (REST `PUT /api/v1/license/offline`) for a computer that must never reach the internet. Core then keeps using its license file and is renewed with files, by email or by offline activation.

Offline activation

For a computer with no internet access:

1. On the offline computer, print its id:

   ```sh
   tangle-core license fingerprint
   ```

2. On a computer with internet access, write a license file for that id:

   ```sh
   tangle-core license issue <key> --fingerprint <id> --out lobby.lic
   ```

   `--name` sets the name shown on the license (default "offline host").

3. Copy the file to the offline computer and install it:

   ```sh
   tangle-core license import lobby.lic
   ```

Core checks the file's signature with a key built into core, with no network call. A file edited after signing is rejected, and a file issued for another computer is rejected with `fingerprint mismatch`.

Floating, subscription and education licenses need a fresh file before their file validity runs out (see Refresh a license). Repeat these steps to bring one.

> **Warning:** If the computer's clock is set more than 24 hours behind the last time Tangle ran, the license counts as invalid. Keep the clock correct.

What happens when a license expires

What the license allows is decided once, when core starts and loads the project. A license that expires, is suspended, moves away, or cannot be validated while core runs does not stop, pause or limit the running show. Devices, timelines, automations and cues continue unchanged.

- Core publishes a `license.status` event when the license changes, and `GET /api/v1/license` reports `pendingRestart: true`.

- The change takes effect at the next start. An expired license then runs as the free designer, or in demo mode with `--demo`.
- Subscriptions and education licenses warn 14 days before they end, and keep working for 14 days of grace after the end date. A perpetual license never expires.

Limits are checked when the project loads

Core checks the project against the edition when it starts:

- A project that goes over a limit (devices, DMX universes, timelines) or uses a feature the edition lacks (timecode, PTP) is refused, with a message naming the limit and the edition that includes it. For example: "Tangle Lite allows 2 timelines and cue lists in total; this project has 3 (upgrade for unlimited timelines)".
- A change made while core runs that would go over a limit is refused with `403 not_licensed`.
- If the license does not allow redundancy, or the cluster lists more nodes than allowed, core runs standalone and says why.

These checks happen at load and on configuration changes, never on the cue path.

License status in the API

`GET /api/v1/license` (scope `read`), MCP `get_license`, or WebSocket `license.status` returns:

FIELD	MEANING
`mode`	`licensed`, `designer` or `demo`.
`status`	The license: `state`, `licensed`, `model`, `edition`, `entitlements`, `limits`, `expiry`, `maintenanceUntil`, `graceUntil`, `fileExpiry`, `installationId`, `hostRole`, `keyHint`, `reason`.
`features`	What this start allows: edition label, device, universe and timeline limits, and whether timecode, sync, monitoring, external API, remote clients and redundancy are on.
`pendingRestart`	The license changed since start; the change applies at the next start.
`notes`	What was turned off at start and why.
`session`	Free designer and demo only: `paused` and `endsAt`.

GUIDES

Simulator

Run virtual projectors, media players and servers, OSC and line-based devices, serial gateways and DMX receivers with tangle-sim to build and test a show without hardware.

tangle-simulator runs virtual devices for testing Tangle without hardware. Each virtual device speaks its real protocol on a real network port, so the real Tangle modules (and any other controller) connect to it exactly as they would to the device. Core needs no special mode and does not know it is talking to a simulator.

Use it to program and demo a show before the gear is on site, and to rehearse failures: a projector that goes offline mid-show, a hung device, a wrong password.

Get it

The simulator is a free tool, with no license check. It comes in three forms:

FORM	NEEDS	START IT WITH
Standalone executable `tangle-sim` (Node.js built in). On macOS it is also in the Tangle .dmg next to Tangle.app	Nothing	`./tangle-sim run examples/demo.json`
Package `tangle-simulator-<version>.tar.gz` (one script, web UI, examples)	Node.js 20 or later	`node tangle-sim.mjs run examples/demo.json`
Source, in the `tangle-simulator` folder of the Tangle source tree	Node.js 22.6 or later, pnpm	see below

From source:

```sh
pnpm install
cd tangle-simulator
node bin/tangle-sim.mjs init      # writes tangle-sim.json: example devices and scenarios
node bin/tangle-sim.mjs run       # starts them
pnpm dev                          # runs examples/demo.json directly
```

On this page, `tangle-sim` stands for whichever form you use.

When it starts, the simulator prints a table of devices with the address and port each one listens on, then the address of the web UI: http://127.0.0.1:4380.

> **Note:** On macOS, the first run asks whether the program may accept incoming connections. Allow it if tangle-core runs on another machine.

Connect Tangle to it

Add a device in Tangle with the matching module and point it at the simulator: `host` = `127.0.0.1` (or the simulator machine's IP address), `port` = the port shown at start. Tutorial: a gallery installation does this in the designer for two projectors and a VLC player. On the command line, with the PJLink module:

```sh
tangle device add -p show.yaml pj-1 -m misc.pjlink.projector --set host=127.0.0.1 --set port=4352
tangle device test -p show.yaml pj-1
```

Virtual devices are also found by Tangle discovery (`tangle discover`, or **Scan the network** in the designer): PJLink class 2 search, mDNS advertisements, SSDP answers and Art-Net ArtPoll replies. Each virtual device reports manufacturer `Tangle` and a stable fake MAC address derived from its id.

Device types

`tangle-sim types` lists every type and its options.

TYPE	WHAT IT IS	DEFAULT PORT
`pjlink`	PJLink projector: power with warm-up and cool-down, inputs, picture and audio mute, lamp hours, error status, password, class 2 search	TCP 4352
`ascii`	Any line-based device over TCP or UDP, described by rules (regular expression match, set state, reply template)	5000
`osc`	OSC device over UDP, or TCP (SLIP or length-prefixed framing): keeps the last arguments at each address and answers a message without arguments with them	UDP 8000
`moxa.nport.gateway`	IP-to-serial gateway (Moxa NPort style) with a line-based device on its serial port (`rules`, as for `ascii`). `mode`: `tcp` (raw socket), `rfc2217` (the client sets the line; a client with the wrong baud rate, parity, data or stop bits gets noise, as on a real line) or `udp`. Answers the Moxa search on UDP 4800	TCP/UDP 4001
`dmx`	Art-Net and sACN receiver: shows the 512 levels per universe, the source and frames per second; answers ArtPoll	UDP 6454 / 5568
`videolan.vlc.player`	VLC: Lua HTTP interface (password `secret` by default) with a playlist (Intro, Loop, Credits), and the RC / telnet console (`rcPort`, `rcPassword`)	TCP 8080
`resolume.arena.mediasrver`	Resolume Arena / Avenue: REST webserver and OSC input (`oscPort`)	TCP 8080

TYPE	WHAT IT IS	DEFAULT PORT
`obs.obs.switcher`	OBS Studio over obs-websocket 5 (`password`); `fail` = stream reconnecting	TCP 4455
`disguise.d3.mediaserver`	disguise director: HTTP API, OSC transport (`oscPort` 7400) with feedback to `feedbackPort` (7401), JSON Machine Control (`machinePort` 9864); `fail` = health problem	TCP 8180
`greenhippo.hippotizer.mediaserver`	Hippotizer V4 REST API component and an OSC input (`oscPort`)	TCP 40512
`avstumpfl.pixera.mediaserver`	Pixera JSON/TCP API: timelines Main (60 fps) and Pre-show (25 fps) with cues; the state shows each timeline's timecode	TCP 1400

The player and media server types run the simulators that ship with their modules, so `tangle-sim` needs the `tangle-connect` folder next to it when run from source (the release package includes them). Their state is read-only in the web UI: drive them through their protocol. Their faults are `offline`, `silent`, `fail` (where noted), `drop` and `reboot`; `latencyMs` and `rejectAuth` have no effect on them. Extra ports left empty are picked at random and shown under `ports`. `examples/video-servers.json` starts all five media servers, and `examples/serial-gateways.json` a projector behind a gateway in each mode (in Tangle, use the generic serial module with `tcp://127.0.0.1:4001`, `rfc2217://127.0.0.1:4002` at 19200 baud, even parity, or `udp://127.0.0.1:4003?listen=4103`).

pjlink

Implements PJLink class 1 commands (`POWR`, `INPT`, `INST`, `AVMT`, `LAMP`, `ERST`, `NAME`, `INF1`, `INF2`, `INFO`, `CLSS`) with the standard answers and errors (`ERR3` for inputs while in standby, for example), the MD5 password check, and class 2 search on UDP. Lamp hours increase while the lamp is on. Idle connections close after 30 s, like a real projector.

OPTION	DEFAULT
`password`	none
`inputs`	`["11","12","31","32","51"]`
`warmSeconds`, `coolSeconds`	5
`lampSpeed`	1 (lamp hours per real hour)
`class`	2

OPTION	DEFAULT
`search`, `searchPort`	true, 4352
`idleSeconds`	30
`manufacturer`, `model`	`Tangle`, `PJLink Simulator`
`serial`, `otherInfo`	

State keys: `power` (`off`, `warming`, `on`, `cooling`), `input`, `videoMute`, `audioMute`, `lampHours`, `errors` (6 digits: fan, lamp, temperature, cover, filter, other; 0 ok, 1 warning, 2 error), `projectorName`.

ascii

Simulates a line-based controller protocol (a matrix switcher, a serial-over-IP device) from rules. The first rule whose `match` matches a received line wins: `set` changes state keys (captures `$1`, `$2` ... are allowed in keys and values), and `reply` is sent back, with captures and `{stateKey}` values filled in. Unmatched lines get `unknownReply`.

OPTION	DEFAULT
`transport`	`tcp` (or `udp`)
`delimiter`	`"\r\n"`, added after every answer
`inputDelimiter`	CR, LF or CRLF
`greeting`	sent when a client connects (TCP)
`rules`	`[{match, flags?, set?, reply?}]`
`unknownReply`	`"ERR"` (`null` for no answer)
`failReply`	`"ERR FAIL"`, the answer to everything while the `fail` fault is on
`push`	`{stateKey: "template"}`, sent to every client when that key changes
`mdns`	`tangle-sim` (service type without `_`), `false` to not advertise
`idleSeconds`	never
`manufacturer`, `model`	shown in discovery and the UI

dmx

Receives Art-Net (ArtDmx) and sACN (E1.31, joining the multicast group of each universe) and shows, per universe, the 512 levels, the source address and name, the sequence and the frames per second. It answers

ArtPoll with an ArtPollReply naming the device and its universes.

OPTION	DEFAULT
`artnet`, `artnetUniverses`	true, `[0]` (Art-Net port-addresses)
`sacn`, `sacnUniverses`	true, `[1]`
`artnetPort`, `sacnPort`	6454, 5568
`manufacturer`, `model`	shown in the UI

The config file

The simulator reads a JSON file, `tangle-sim.json` by default. `tangle-sim init [file]` writes a working example with two PJLink projectors, a video matrix (ASCII over TCP), a DMX receiver and three scenarios.

```json
{
  "ui": { "host": "127.0.0.1", "port": 4380 },
  "discovery": { "mdns": true, "ssdp": true },
  "devices": [
    { "id": "pj-1", "type": "pjlink", "name": "Hall projector", "warmSeconds": 5 },
    { "id": "pj-2", "type": "pjlink", "name": "Stage projector", "password": "secret",
"state": { "lampHours": 1790 } },
    {
      "id": "matrix",
      "type": "ascii",
      "name": "Video matrix 4x2",
      "port": 5023,
      "state": { "out1": 1, "out2": 1 },
      "rules": [
        { "match": "^([1-4])\\*([12])!$", "set": { "out$2": "$1" }, "reply": "Out$2 In$1 All"
},
        { "match": "^([12])!$", "reply": "{out$1}" }
      ],
      "unknownReply": "E10"
    },
    { "id": "dimmers", "type": "dmx", "name": "Dimmer rack", "artnetUniverses": [0, 1],
"sacnUniverses": [1] }
  ],
  "scenarios": {
    "stage-pj-dies": {
      "description": "Stage projector turns on, then goes offline 10 s later",
      "steps": [
        { "after": 0, "device": "pj-2", "state": { "power": "on" } },
        { "after": 10000, "device": "pj-2", "faults": { "offline": true } },
        { "after": 30000, "device": "pj-2", "faults": { "offline": false } }
      ]
    }
  }
}
```

Every device has an `id` (letters, digits, `.`, `_`, `-`), a `type`, and optionally `name`, `address` (IPv4, default `0.0.0.0`), `port`, `state` (initial state), `faults`, `discovery` (default `true`) and the options of its type.

- **Ports.** Devices without a `port` get their protocol's port, or the next free one: ten `pjlink` devices without ports start on 4352 to 4361.
- **Separate addresses.** For discovery to list several devices of one type separately, give each its own `address`. On Linux every `127.x.y.z` works already. On macOS add an alias first: `sudo ifconfig lo0 alias 127.0.0.2`, then `"address": "127.0.0.2"`.
- **Errors.** A bad config stops the start with the path of the wrong field, for example `devices[3].port: must be an integer from 1 to 65535`. A device whose port is taken by another program shows status `error` with the reason; the other devices keep running.

Faults

Every device has faults you can switch on from the web UI, the CLI, the API, the config file or a scenario. They apply at once.

FAULT	EFFECT	WHAT A TANGLE MODULE REPORTS
`offline`	Connections refused, open ones closed, UDP ignored	`offline`
`silent`	Accepts and reads, never answers (a hung device)	`timeout`
`latencyMs`	Every answer delayed by this many milliseconds, in order (a slow device or network)	slow actions, `timeout` past the limits
`fail`	Device failure. PJLink: lamp error and `ERR4`, projector switches off; ascii: `failReply`; dmx: stops updating levels	depends on the module, often `degraded`
`rejectAuth`	Password always rejected. PJLink: `PJLINK ERRA`	`error`

Two actions act once:

- `drop`: cut every open connection; the device stays up.
- `reboot` (seconds, default 5): drop connections, stay offline for that time, then come back with the initial state from the config (a projector comes back off).

Scenarios

A scenario is a named list of timed steps. Each step has `after` (milliseconds from the scenario start), a `device`, and one or more of `state`, `faults` or `action` (`drop` or `reboot`, with `seconds` for reboot). Optional scenario fields: `description`, `autoStart: true` (start when the simulator starts) and `loop: true` (start again after the last step).

```JSON
"slow-network": {
  "description": "Every device answers 800 ms late for 30 s",
  "steps": [
    { "after": 0, "device": "pj-1", "faults": { "latencyMs": 800 } },
    { "after": 30000, "device": "pj-1", "faults": { "latencyMs": 0 } }
  ]
}
```

Start and stop scenarios from the web UI, with `tangle-sim scenario <name>` and `tangle-sim scenario <name> stop`, or through the API. Step timing is within about 10 ms: good for testing, not a show clock.

Web UI

Open http://127.0.0.1:4380. The page shows one card per device with its type, id, address and port, status, live state, faults, actions and protocol log (the last 200 lines). For `dmx` devices the card shows the channel levels. Changes appear without reloading, so you can watch a Tangle dashboard turn a projector on and see `%1POWR 1` arrive in its log.

From a card you can:

- edit the device's state;
- toggle **Offline**, **Silent**, **Fail** and **Reject auth**, and set **Latency** in ms;
- **Drop connections** or **Reboot 5 s**;
- **Edit** or **Remove** the device.

The Scenarios section lists the scenarios with buttons to start and stop them. A **Skin** menu switches between the Console and Lobby looks; the page follows the OS light or dark mode.

Add and edit devices

Click **Add device**, pick a **Type**, then fill in **ID**, **Name**, **Address**, **Port** (empty for automatic), **Discoverable**, the type's **Settings** (with help text and example rules) and an optional **Initial state** as JSON. **Edit** opens the same form with the device's settings as written in the config file; saving restarts the device with its id and log kept.

Every add, edit or remove is saved to the config file, keeping its other settings. Automatic ports are not written, so a restart gives the same devices. Without a config file, `tangle-sim run` starts with no devices and creates `tangle-sim.json` on the first change.

The form checks values like the config file does: an id already in use shows `id: "pj-1" is already used` and the form stays open. A device used by a scenario cannot be removed; the message names the scenario.

Command line

```sh
tangle-sim run [config.json]          # default tangle-sim.json
tangle-sim init [file]                # write the example config
tangle-sim types                      # device types and their options
```

`run` options:

OPTION	EFFECT
`--ui-host <ip>`, `--ui-port <n>`	Where the web UI and API listen (default `127.0.0.1:4380`)
`--no-mdns`, `--no-ssdp`	Do not advertise devices

OPTION	EFFECT
`-q`, `--quiet`	Do not print protocol lines

Control a running simulator (these call its API; `--url` or `TANGLE_SIM_URL` changes the address, default `http://127.0.0.1:4380`):

```sh
tangle-sim status                                  # devices, ports, status and faults
tangle-sim show pj-1                               # full state and recent log
tangle-sim set pj-1 lampHours=1990 errors=010000   # change state
tangle-sim fault pj-1 offline                      # offline=off to clear; silent, fail,
rejectAuth, latency=500
tangle-sim clear pj-1                              # remove every fault
tangle-sim drop pj-1
tangle-sim reboot pj-2 10
tangle-sim add pjlink pj-3 name="Lobby PJ" password=secret
tangle-sim remove pj-3
tangle-sim scenario                                # list scenarios
tangle-sim scenario stage-pj-dies                  # start one
```

HTTP API

The web UI and the CLI use a local HTTP API, handy for test scripts.

METHOD	PATH	WHAT IT DOES
GET	`/api/devices`	Every device
GET	`/api/devices/{id}`	One device, with its log
GET	`/api/devices/{id}/config`	Its settings as written in the config file
GET	`/api/types`	Device types and their options
POST	`/api/devices`	Add a device
PUT	`/api/devices/{id}`	Replace its settings (restarts it)
DELETE	`/api/devices/{id}`	Remove it
PATCH	`/api/devices/{id}/state`	Change state (JSON body)
PATCH	`/api/devices/{id}/faults`	Change faults (JSON body)
POST	`/api/devices/{id}/actions/drop`	Cut connections
POST	`/api/devices/{id}/actions/reboot`	Reboot, body `{"seconds": 5}`

METHOD	PATH	WHAT IT DOES
GET	`/api/scenarios`	Scenarios and whether they run
POST	`/api/scenarios/{name}/start`, `/api/scenarios/{name}/stop`	Start or stop a scenario
GET	`/api/events`	Server-Sent Events: `device`, `log`, `scenarios`

```sh
curl -X PATCH http://127.0.0.1:4380/api/devices/pj-1/faults -H 'content-type:
application/json' -d '{"offline": true}'
```

Warning: The API has no authentication. It listens on 127.0.0.1 unless you pass `--ui-host`; only do that on a trusted network.

Limits

- Local serial (RS-232) ports are not simulated; `moxa.nport.gateway` covers devices behind IP-to-serial gateways.
- Timecode generators and video or audio streams are not simulated. Media servers report their timelines' timecode, which Tangle timelines can chase.
- New device types are classes extending `VirtualDevice` in the simulator source (`src/device.ts`, registered in `src/types/index.ts`).

GUIDES

Operations

Run tangle-core in production: run options, services, file locations, logs, API tokens, remote access, backups, ports and upgrades.

This page covers running tangle-core on a show or building machine: how to start it, install it as a service, where it keeps its files, how to secure access, and how to back up and upgrade it.

Run a project

```sh
tangle-core run --project /shows/main.yaml
```

Core loads the project, starts the device modules and serves every client surface on port 7340:

SURFACE	ADDRESS
Designer	`http://127.0.0.1:7340/designer`
Dashboards	`http://127.0.0.1:7340/d/<dashboard id>`
REST API	`http://127.0.0.1:7340/api/v1/...`, interactive docs at `/api/docs`, OpenAPI at `/api/v1/openapi.json`
WebSocket	`ws://127.0.0.1:7340/ws`
MCP	`http://127.0.0.1:7340/mcp`

Before a show, check the project file without running it:

```sh
tangle-core check --project /shows/main.yaml
```

It exits with 0 when the project is valid and prints the number of rooms, devices, scenes and timelines.

`tangle-core run` options

OPTION	MEANING
`-p`, `--project <file>`	Project file (`.json`, `.yaml` or `.yml`). Required.

OPTION	MEANING
`-m`, `--modules <folder>`	Extra folder of modules, one sub-folder per module. Repeatable. Installed modules are always used; a module in an extra folder wins over an installed one with the same id.
`--listen <addr:port>`	Address to serve clients on. Default `127.0.0.1:7340`.
`--allow-remote`	Allow listening on a network address, so other machines can connect.
`--insecure-http`	Accept unencrypted remote connections. Required with a network address until TLS is available.
`--node-id <id>`	Name of this node, shown to clients. Default `core-1`. Required and unique per node with redundancy.
`--ui <folder>`	Folder of the built dashboard app. Default: the `ui` folder next to the `tangle-core` program. Also `TANGLE_UI`.
`--demo`	Without a license, run in demo mode instead of the free designer. See Licensing.
`--log-json`	Write console logs as JSON lines.
`--log-dir <folder>`	Log folder. Default: the platform's log folder.
`--peer`, `--role`, `--cluster-listen`, `--cluster-key`, `--cluster-key-file`, `--no-project-sync`, `--no-discovery`	Redundancy. See Redundancy.

Other commands:

COMMAND	WHAT IT DOES
`tangle-core check -p <file>`	Validate a project file and exit.
`tangle-core modules [-m <folder>]`	List the device modules found, with version and folder.
`tangle-core token create/list/revoke`	Manage API tokens. See API tokens.
`tangle-core license ...`	Activate, check, refresh or move the license. See Licensing.
`tangle-core service install/uninstall`	Run at boot. See Run as a service.

COMMAND	WHAT IT DOES
`tangle-core mcp`	MCP over stdio for desktop assistants, forwarding to a running core. See MCP.

Stopping

Ctrl-C, or SIGTERM on macOS and Linux (what launchd and systemd send), stops core cleanly: modules get a `shutdown` message, and core exits within about 5 seconds.

Run as a service

A service starts at boot, restarts after a crash and stops cleanly. Core installs itself as a launchd daemon on macOS, a systemd unit on Linux, or a Windows Service. Put the `run` options after `--`.

macOS / Linux:

```sh
sudo tangle-core service install -- --project /shows/main.yaml --node-id core-a
```

Windows (administrator PowerShell):

```sh
tangle-core service install -- --project C:\Shows\main.yaml
```

To accept connections from other machines, add `--listen 0.0.0.0:7340 --allow-remote --insecure-http` to the options.

What `service install` does:

- Checks the options and loads the project first, so a mistake fails now and not at boot.
- Makes the project, module, log and UI paths absolute (services do not start in your current folder).
- Keeps the environment variables `PATH`, `TANGLE_HOME`, `TANGLE_PYTHON`, `TANGLE_NODE` and `TANGLE_LOG`, so `node` and `python3` from Homebrew or a virtualenv are still found.
- With redundancy, takes the shared key from `TANGLE_CLUSTER_KEY` (or `--cluster-key`) and writes it to `cluster.key` in the data folder, readable only by the service, instead of putting it in the service definition.
- Installs and starts the service, named `tangle-core`.

	SERVICE DEFINITION
macOS	`/Library/LaunchDaemons/tangle-core.plist` (with `--user`: `~/Library/LaunchAgents/tangle-core.plist`)

	SERVICE DEFINITION
Linux	`/etc/systemd/system/tangle-core.service` (with `--user`: `~/.config/systemd/user/tangle-core.service`)
Windows	Windows Service `tangle-core` ("Tangle core")

With `--user` (macOS and Linux), the service runs for the current user and starts at login instead of boot.

To remove the service:

```sh
sudo tangle-core service uninstall          # macOS, Linux
sudo tangle-core service uninstall --user   # a --user service
tangle-core service uninstall               # Windows, administrator PowerShell
```

To restart it with your operating system's tools:

	COMMAND
macOS	`sudo launchctl kickstart -k system/tangle-core`
Linux	`sudo systemctl restart tangle-core`
Windows	`Restart-Service tangle-core`

Note: A boot service on macOS and Linux runs as root and uses the machine-wide folders (see File locations). Manage its tokens and license with `sudo`, for example `sudo tangle-core token create ...`, so they land in the same folder.

File locations

	MACOS	WINDOWS	LINUX
Data (modules, tokens, license, dashboards, Sentinel history)	`~/Library/Application Support/Tangle`	`%ProgramData%\Tangle`	`~/.local/share/tangle`
Logs, `audit.log`	`~/Library/Logs/Tangle`	`%ProgramData%\Tangle\logs`	`~/.local/state/tangle/logs`
Data, when run as root (boot service)	`/Library/Application Support/Tangle`		`/var/lib/tangle`
Logs, when run as root (boot service)	`/Library/Logs/Tangle`		`/var/log/tangle`

`TANGLE_HOME` moves everything into one folder: data in `$TANGLE_HOME`, logs in `$TANGLE_HOME/logs`. This is useful for portable installs.

Inside the data folder:

PATH	CONTENTS
`modules/`	Installed device modules.
`module-data/`	Writable data of modules.
`tokens.json`	API tokens (hashes only).
`license/`	The license file of this machine.
`dashboards/<project id>/`	Dashboard drafts and every published version.
`sentinel/<project id>/history.jsonl`	Sentinel alert history.
`cluster.key`	Redundancy key of an installed service.
`cluster.json`	Cluster membership and key saved at pairing, readable by the Tangle account only. Back it up with the project: it holds the cluster key. Remove it with `tangle-core cluster leave`.
`secrets.json`	Device and channel secrets, only when the OS credential store is not available (see below).

In a cluster, the primary copies device passwords to every backup over the encrypted cluster channel, so each backup's credential store holds them too. Device passwords and keys, and notification channel secrets, go to the OS credential store: macOS Keychain, Windows Credential Manager, or Secret Service on Linux. They never go in the project file. When no credential store is available (for example on a headless Linux server), core logs a warning and keeps them in `secrets.json`, readable only by its owner. Set `TANGLE_SECRETS=file` to always use the file.

Logs

Core logs to the console (standard error) and to a daily log file in the log folder. Files are JSON lines named after `tangle-core`, and the last 14 days are kept.

Set the level with `TANGLE_LOG`:

```sh
TANGLE_LOG=debug tangle-core run --project show.yaml
```

The default is `info`. `TANGLE_LOG` accepts the usual filter syntax, for example `TANGLE_LOG=info,tangle_cluster=debug` for more detail from redundancy only. `--log-json` makes the console output JSON too.

Audit log

`audit.log` in the log folder records every configuration change and control action as one JSON line: who did it, through which surface, what, on what, and the outcome.

```JSON
{"at":1760362800000,"surface":"rest","caller":"stage-
ipad","action":"timelines.control","target":"timeline show play","outcome":"ok"}
```

`surface` is `rest`, `mcp`, `ws` or `cli`. `caller` is the token name. `outcome` is `ok` or an error code such as `missing_scope` or `not_primary`.

API tokens

REST, MCP, and WebSocket clients on other machines need an API token.

```SH
tangle-core token create --name stage-ipad --scopes read,control   # prints the secret once
tangle-core token list
tangle-core token revoke stage-ipad
```

The secret (starting with `tgl_`) is printed once and never stored: only its hash goes in `tokens.json`. Copy it when you create the token. Without `--scopes`, a token gets `read,control`.

SCOPE	ALLOWS
`read`	Read the project, state and status.
`control`	Send intents, drive timelines, run automations, acknowledge and silence alerts, push metrics.
`configure`	Change the project. The designer needs it to edit.
`admin`	Tokens, licensing and node settings.
`live`	Lets an MCP assistant change things while a timeline plays. Without it, MCP writes and control are refused with `409 show_running` during a show.

Give each client its own token with the fewest scopes it needs, so you can revoke one without touching the others. A revoked token stops working at once, without a restart; WebSocket sessions using it are disconnected within 2 seconds.

Clients on the same machine can use the WebSocket without a token, with `read` and `control`.

Tokens belong to one machine. With redundancy, create each client's token on every node.

Remote access

By default core listens on `127.0.0.1` and only the local machine can connect. To accept other machines:

```sh
tangle-core run --project show.yaml --listen 0.0.0.0:7340 --allow-remote --insecure-http
```

Core refuses to listen on a network address without `--allow-remote`, and, because there is no TLS yet, without `--insecure-http`.

> **Warning:** Traffic, including API tokens, is not encrypted. Use remote access on trusted networks only: a dedicated show or control network, or a VPN. Do not expose port 7340 to the internet.

- REST and MCP calls from other machines need a token. WebSocket clients from other machines must send a token in their `hello`.
- In the free designer mode, remote access is always off: core serves this computer only, whatever the options.
- On Lite, the built-in designer and dashboards work remotely, but third-party REST and MCP calls are refused. See Licensing.

Ports and firewall

PORT	PROTOCOL	DIRECTION	USED FOR
7340	TCP	inbound	Designer, dashboards, REST, WebSocket and MCP. Change with `--listen`.
7341	UDP and TCP	between cluster nodes	Redundancy heartbeats (UDP) and mirroring (TCP). Change with `--cluster-listen`.
7342	UDP and TCP	local network	Core discovery beacons (UDP multicast to 239.255.73.42) and pairing (TCP, only while a core is not in a cluster). Off with `--no-discovery`.

Core also makes outbound connections:

- To devices, on the ports their modules use (see each module in the Device catalog).
- To the license server over HTTPS, to activate and refresh the license. Core runs without it; see Offline activation.

- For Sentinel: SMTP (587, 465 or 25), HTTPS to Twilio, ntfy, Pushover, Slack, Teams and your webhooks, and the hosts your probes check.

Open 7340 only to the networks your designer and dashboard clients use, 7341 only between cluster nodes, and 7342 only on the local network where you pair cores.

Back up a project

Core writes every change back to the project file, in the same format (JSON or YAML) as it was loaded. The file is replaced in one step, so a crash never leaves it half written. Core writes the file from its own copy of the project, so formatting and YAML comments are not kept.

Back up:

- The project file.
- `tangle.lock` next to the project file, if you use one. It pins module versions; at start core activates the locked versions and warns about any that are missing.
- `dashboards/<project id>/` in the data folder: drafts and published dashboard versions.
- Optionally `sentinel/<project id>/` for the alert history.

Device passwords and channel secrets are not in the project file. Keep them in your own password manager so you can enter them again on a new machine.

Do not copy `license/` or `tokens.json` to another machine: the license file only works on the machine it was issued for, and tokens are best created again per machine.

> **Note:** If core cannot write the project file (disk full, permissions), it logs an error and the change stays live in memory. Check the log after changes made during a show.

To restore, put the project file back and start core with it. You can also export the project as JSON from the designer's **Settings** section, or read it with `GET /api/v1/project`.

Upgrades

Release archives contain the `tangle-core` program and a `ui` folder. Keep `ui` next to the program.

1. Stop the show, or plan a moment when core can restart.
2. Replace `tangle-core` and the `ui` folder with the new version, in the same place. The service definition points to the program's path, so it picks up the new version.
3. Restart the service (see Run as a service), or start core again.
4. Check the log, the designer and `tangle-core license status`.

Device modules are installed and updated separately with the `tangle` CLI. Modules used by a running project are not removed or replaced under it. See CLI.

A perpetual license includes updates released until the `updates until` date shown by `tangle-core license status`.

With redundancy, keep every node on the same version. See Redundancy.

GUIDES

End-user views

Give the people using a site one place for their dashboards and maps, at /view, with its own menu and access. Embed maps, web pages and cameras in dashboards.

The designer is for the people who build a project. An end-user view is for the people who use it: a front desk, a facility manager, a teacher. It shows a menu of dashboards and maps and opens them full screen, on a tablet, a wall panel or a phone.

Open a view

```HTTP
http://<core address>:7340/view
http://<core address>:7340/view/<view id>
```

`/view` opens the project's only view, or offers a choice when there are several. The address keeps what is shown (`#d/<dashboard>/<page>` or `#m/<map>`), so a bookmark or a reload comes back to the same screen.

Until a project has a view of its own, `/view` shows a built-in view called **Everything**: every dashboard, then every map, open to anyone on the network.

> **Note:** As for dashboards, other machines reach a view only when core accepts them and has a license (or runs with `--demo`). See Dashboards.

Make a view

In the designer, open **End-user views**.

1. Click **New view**, or **Customize** next to Everything to start from every dashboard and map.
2. Name it. The id is its address (`/view/front-desk`).
3. Choose **Who can open it**: anyone on the network, anyone with the PIN (4 to 12 digits), or enrolled panels only.
4. Choose the skin and theme, and what it **Opens on** (by default, the first item).
5. Under **Menu**, add the dashboards and maps people need, put them in order, and rename them if the menu should say something else.
6. Click **Save**, then **Open** to try it.

A view's access is its own: a PIN view opens its dashboards and maps without asking for their own PINs. Deleting a dashboard or a map removes it from every view.

Views are stored in the project (`views`), and can also be written through the REST API (`/api/v1/views`) and MCP like any other part of the project.

On phones and panels

On a phone the menu folds into a ☰ button. With `?kiosk=1` the view stays awake and dims like a kiosk dashboard. An enrolled panel opens a `panels` view with `?panel=<token>`, as described in Enrol a wall panel.

When the connection drops, the view says so and reconnects on its own.

Embedded content

Five widgets bring other things into a dashboard. Add them from the builder's palette like any other widget.

Map

Shows one of the project's maps, with each placed device and its status, and people or assets reported by tracking sources. Drag to pan, pinch or scroll to zoom; zoom and fit buttons are in the corner. Tap a device to see its name, status and room.

SETTING	MEANING
Map	The map to show
Names	Device names next to their markers (on by default)
Live positions	People and assets from tracking sources (on by default)

Web page

Shows another site in a frame. The address must start with `http://`, `https://` or `/`.

SETTING	MEANING
Address	The page to show
Reload every	Seconds between reloads (5 or more), empty for never
Extra permissions	More sandbox flags, such as `allow-downloads allow-modals`

The page runs sandboxed: it can run scripts and forms but cannot take over the dashboard. Videos can autoplay (muted) and go full screen.

YouTube and Vimeo share links (`youtube.com/watch?v=…`, `youtu.be/…`, `vimeo.com/…`) are turned into their player address. To autoplay a YouTube video, add `?autoplay=1&mute=1` to an embed address such as `https://www.youtube.com/embed/<id>?autoplay=1&mute=1&rel=0`.

Many sites refuse to be shown inside another page (`X-Frame-Options` or `frame-ancestors`), and no setting can change that. For well-known ones such as Google search, Gmail, Facebook, Instagram, X, LinkedIn, GitHub and Amazon, the widget says **This site refuses to be embedded**; for others the frame stays empty or shows the browser's error. Use the site's own embed link when it has one (Google Maps **Share > Embed a map**, a published Google Doc or Calendar), or a link widget that opens the site.

Camera

Shows a camera's video, and with a camera device, a pan, tilt and zoom pad made of the module's own controls.

SETTING	MEANING
Camera device	The device whose PTZ controls and addresses to use
Stream address	The video address. Empty: the camera device's `streamUrl`, else its `snapshotUrl`
Format	From the address, MJPEG, JPEG snapshots, HLS, WebRTC (WHEP), or the camera's web page
Snapshot every	Seconds between snapshots (default 2)
PTZ pad	Show the pan, tilt and zoom pad (on by default)

FORMAT	WORKS IN	TYPICAL ADDRESS
MJPEG	Every browser	`http://<camera>/cgi-bin/mjpeg`
Snapshot	Every browser	`http://<camera>/snapshot.jpg`
HLS	Safari, recent Chrome and Edge	`http://<gateway>/<camera>/index.m3u8`
WebRTC (WHEP)	Every browser	`http://<gateway>:8889/<camera>/whep`

Browsers cannot play RTSP. For an RTSP-only camera, run a gateway such as go2rtc or MediaMTX and use its WebRTC or HLS address. A stream that fails is retried every 10 seconds.

The VISCA and Panasonic AW camera modules have `streamUrl` and `snapshotUrl` settings, so every dashboard showing that camera uses the same address.

Navigation bar

A row of buttons, or a column with **Direction: Vertical**. Each button has a label and a destination; the one for the current page or map is lit.

Link

Text, or a button with **Style: Button**, that opens a destination, optionally in a new tab.

Destinations

Navigation bars and links accept:

DESTINATION	OPENS
A page of this dashboard	That page
Another dashboard (and a page)	That dashboard; inside a view, without leaving it
A map	The map, inside a view; outside one, at `/view/#m/<map>`
An address (`https://...` or `/path`)	That address

In the project file they are strings: `page:<id>`, `dashboard:<id>` or `dashboard:<id>/<page>`, `map:<id>`, or the address.

GUIDES

Listeners

Let other systems trigger Tangle with their own messages - WebSocket, HTTP, TCP, UDP, OSC or serial - matched by rules that set variables, run steps and automations, and answer the sender.

A listener opens a port (or a serial line) for other systems: a touch panel, a show controller, a Stream Deck through Companion, QLab, TouchDesigner, a building management system, a webhook. They send their own strings, JSON or OSC, in their own shape. Each message is checked against the listener's **rules**: a matching rule sets variables, runs steps and automations, starts timelines and answers the sender.

The REST API and the WebSocket API do the same with Tangle's own messages and a token. Use a listener when the other system cannot be changed to speak them.

Listeners are stored in the project (`project.listeners`, contract 0.2.25 and later) and opened by the primary node, or by the node named in `node`.

Make a listener

1. Click **Listeners** in the project tree, then **New listener**.
2. Give it a **Name**. **Module** stays on **Built-in listeners** unless you installed another listener module.
3. Under **Where it listens**, choose the **Transport** and fill in its settings:

TRANSPORT	SETTINGS	A MESSAGE IS
`websocket`	**Port**, **WebSocket path** (default `/`), **Listen on address**	One text frame. The reply goes back on the same connection.
`http`	**Port**, **Listen on address**	One request. The reply is the HTTP response.
`tcp`	**Port**, **Listen on address**, **Message format**, **End of message**	One line (or up to the end-of-message characters you set, such as `\r\n` or `\x03`).
`udp`	**Port**, **Listen on address**, **Message format**	One datagram. With format `osc`, one OSC message (bundles are split).
`serial`	**Serial port**, **Baud rate**, **Data bits**, **Parity**, **Stop bits**, **Message format**, **End of message**	One line. The serial port can be a network serial port (`tcp://gateway:4001`), see Devices.

Message format is `text`, `json`, `hex` (raw bytes shown as hex) or `osc`. **Listen on address** `0.0.0.0` listens on every network card, `127.0.0.1` on this computer only. Ports under 1024 need Tangle to run with more rights.

4. Under **Who may send**:
   - **Allowed hosts**: addresses or ranges (one per line, `192.168.1.0/24`). Empty: this computer and private networks only. `*` allows everyone.
   - **Token**: a secret senders must give, as `Authorization: Bearer <token>` or `?token=<token>` over HTTP and WebSocket, or as the first line over TCP and serial. It is kept in the credential store and never shown again.
   - **Messages per second** (default 50), **Burst** (100) and **Largest message** (65536 bytes). Extra messages are dropped and counted, never queued.
5. Add **rules**, then click **Create listener**.

Open closes the listener without deleting it when unticked.

Rules

Rules are checked in order and the first that matches wins, unless it has **Keep checking the next rules**. Each rule has a **Match**:

MATCH	MATCHES	CAPTURES
Any message	Every message	
Exact text	The whole message (a trailing line end is ignored), optionally ignoring case	
Starts with	Messages that start with the text	`{{rest}}`
Regular expression	A regular expression (flags `i`, `m`, `s`, `x`)	Named groups as `{{name}}`, every group as `{{_1}}`, `{{_2}}`...
JSON value	The message parsed as JSON, optionally at a path (`$.cue.number`)	`{{value}}`, and named paths
OSC address	An OSC address pattern (`/cue/*/go`, `?`, `[a-z]`, `{a,b}`)	Wildcards as `{{_1}}`..., arguments as `{{args}}`, `{{arg1}}`...
HTTP route	A method and a path (`/cue/:number/*`)	`:name` as `{{name}}`, `*` as `{{rest}}`, plus `{{query}}`, `{{headers}}` and `{{body}}`

Every run also gets `{{listener}}`, `{{rule}}`, `{{message}}`, `{{json}}` (the message as JSON, when it parses) and `{{from}}` (the sender's address). Captures that look like numbers become numbers, unless the rule has **Keep captures as text**.

What a matching rule does:

- **Set variables**: each variable gets a template (`{{number}}`) or an expression. A project variable is set for everyone (and its variable triggers fire); any other name is a run variable for the rule's steps.
- **Steps**: any automation step: an intent, a device action, a timeline, **Run an automation**, an if, and so on, with the captures as `{{name}}`.
- **Answer**: a reply template. `"{{name}}"` alone keeps the capture's type; `"OK {{cue}}"` fills it in as text; JSON objects are filled inside. Over HTTP you can also set the status and headers. **Answer when the steps are done** waits for them (at most 10 s) before replying.

Automations can also start on a rule with the trigger **A listener message matches** (Automations). That keeps the work in one automation that a schedule or a button can run too.

When no rule matches sets the reply for other messages (HTTP answers 404 by default) and whether they are logged.

Example: start the gallery from another system

This HTTP listener lets any system on the network start and stop the room built in the tutorial with a plain HTTP request:

```JSON
{
  "id": "gallery-http",
  "name": "Gallery HTTP",
  "module": "tangle.builtin.listener",
  "config": { "transport": "http", "port": 9100 },
  "rules": [
    {
      "id": "start", "name": "Start",
      "match": { "kind": "http", "method": "POST", "path": "/gallery/start" },
      "steps": [{ "kind": "automation", "automationId": "start-installation", "wait": false
}],
      "reply": "OK"
    },
    {
      "id": "stop", "name": "Stop",
      "match": { "kind": "http", "method": "POST", "path": "/gallery/stop" },
      "steps": [{ "kind": "automation", "automationId": "stop-installation", "wait": false }],
      "reply": "OK"
    }
  ]
}
```

```SH
curl -X POST http://<core address>:9100/gallery/start     # OK
```

Other examples:

- QLab or a lighting console sends OSC `/tangle/cue/12` over UDP: an **OSC address** rule `/tangle/cue/*` runs a timeline step `go` with `cue` `{{_1}}`.
- A control system sends `SCENE 3\r` over TCP: a **Regular expression** rule `^SCENE (?<n>\d+)$` sets the project variable `scene` to `{{n}}`, and an automation that starts when `scene` changes does the rest.
- A webhook posts JSON: a **JSON value** rule at `$.event` with the value `door.open` runs the steps.

Messages and tests

The listener page shows its **Status** (opening, listening, error, closed) and the **Messages** log, live: each message with its sender, the rule that matched, the captures, the runs it started and the reply. Filter it by matched, unmatched, dropped or test messages.

Try it sends a message to the saved rules without a real sender. By default it is a dry run that only shows what would match; tick **Also set variables and run the steps** to run them as a real message would.

API

- REST: `GET /api/v1/listeners` (with status), `POST /api/v1/listeners`, `GET`, `PUT` and `DELETE /api/v1/listeners/{id}`, `GET /api/v1/listeners/{id}/log`, `GET /api/v1/listeners/{id}/runs`, `POST /api/v1/listeners/{id}/test` (`{"text": "SCENE 3", "run": false}`; `bytes`, `osc` or `http` for other kinds of message).
- Events: `listener.message:<id>` for every message received.
- MCP: `get_listeners`, `get_listener_runs`, `test_listener`, and `create_listener` / `update_listener`.

Listener modules

The built-in listener (`tangle.builtin.listener`) covers WebSocket, HTTP, TCP, UDP (text, JSON, hex, OSC) and serial. Other modules can open listeners too: a module of category `listener` declares its transports and settings in a `listener` block of its manifest, and only owns the socket and the framing. Core does the matching, the security checks, the rate limits, the log and the tester. See Module SDK.

GUIDES

Device creator

Teach Tangle a device it has no driver for - connection, commands, feedback and controls in the designer, optional code, then save and install. Share the result on the community repository.

The device creator writes a driver for a device Tangle does not know yet, without leaving the designer: say how to reach the device, what to send, how to read its answers and which controls it gets, and add code only where settings are not enough. **Save and install** turns it into a module on the core, ready for scenes, timelines, automations and dashboards like any bundled driver.

Open it with **Device creator** at the bottom of the sidebar.

Start a driver

Pick the connection the device speaks:

TILE	FOR
TCP	Telnet-style text or binary commands on a port
UDP	Datagrams: show controllers, media players
Serial (RS-232/422/485)	A local serial port or one reached through an IP gateway
HTTP / HTTPS	REST APIs, JSON or text
WebSocket	Live JSON or text messages, `ws` or `wss`
MIDI	Notes, controllers, program changes, SysEx
Modbus (TCP or RTU)	Registers and coils, over TCP or RS-485
OSC (Open Sound Control)	Addresses with typed arguments: consoles, show controllers, media servers

Import a driver.json opens a driver made elsewhere. **Your drivers** lists the drivers saved on this core.

The tabs

The editor checks the driver as you type: the number next to a tab and the **Checks** list say what is left to fix. Unsaved work is kept in the browser.

Device

Manufacturer, **Product or protocol** and **Category** make the driver's id, `<manufacturer>.<product>.<category>` (use `misc` as the manufacturer for a protocol). Changing them later makes a new driver. Add the **Name shown**, **Version**, **Author**, **Description** and the **Models it works with**.

Connection

The connection type and its defaults: the default port (each device sets its own address and can change the port), whether messages are **Text** or **Bytes (hex)**, what ends each reply, whether the device asks for a password, the answer timeout and a pause between sends.

Device settings adds settings each device fills in (a channel, a zone), used in commands as `{key}`.

One driver can reach the same device in several ways, like LAN and RS-232: **+ Add a connection** adds another one. Commands, controls and code are shared; a command can send something else on one connection, and a connection can have its own feedback rules. Each device picks its connection in its settings.

Commands

Each command is a message template, for example `PWR ON\r`. Templates take escapes (`\r`, `\n`, `\x02`) and `{name}` placeholders with filters: `{level|scale:0:100}`, `{n|hex:2}`, `{x|pad:3}`, `{src|map:hdmi1=1,hdmi2=2}`. A command can wait for an answer, and the preview shows what goes on the wire.

- HTTP commands have a method, a path, headers and a body.
- OSC commands have an address (`/ch/{ch|pad:2}/mix/fader`) and typed arguments (int, float, string, blob, bool…); several messages can go in one bundle.
- Modbus commands write a register, read one, or run a function code.
- **Runs code** runs your own function from the Code tab instead of a template.

On its own picks the commands sent when the connection opens and the ones sent regularly (a status query), with the interval. Regular commands also tell Tangle the device is there.

Feedback

Rules that read what the device sends. A rule matches each message with a regular expression (an OSC address for OSC drivers); its first group is the value. A **map** turns raw values into others (`0=false, 1=true`), and the rule sets a value of this driver, a standard state (Power, Mute, Level 0..1 with the device's range, Input, Preset) or raises an event. **Try a message** shows what the rules make of a message you paste.

Values of this driver are shown on the device page and can be used in controls, automations and dashboards.

Modbus drivers have a **Registers** tab instead: table, address, data type, byte order, scale, offset, unit and map of each register, read every poll interval.

Controls

Standard actions maps commands to Tangle's standard actions (power on and off, mute, select input, set level, recall preset), so scenes, intents, automations and dashboards use the device like any other. Tick **Assume the action worked** for devices that do not report their state.

Controls adds controls for the device page and dashboards; the standard actions get theirs already.

Code

Optional JavaScript (`script.js`) that runs in the driver's own process on the core, with Node.js. **Start from the template** gives:

```JS
export default {
  async connect(api) {},            // after the connection opens
  message(text, api, json) {},      // every message
  async poll(api) {},               // every poll interval
  actions: {                        // commands marked "Runs code"
    async login(params, api) {
      await api.request(`LOGIN ${api.config.password}\r`, { match: /^OK/ });
      api.set("loggedIn", true);
    },
  },
};
```

`api` can `send`, `request` (send and wait for a matching answer), `run` another command, call `http`, `midi`, `modbus` and `osc` helpers, `set` values, `emit` events, `log`, `sleep`, and read `config` and `variant`.

Try it

After saving, add a device that uses the driver and try it here: run each command, watch the traffic in both directions and the values the feedback rules set.

Files

Download the **Module folder (.zip)**, a module that needs nothing else: unzip it into another core's modules folder. **driver.json** alone is the definition, to edit or share.

Save and install

Save and install builds the module, installs it on this core, reloads its process and restarts the devices that use it. The driver then appears in the **Driver library** of every device page.

> **Tip:** Build and test a driver against the simulator before the device is on site: its `ascii` type answers line-based commands from rules, `osc` stores and answers OSC messages, and `moxa.nport.gateway` puts a device behind an IP-to-serial gateway. See Simulator.

Share it on the community repository

The community repository (community.tangle.systems) is where people share the drivers they make.

1. Sign in under **Settings > Community** with the email and password of your Tangle license account. The first time, choose a username: it becomes your namespace, so your drivers are named `<username>/<driver id>`.
2. Open your saved driver in the device creator and click **Publish to community**. Each published version is final: publish a new version to change it.

To install someone else's driver, open the **Driver library** on a device page and tick **Also search all community drivers**. **Use** installs it and picks it for the device. **My community drivers** lists your own. A driver that does not behave can be reported from the library.

API

- REST: `/api/v1/drivers` (list, get, files, check, save, delete) and `/api/v1/community` (status, sign-in, register, sign-out, search, install, publish).
- MCP: `list_drivers`, `get_driver`, `get_driver_files`, `check_driver`, `save_driver`, `delete_driver` and `community`.
- CLI: `tangle-core community status` and `tangle-core community search [words] [--mine]`.
- Core finds the builder with `--device-creator <dir>` (default: `device-creator/` next to the core program) and keeps saved drivers in its data folder under `drivers/`.

To write a module by hand instead, with full control of the protocol, see Module SDK.

REFERENCE

REST API

Configure and control a Tangle project over HTTP and JSON. Tokens, scopes, conventions, errors and every route of /api/v1.

tangle-core serves a REST API at `http://<host>:7340/api/v1`. Every action the designer, dashboards and assistants can take is available here, with the same validation, scope checks and audit log. Request and response bodies use the Tangle contract types (the same JSON as the project file).

Base URL and documentation

WHAT	WHERE
API base	`http://<host>:7340/api/v1`
OpenAPI 3.1 document	`GET /api/v1/openapi.json` (no token needed)
Interactive documentation	`GET /api/docs` (no token needed)

The OpenAPI document is generated from the same route table that serves the API, so it always matches the running core. Its component schemas are the contract types. The `/api/docs` page loads its viewer from a public CDN, so the browser that opens it needs internet access.

By default tangle-core listens on `127.0.0.1:7340` (this machine only). To accept other machines, start it with `--listen 0.0.0.0:7340 --allow-remote --insecure-http` (see CLI). There is no TLS yet: use remote access on trusted networks only.

Authentication

Send an API token in the `Authorization` header:

```http
GET /api/v1/project HTTP/1.1
Host: 127.0.0.1:7340
Authorization: Bearer tgl_0f3c...
```

Create tokens on the machine that runs tangle-core:

```sh
tangle-core token create --name stage-script --scopes read,control
tangle-core token list
tangle-core token revoke stage-script
```

- The secret (`tgl_` followed by 64 hex characters) is printed once. Only its SHA-256 hash is stored, in `tokens.json` in the Tangle data folder.
- Without `--scopes`, a token gets `read,control`.
- A revoked token stops working at once, without restarting tangle-core.
- For clients that cannot set headers (a browser `EventSource`), pass the token as `?token=<token>` in the query string.
- A missing, unknown or revoked token gets `401` with a `WWW-Authenticate: Bearer` header.

For a core installed as a boot service, manage its tokens with `sudo tangle-core token ...` (macOS, Linux). See CLI.

> **Note:** REST and MCP access for third-party software is part of Tangle Pro and Ultimate (and the demo mode). On other editions these requests get `403 not_licensed`. The built-in designer and dashboards always work. See Licensing.

Scopes

SCOPE	ALLOWS
`read`	Read the project, live state, timeline states, alerts, metrics, license, events.
`control`	Send intents, drive timelines, run automations, acknowledge and silence alerts, push metrics, resume a paused session.
`configure`	Change the project (create, replace, delete resources), test devices, scan the network, edit and publish dashboards, test notification channels.
`admin`	Administration: licensing (activate, sign in, refresh, deactivate, offline), pairing and accepting cores.
`live`	Lets an MCP assistant change things while a timeline plays. It has no effect on REST. See MCP.

Scopes do not include each other: a token that should read and configure needs `read,configure`. A token without the scope a route needs gets `403 missing_scope`.

Dashboard sessions

Browsers that open a dashboard get a short-lived session token from `POST /api/v1/dashboards/{id}/session` (no token needed to call it). Session tokens start with `tgd_`, last 12 hours and carry the `read` and `control` scopes. See Dashboards.

Conventions

JSON and resource ids

- Send `content-type: application/json`. A body that is not JSON gets `400 invalid_json`.
- Every resource has an `id`: lowercase letters, digits, `.`, `_` and `-`, starting with a letter or digit, at most 128 characters.
- Times are milliseconds since the Unix epoch (UTC), unless a field says otherwise.
- Cross-origin requests are allowed from any origin (requests carry bearer tokens, never cookies). The `ETag` header is exposed to browsers.

Errors

Errors use RFC 9457 problem details, with content type `application/problem+json` and a stable machine-readable `code`:

```json
{
  "type": "https://tangle.dev/problems/device_not_found",
  "title": "Not found",
  "status": 404,
  "detail": "device proj-9 does not exist",
  "code": "device_not_found"
}
```

Validation errors (`422`) add an `errors` array with one entry per problem. `not_primary` adds `primary`.

STATUS	CODE	MEANING
400	`invalid_json`	The body is not JSON.
400	`invalid_wait`	`?wait=` is not like `5s` or `500ms`.
400	`topics_required`	`GET /events` without `?topics=`.
400	`unsupported`	The timeline does not support this transport operation.
401	`unauthorized`	Missing, unknown or revoked token, or expired dashboard session.
401	`pin_required`, `wrong_pin`	Dashboard session: PIN missing or wrong.
403	`missing_scope`	The token lacks the scope this route needs.
403	`not_licensed`	The edition does not include this (external API, monitoring, or the change exceeds a license limit).
403	`panels_only`	The dashboard accepts enrolled panels only.

STATUS	CODE	MEANING
404	`<kind>_not_found`	For example `device_not_found`, `timeline_not_found`, `intent_not_found`, `automation_not_found`.
404	`no_draft`	Publishing a dashboard that has no draft.
409	`already_exists`	`POST` with an id that is already used.
409	`timeline_running`	Editing a timeline (or its tracks and cues) while it plays, without `allowLive`.
409	`show_mode`	Network scan while a timeline plays.
409	`show_running`	MCP only: a change while a timeline plays, without the `live` scope.
409	`not_primary`	This node is a redundancy backup. `primary` holds the primary's address.
409	`disabled`	Running a disabled automation.
409	`not_paused`	Resuming when no designer or demo session is paused.
409	`busy`, `conflict`	The project changed too fast; try again.
412	`precondition_failed`	`If-Match` does not match the current ETag.
422	`invalid`	The body, or the project as it would be after the change, is invalid.
429	`too_many_attempts`	More than 5 wrong dashboard PINs in a minute.
503	`unavailable`	A service is not ready (stopped player, storage not configured, credential store error).

ETags and If-Match

`GET` on a single resource (and on `/project`) returns a strong `ETag`. Send it back in `If-Match` on `PUT` or `DELETE` to make sure nobody changed the resource since you read it. If it changed, the request fails with `412 precondition_failed` and nothing is applied.

- `If-Match: *` matches any existing version.
- A `PUT` with an `If-Match` (other than `*`) on a resource that does not exist fails with `412`.
- Successful writes return the new `ETag`.

Dry runs

Add `?dryRun=true` (or `?dryRun`, `?dryRun=1`) to any create, replace or delete. Core validates the change against the whole project and returns what would happen, without applying it:

```json
{
  "dryRun": true,
  "result": { "id": "pj-2", "name": "Projector 2", "moduleId": "misc.pjlink.projector",
"room": "hall" },
  "changes": [{ "op": "create", "kind": "device", "id": "pj-2" }]
}
```

`result` is the resource as it would be (absent for a delete). `changes` lists contract `ProjectChange` entries (`op`: `create`, `update` or `delete`; `kind`; `id`; `parent` for tracks and cues, such as `show` or `show/lights`).

How changes are applied

- Each write is atomic. Core applies the change to a copy of the project, validates the whole result (schema and cross-references, such as a scene pointing at a device that no longer exists), then swaps it in. A mistake changes nothing and the `422` lists each problem.
- Changes are saved to the project file (JSON or YAML, following its extension) and recorded in `audit.log` in the log folder, with the token name and surface.
- Devices are started or stopped to match.
- Device settings are checked against the module's `configSchema`. Secret settings (marked `x-secret` in the module) are removed from the body and stored in the OS credential store, never in the project file.
- Notification channel secrets (passwords, tokens, webhook URLs) also go to the credential store. The project keeps `********`; send `********` back to keep the stored value.
- A dashboard PIN sent as `access.pin` is stored as a salted hash and never returned.
- Every project change publishes a `project.changed` event.

Live timelines

Editing a timeline, or one of its tracks or cues, while it plays is refused with `409 timeline_running`. Add `?allowLive=true` to edit it anyway. Network discovery is always refused while any timeline plays (`409 show_mode`), because a scan floods the show network.

MCP assistants have an extra protection: without the `live` scope, every change and control action is refused while a timeline plays. REST clients are not subject to it. See MCP.

Redundancy: 409 not_primary

In a redundant cluster, only the primary accepts writes and control. A backup answers changes, intents, transport, automation runs, and alert acknowledgement or silencing with `409`:

```json
{
  "type": "https://tangle.dev/problems/not_primary",
  "title": "Conflict",
  "status": 409,
  "detail": "this node is a backup; send changes to the primary",
  "code": "not_primary",
  "primary": "http://10.0.0.1:7340"
}
```

`primary` is `null` while the backup does not know the primary yet. Reads work on every node. See Redundancy.

Routes

Scope is the scope the token needs. Paths are relative to `http://<host>:7340`.

Project and modules

METHOD	PATH	SCOPE	DESCRIPTION
GET	`/api/v1/project`	read	The whole project (contract `Project`), with an `ETag`. Dashboard PIN hashes are left out.
GET	`/api/v1/modules`	read	Installed modules: `[{ "manifest": ModuleManifest, "devices": ["pj-1", ...] }]`.
GET	`/api/v1/modules/catalog`	read	Installed modules and the modules available to install (folders scanned on each call).
POST	`/api/v1/modules/catalog/{id}/install`	configure	Install or update a module from the catalog; devices using it restart.
POST	`/api/v1/project/save`	configure	Write the open project's file now (changes are also saved as they are made). Returns `{ savedAt, path }`.
PUT	`/api/v1/project/variables`	configure	Declare the project's variables (a list of `ProjectVariable`).
PUT	`/api/v1/project/units`	configure	The project's units: metric or imperial, and overrides per quantity (`UnitPreference`).
PUT	`/api/v1/project/controls`	configure	The default dashboard control style and the highlight colours of each style.

METHOD	PATH	SCOPE	DESCRIPTION
PUT	`/api/v1/project/cluster`	configure	How the cluster shares the work: standby or shared mode, folder placement, auto return, node health limits (`ClusterSettings`).
GET	`/api/v1/units`	read	Every quantity and unit (°C, °F, W, kW, h…) with how to convert between them.
GET	`/api/v1/templates`	read	Project templates: the kinds of project the wizard can start from.
POST	`/api/v1/templates/{id}/apply`	configure	Add a template's rooms, planned devices, timelines, automations, dashboards and alert rules to the project (body: answers, devices; `?dryRun`).

Project library

METHOD	PATH	SCOPE	DESCRIPTION
GET	`/api/v1/projects`	read	The library (`ProjectLibrary`): the open project (`loaded: false` when none) and every project on this core.
POST	`/api/v1/projects`	configure	Create an empty project (`ProjectCreateRequest`: `name`, optional `id`, `open`).
POST	`/api/v1/projects/import`	configure	Import a `.tangle` file or a project `.json` / `.yaml` as a new project: the file as the body, `?name=`, `?open=true`.
POST	`/api/v1/projects/{id}/open`	configure	Open a project; the open one closes. Refused with `409 show_running` while a timeline plays, unless `{ "force": true }`. Answers `202`; clients get `project.switching`, then `project.changed`.
POST	`/api/v1/projects/{id}/duplicate`	configure	Copy a project with its dashboards and plans under a new name; `open: true` is Save as.
GET	`/api/v1/projects/{id}/export`	read	One `.tangle` file (a zip: project, dashboards, plans, module lock; no device passwords).
DELETE	`/api/v1/projects/{id}`	configure	Delete a project, its dashboards, plans and alert history (not the open one).
POST	`/api/v1/project/close`	configure	Close the open project: devices stop and changes are refused until another opens. Same `force` rule as open.

Project resources

Each resource type supports the same five routes:

METHOD	PATH	SCOPE	DESCRIPTION	SUCCESS
GET	collection path	read	List all resources of this type.	`200` with an array
POST	collection path	configure	Create one. The body must include `id`.	`201` with the resource, `Location` and `ETag`
GET	item path	read	Read one.	`200` with the resource and `ETag`
PUT	item path	configure	Create or replace one with a complete new version. The body `id` may be left out; if given it must match the path.	`200` (replaced) or `201` (created), with `ETag`
DELETE	item path	configure	Delete one.	`204`

Create, replace and delete accept `?dryRun` and `?allowLive`; replace and delete accept `If-Match`.

RESOURCE	COLLECTION PATH	ITEM PATH	BODY (CONTRACT TYPE)
Rooms	`/api/v1/rooms`	`/api/v1/rooms/{id}`	`Room`
Devices	`/api/v1/devices`	`/api/v1/devices/{id}`	`DeviceInstance`
Scenes	`/api/v1/scenes`	`/api/v1/scenes/{id}`	`Scene`
Timelines	`/api/v1/timelines`	`/api/v1/timelines/{timeline}`	`Timeline`
Tracks	`/api/v1/timelines/{timeline}/tracks`	`/api/v1/timelines/{timeline}/tracks/{track}`	`Track`
Cues	`/api/v1/timelines/{timeline}/tracks/{track}/cues`	`/api/v1/timelines/{timeline}/tracks/{track}/cues/{id}`	`Cue`
Dashboards	`/api/v1/dashboards`	`/api/v1/dashboards/{id}`	`Dashboard`
Automations	`/api/v1/automations`	`/api/v1/automations/{id}`	`Automation`
Alert rules	`/api/v1/alertRules`	`/api/v1/alertRules/{id}`	`AlertRule`

RESOURCE	COLLECTION PATH	ITEM PATH	BODY (CONTRACT TYPE)
Notification channels	`/api/v1/notificationChannels`	`/api/v1/notificationChannels/{id}`	`NotificationChannel`
Maintenance windows	`/api/v1/maintenanceWindows`	`/api/v1/maintenanceWindows/{id}`	`MaintenanceWindow`
Probes	`/api/v1/probes`	`/api/v1/probes/{id}`	`Probe`
Listeners	`/api/v1/listeners`	`/api/v1/listeners/{id}`	`Listener`
Maps	`/api/v1/maps`	`/api/v1/maps/{id}`	`Map`
End-user views	`/api/v1/views`	`/api/v1/views/{id}`	`EndUserView`
Lighting sequences	`/api/v1/sequences`	`/api/v1/sequences/{id}`	`Sequence`
Lighting presets	`/api/v1/presets`	`/api/v1/presets/{id}`	`LightingPreset`
Lighting groups	`/api/v1/lightingGroups`	`/api/v1/lightingGroups/{id}`	`LightingGroup`
Executors	`/api/v1/executors`	`/api/v1/executors/{id}`	`Executor`

The fields of each type are described in Project file. Notes per resource:

- **Tracks and cues** live inside their timeline: a missing parent timeline or track gets `404`. Changing them counts as changing the timeline (`timeline_running` applies).
- **Dashboards**: a `POST` or `PUT` publishes the dashboard directly. Core increases `version`, keeps the published version in the dashboard's history, and screens showing it reload. To edit without affecting screens, use the draft routes.
- **Notification channels**: Slack and Teams channels need `settings.webhookUrl`.

Live state

METHOD	PATH	SCOPE	DESCRIPTION
GET	`/api/v1/snapshot`	read	Every device's status and state, and every timeline's state (contract `Snapshot`: `{ devices: [DeviceSnapshot], timelines: [TimelineState] }`).

METHOD	PATH	SCOPE	DESCRIPTION
GET	`/api/v1/devices/{id}/state`	read	One device: `{ device, status, state }`. `status` is one of `connecting`, `online`, `degraded`, `offline`, `timeout`, `error`; `state` maps capability id to state values.
GET	`/api/v1/state/timelines`	read	Transport state of every timeline (array of `TimelineState`).
GET	`/api/v1/state/listeners`	read	Every listener: open or not, clients, message counts.
GET	`/api/v1/cluster/placement`	read	Shared mode: which node runs each device, and which devices moved off their preferred node.
POST	`/api/v1/cluster/rebalance`	control	Shared mode: spread the devices across the nodes again and bring moved devices back (`dryRun` to preview).
GET	`/api/v1/cluster`	read	Redundancy status (contract `ClusterStatus`): this node's role, the primary, quorum, clock offset and every node's health, sync state and cue log. A standalone node answers `{ "role": "standalone", "nodes": [] }`.
GET	`/api/v1/cluster/discovered`	read	Cores found on the local network that are not in this cluster (contract `DiscoveredCore`), each with its state: `new`, `separate` or `clustered`. Only a primary or standalone core lists them.
POST	`/api/v1/cluster/discovered/{id}/accept`	admin	Accept a discovered core: body `{ "mode": "separate" }` or `{ "mode": "redundant", "code": "482913" }` (contract `ClusterAccept`). `redundant` needs that core's pairing code; a wrong code answers `wrong_pairing_code`. See Redundancy.
GET	`/api/v1/cluster/pairing`	admin	This core's current 6-digit pairing code, to type on the main core.
GET	`/api/v1/events`	read	Live events as Server-Sent Events. See Event stream.

Intents

METHOD	PATH	SCOPE	DESCRIPTION
POST	`/api/v1/intents`	control	Send an intent (contract `IntentRequest`).
GET	`/api/v1/intents/{intentId}`	read	Current status of an intent (contract `IntentStatus`).

Request body:

FIELD	TYPE	REQUIRED	DESCRIPTION
`name`	string	yes	`power.on`, `power.off`, `level.set`, `mute.on`, `mute.off`, `input.select`, `transport.play`, `preset.recall` (any `<capability>.<action>` of the standard capabilities power, level, mute, input, transport, preset); `scene.recall` with `params.scene`; `action` with `params.capability`, `params.action`, `params.params`; `release` to drop priority holds.
`target`	object	yes	`{ "kind": "device", "id" }`, `{ "kind": "room", "id" }`, `{ "kind": "tag", "tag" }` or `{ "kind": "all" }`.
`params`	object	no	Passed to the device action. `holdMs` keeps this intent's priority on the state it sets for that long.
`priority`	integer 0-100	no	Higher wins on conflict. Default 30 for API calls. 90 and above hold until a `release`.
`at`	timestamp	no	When to run it.
`expiresAt`	timestamp	no	Drop it if not run by then.

Without `?wait`, core answers `202 Accepted` as soon as the intent is accepted, with a `Location: /api/v1/intents/{intentId}` header. With `?wait=5s` (or `500ms`, or a number of seconds; at most 60 s) it answers `200` with the final status, or the latest status when the time runs out.

Response (`IntentStatus`):

FIELD	DESCRIPTION
`intentId`	UUID of the intent.
`state`	`accepted`, `rejected`, `scheduled`, `running`, `completed`, `partial`, `failed` or `cancelled`.
`reason`	Why it was rejected or failed.

FIELD	DESCRIPTION
`devices`	`[{ "deviceId", "outcome": "ok" | "failed" | "overridden" | "skipped", "error" }]`.
`updatedAt`	Timestamp.

Timelines

METHOD	PATH	SCOPE	DESCRIPTION
POST	`/api/v1/timelines/{timeline}/transport`	control	Play, pause, stop, seek or GO. Body `{ "op": "play" | "pause" | "stop" | "seek" | "go", "positionMs" }`; `positionMs` is required for `seek`. Returns the new `TimelineState`.

`TimelineState` fields: `timelineId`, `status` (`stopped`, `playing`, `paused`, `waiting`), `positionMs`, `sampledAt` (when `positionMs` was sampled, for interpolation), `lastCueId`, `nextCueId`, `chasing`, `clockLost`, `anchor`. See Timelines.

Devices: tests, discovery, logs and removal

METHOD	PATH	SCOPE	DESCRIPTION
GET	`/api/v1/devices/{id}/log`	read	The device's log, newest first: `?limit`, `?before`, `?kinds` (action, status, state, event, note), `?level` (info, warning, error).
POST	`/api/v1/devices/{id}/log`	control	Write a note into the device's log: `{ "text", "level" }`.
DELETE	`/api/v1/devices/{id}/log`	configure	Empty the device's log.
GET	`/api/v1/devices/log`	read	Logs of several devices merged, newest first: `?devices=a,b` and the same filters.
POST	`/api/v1/devices/references`	read	What uses these devices and their sub-devices (body: ids): scenes, cues, automations, maps, dashboards, lighting, alert rules.
POST	`/api/v1/devices/remove`	configure	Delete devices, their sub-devices and their references in one change: body `{ "ids", "references": "remove" | "keep" }` (`keep` turns their timeline cues into comment cues), `?dryRun`.

METHOD	PATH	SCOPE	DESCRIPTION
POST	`/api/v1/devices/{id}/configurators/{configurator}/run`	configure	Run a setup assistant step through the device's module (body: `step`, `answers`, `config`). Settings it returns are saved (secrets to the credential store) and never sent back.
POST	`/api/v1/devices/{id}/test`	configure	Test a device's connection in a separate module process; the running device is not touched. Optional body `{ "config": { ... } }` tests other settings before you save them (stored secrets are filled in). Returns `{ ok, status, detail, state, elapsedMs }`; `ok` is false when the device did not come online. A test takes 10 s at most.
POST	`/api/v1/discovery`	configure	Scan the network (mDNS, SSDP and module searches). Optional body `{ "seconds": 1-30 }`, default 5. Returns `[{ address, name, manufacturer, model, via, service, suggestions }]`, where `suggestions` lists matching installed modules. Refused while a timeline plays (`409 show_mode`).

Dashboards

Besides the resource routes, dashboards have a draft and a version history. Drafts and published versions are kept in the Tangle data folder.

METHOD	PATH	SCOPE	DESCRIPTION
GET	`/api/v1/dashboards/{id}/draft`	configure	`{ "draft": true, "dashboard" }` when a draft exists, otherwise `{ "draft": false, "dashboard" }` with the published version.
PUT	`/api/v1/dashboards/{id}/draft`	configure	Save a draft (contract `Dashboard`). Screens keep the published version. Returns `{ "saved": true }`.
POST	`/api/v1/dashboards/{id}/publish`	configure	Publish the draft, or the `Dashboard` in the body if one is sent. Screens reload to it. Returns `{ version, dashboard }`. `404 no_draft` without a draft or body.
GET	`/api/v1/dashboards/{id}/versions`	read	Published versions, newest first: `[{ version, publishedAt, by, name }]`.
GET	`/api/v1/dashboards/{id}/versions/{version}`	read	One published version: `{ version, publishedAt, by, dashboard }`.

METHOD	PATH	SCOPE	DESCRIPTION
POST	`/api/v1/dashboards/{id}/versions/{version}/restore`	configure	Publish an old version again, as a new version.
POST	`/api/v1/dashboards/{id}/session`	none	Open a dashboard in a browser. Optional body `{ "pin": "1234" }`. Returns `{ token, expiresAt, dashboardId, access }`.

Dashboard checks name the page and widget at fault, for example `page main, widget w1: unknown widget type knob`.

The session route follows the dashboard's `access.mode`: `open` dashboards get a session at once; `pin` dashboards need the right PIN (`401 pin_required` or `wrong_pin`, `429 too_many_attempts` after 5 wrong PINs in a minute); `panels` dashboards answer `403 panels_only` (enrolled panels use their own API token). A session stops working when it expires or when the dashboard switches to `panels`.

Automations

METHOD	PATH	SCOPE	DESCRIPTION
POST	`/api/v1/automations/{id}/run`	control	Run an automation now. Optional body `{ "force": true, "params": {} }`: `force` skips its conditions, `params` fills its params. Returns the contract `AutomationRun` (`runId`, `automationId`, `state`, `trigger`, `reason`, `step`, `startedAt`, `endedAt`): started, or skipped with a reason. `409 not_primary` on a backup.
POST	`/api/v1/automations/{id}/stop`	control	Stop every run of the automation in progress.
GET	`/api/v1/automations/{id}/runs`	read	The last 50 runs, newest first (kept in memory).
POST, GET	`/api/v1/macros/{id}/run`, `/runs`, `/stop`	control, read	Deprecated aliases (macros are automations since contract 0.2.30). `run` does not check conditions.

See Automations.

Variables, controls and functions

METHOD	PATH	SCOPE	DESCRIPTION
GET	`/api/v1/variables`	read	Project variables with their current values.
GET	`/api/v1/variables/{name}`	read	One project variable.
PUT	`/api/v1/variables/{name}`	control	Set a project variable: `{ "value": ... }`. Variable triggers fire.
GET	`/api/v1/controls`	read	Dashboard controls with a kept value or override (value, label, visible, enabled, props).
GET	`/api/v1/controls/{id}`	read	One control's state.
PUT	`/api/v1/controls/{id}`	control	Set a control (`ControlSetRequest`: value, label, visible, enabled; `null` clears). Every screen follows and control triggers fire.
POST	`/api/v1/controls/{id}/events`	control	Report a control event (`press`, `release`, `click` or `hold`); control triggers waiting for it fire.
POST	`/api/v1/controls/{id}/rename`	configure	Rename a control: `{ "to" }`. Every reference in the project and in dashboard drafts follows.
GET	`/api/v1/functions`	read	Every library function (core `math`, `text`, `time`, `list`, `color`, and library modules) with its signature.
POST	`/api/v1/functions/{name}/call`	control	Call a function: `{ "args": [...] }` or named `{ "args": {...} }`. Returns its value.

Listeners

Besides the resource routes:

METHOD	PATH	SCOPE	DESCRIPTION
GET	`/api/v1/listeners/{id}/log`	read	Messages the listener received, oldest first: `?since=<seq>&limit=<n>`.
GET	`/api/v1/listeners/{id}/runs`	read	Recent runs of its rules' steps, newest first.
POST	`/api/v1/listeners/{id}/test`	configure	Check a message against its rules: `{ "text" }` (or `bytes`, `osc`, `http`), dry run unless `"run": true`.

See Listeners.

Device creator and community

METHOD	PATH	SCOPE	DESCRIPTION
GET	`/api/v1/drivers`	read	Drivers made with the device creator.
GET	`/api/v1/drivers/{id}`	read	A driver's definition and code.
GET	`/api/v1/drivers/{id}/files`	read	Every file of the driver's module, to download it.
POST	`/api/v1/drivers/check`	configure	Check a definition and its code without saving.
POST	`/api/v1/drivers`	configure	Save a driver: builds and installs its module; devices using it restart.
DELETE	`/api/v1/drivers/{id}`	configure	Delete a driver's source (its installed module keeps working).
GET	`/api/v1/community`	read	The community server and who is signed in on this core.
POST	`/api/v1/community/sign-in`	configure	Sign in with the license account's email and password.
POST	`/api/v1/community/register`	configure	Choose the username (namespace) at the first sign-in.
POST	`/api/v1/community/sign-out`	configure	Sign out on this core.
POST	`/api/v1/community/password-reset`	configure	Email a password reset link.
GET	`/api/v1/community/search`	read	Search drivers: `?q=`, `scope=mine` (default) or `scope=all`.
POST	`/api/v1/community/install`	configure	Install or update a community driver; devices using it restart.
POST	`/api/v1/community/publish`	configure	Publish a device creator driver under your namespace.

See Device creator.

Maps, plans and images

METHOD	PATH	SCOPE	DESCRIPTION
POST	`/api/v1/plans`	configure	Import a map plan: the file as the body (SVG, DXF, DWG, PNG or JPEG), `?name=`. DWG needs a converter on the core's computer.
GET	`/api/v1/plans/{file}`	read	A stored plan file.
GET	`/api/v1/maps/{id}/positions`	read	Placed devices and live positions from tracking sources.
POST	`/api/v1/positions`	control	Push positions from a tracking source: 1 to 1000 positions, in map metres. Returns `{ accepted, rejected }`.
POST	`/api/v1/assets`	configure	Upload a dashboard image (PNG, JPEG, WebP, GIF or SVG, 10 MB at most): the file as the body, `?name=`.
GET	`/api/v1/assets/{file}`	read	A stored dashboard image. Its name is its content hash, so it never changes.
GET	`/api/v1/views/public`	none	The end-user views a browser can open (id, name, skin, theme, access mode).
POST	`/api/v1/views/{id}/session`	none	Open an end-user view in a browser; returns a session token.

Lighting

METHOD	PATH	SCOPE	DESCRIPTION
GET	`/api/v1/lighting/state`	read	Every sequence's live cue, level and pause, the grand master, and your programmer.
POST	`/api/v1/lighting/command`	control	Play lighting: go, back, goto, off, fader, flash, pause a sequence, or the grand master.
POST	`/api/v1/lighting/programmer`	control	Change your programmer: select lights, set values, fan, blind, highlight, clear.
POST	`/api/v1/lighting/store`	configure	Store the programmer into a cue, a preset or a group.
GET	`/api/v1/lighting/library`	read	Search the bundled fixture library: `?q=<manufacturer or model>&limit=50`.

Sentinel (monitoring)

Monitoring is part of Tangle Pro and Ultimate. See Sentinel.

METHOD	PATH	SCOPE	DESCRIPTION
GET	`/api/v1/sentinel`	read	Overview: `active`, `licensed`, `counts` (`critical`, `warning`, `info`, `pending`), firing `alerts`, `rooms`, health of `devices`, `cores` and `probes`, `metrics` count, `notificationsQueued`.
GET	`/api/v1/sentinel/alerts`	read	`{ "alerts": [Alert] }` (pending and firing). `?history=true` adds `history` (recent transitions, newest first); `?limit=` caps it (default 100).
POST	`/api/v1/sentinel/ack`	control	Acknowledge an alert: `{ "alertId", "note" }`. Stops repeat notifications. Returns the alert.
POST	`/api/v1/sentinel/silence`	control	Silence one alert or every alert of a rule: `{ "alertId" }` or `{ "ruleId" }`, plus `until` (timestamp in the future, required). Returns `{ until, alerts, ruleId }`.
POST	`/api/v1/sentinel/metrics`	control	Push custom metric samples: `[{ "name", "value", "labels", "at" }]` (1 to 1000 per call; `{ "samples": [...] }` also works). Returns `{ "accepted": n }`.
GET	`/api/v1/sentinel/metrics`	read	Current custom and probe metric values. `?samples=true` adds 24 h of per-minute samples.
POST	`/api/v1/sentinel/channels/{id}/test`	configure	Send a test notification. Returns `{ ok, channelId, error }`.

Alert ids look like `<ruleId>/<subject kind>:<subject id>`, for example `offline/device:proj-1`. Acknowledging and silencing run on the primary only (`409 not_primary` on a backup).

License

METHOD	PATH	SCOPE	DESCRIPTION
GET	`/api/v1/license`	read	Contract `LicenseInfo`: `mode` (`licensed`, `designer`, `demo`), `status` (state, edition, entitlements, limits, dates), `features`, `session` (`paused`, `endsAt`) in designer or demo mode, `pendingRestart`, `notes`.
POST	`/api/v1/license/resume`	control	Resume device connections and timelines after the hourly pause of the free designer or demo mode. Returns the new `LicenseInfo`. `409 not_paused` when nothing is paused.

METHOD	PATH	SCOPE	DESCRIPTION
POST	`/api/v1/license/activate`	admin	Activate this computer with a license key (applies at the next start).
POST	`/api/v1/license/sign-in`	admin	Sign in with the license owner's email and password; lists their licenses, or activates the one chosen.
POST	`/api/v1/license/password-reset`	admin	Email a password reset link to a license owner.
POST	`/api/v1/license/request-file`	admin	A license request for this computer, to email to Tangle licensing.
POST	`/api/v1/license/import`	admin	Install a license file made elsewhere (offline activation, or the answer to a request).
POST	`/api/v1/license/refresh`	admin	Fetch a fresh license file now.
POST	`/api/v1/license/deactivate`	admin	Remove this computer from its installation.
PUT	`/api/v1/license/offline`	admin	Offline mode: never contact the license server.

See Licensing.

Event stream (SSE)

`GET /api/v1/events?topics=<list>` streams live events as Server-Sent Events. `topics` is required: a comma-separated list of `<event type>:<subject id>` or `<event type>:*`.

```sh
curl -N "http://127.0.0.1:7340/api/v1/events?topics=device.state:*,timeline.state:show" \
  -H "Authorization: Bearer $TOKEN"
```

```
event: device.state:proj-1
data: {"type":"device.state","deviceId":"proj-1","capability":"tangle.power","values":
{"on":true},"at":1760000000000}
```
TEXT

- Each SSE event name is the topic; the data is the event JSON (contract `TangleEvent`).
- A client that falls behind gets an event named `lagged` with `{"missed": n}`.
- A keep-alive comment is sent every 15 seconds.
- The topics are the same as on the WebSocket. See WebSocket events for the list and each event's fields.

In a browser, use `EventSource` with `?token=`:

```
const es = new EventSource(`/api/v1/events?topics=device.status:*&token=${token}`);
es.addEventListener("device.status:proj-1", (e) ⇒ console.log(JSON.parse(e.data)));
```
JS

Examples

Set a shell variable with your token first:

```
export TOKEN=tgl_...
```
SH

Read the project:

```
curl http://127.0.0.1:7340/api/v1/project -H "Authorization: Bearer $TOKEN"
```
SH

Recall a scene in a room and wait up to 5 s for the result:

```
curl -X POST "http://127.0.0.1:7340/api/v1/intents?wait=5s" \
  -H "Authorization: Bearer $TOKEN" -H "content-type: application/json" \
  -d '{"name":"scene.recall","target":{"kind":"room","id":"room-a"},"params":
{"scene":"presentation"}}'
```
SH

Check a new device before adding it, then add it:

```
curl -X POST "http://127.0.0.1:7340/api/v1/devices?dryRun=true" \
  -H "Authorization: Bearer $TOKEN" -H "content-type: application/json" \
  -d '{"id":"pj-2","name":"Projector 2","moduleId":"misc.pjlink.projector","room":"room-
a","config":{"host":"10.0.0.21"}}'

curl -X POST http://127.0.0.1:7340/api/v1/devices \
  -H "Authorization: Bearer $TOKEN" -H "content-type: application/json" \
  -d '{"id":"pj-2","name":"Projector 2","moduleId":"misc.pjlink.projector","room":"room-
a","config":{"host":"10.0.0.21"}}'
```
SH

Update a scene safely (read its ETag, then send it back):

```sh
curl -i http://127.0.0.1:7340/api/v1/scenes/presentation -H "Authorization: Bearer $TOKEN"
# ETag: "3f2a9c1d0b7e4a55"

curl -X PUT http://127.0.0.1:7340/api/v1/scenes/presentation \
  -H "Authorization: Bearer $TOKEN" -H "content-type: application/json" \
  -H 'If-Match: "3f2a9c1d0b7e4a55"' \
  -d '{"id":"presentation","name":"Presentation","actions":[{"deviceId":"pj-
2","capability":"tangle.power","action":"on"}]}'
```

Start a timeline and jump in a cue list:

```sh
curl -X POST http://127.0.0.1:7340/api/v1/timelines/show/transport \
  -H "Authorization: Bearer $TOKEN" -H "content-type: application/json" -d '{"op":"play"}'

curl -X POST http://127.0.0.1:7340/api/v1/timelines/show/transport \
  -H "Authorization: Bearer $TOKEN" -H "content-type: application/json" -d
'{"op":"seek","positionMs":90000}'
```

Push a metric from a script:

```sh
curl -X POST http://127.0.0.1:7340/api/v1/sentinel/metrics \
  -H "Authorization: Bearer $TOKEN" -H "content-type: application/json" \
  -d '[{"name":"room.temperature","value":31.5,"labels":{"room":"server"}}]'
```

REFERENCE

WebSocket

The live client protocol at /ws: hello and authentication, subscriptions, events, intents, timeline control, automations, sentinel and errors.

tangle-core serves a WebSocket at `ws://<host>:7340/ws`. Dashboards use it, and so can any tool that needs live state with low latency. It is for live control: to change the project, use the REST API or MCP.

Overview

- Every message is one JSON object in a text frame.
- Every message carries `"v": 1` (the protocol version) and a `type`.
- Requests from the client carry an `id` (any non-empty string). Core answers each request with an `ack` or an `error` that has the same `id`.
- The client sends `hello` first. Core answers `welcome` with a full snapshot of every device and timeline.
- After that, the client subscribes to topics and receives `event` messages, sends intents, drives timelines, and asks for snapshots.

```text
client                              core
  | hello ------------------------->  |
  |  <-------------------- welcome    |  (snapshot)
  | subscribe ---------------------->  |
  |  <----------------------- ack     |
  |  <---------------------- event    |  (as things change)
  | intent ------------------------->  |
  |  <----------------------- ack     |  (IntentStatus)
```

The message types are defined in the contract (`ClientMessage`, `ServerMessage`), so clients can validate them.

Connect and authenticate

Send `hello` within 5 seconds of opening the socket, or core closes it.

```json
{ "v": 1, "id": "1", "type": "hello", "client": "my-tool/1.0.0", "token": "tgl_..." }
```

FIELD	REQUIRED	DESCRIPTION
`v`	yes	Protocol version, `1`.
`id`	yes	Request id.
`type`	yes	`"hello"`.
`client`	yes	Client name and version, for example `tangle-interact/0.1.0`. Shown in logs and the audit log.
`token`	no	An API token (`tgl_...`) or a dashboard session token (`tgd_...`).

Who may connect:

- **This machine, no token:** clients connecting from the same machine (loopback) may leave out `token`. They get the `read` and `control` scopes, like a local dashboard.
- **Other machines:** must send a token. Without one, core answers `error` with code `unauthorized` and closes the socket.
- **With a token:** the client gets that token's scopes. Core checks the token again every 2 seconds; when it is revoked or a dashboard session expires, core sends an `error` (no `id`) and closes the socket.

Create tokens with `tangle-core token create` (see CLI). Dashboards opened in a browser get a session token from `POST /api/v1/dashboards/{id}/session` (see REST API). Scopes are described in REST API.

If the first message is not a valid `hello`, core answers `error` and closes the socket: `hello_required` (another type), `unsupported_version` (`v` is not 1) or `invalid_message`.

welcome

```JSON
{
  "v": 1,
  "id": "1",
  "type": "welcome",
  "nodeId": "core-a",
  "role": "primary",
  "contractVersion": "0.2.43",
  "cluster": { "role": "primary", "nodes": [] },
  "license": { "mode": "licensed", "status": { }, "features": { } },
  "snapshot": {
    "devices": [
      {
        "device": { "id": "proj-1", "name": "Projector", "moduleId": "misc.pjlink.projector"
},
        "status": "online",
        "state": { "tangle.power": { "on": true } }
      }
    ],
    "timelines": []
  }
}
```

FIELD	DESCRIPTION
`nodeId`	Name of this core node (`--node-id`, default `core-1`).
`role`	`standalone`, `primary` or `backup`.
`contractVersion`	Contract version of this core.
`cluster`	Redundancy status (contract `ClusterStatus`). See Redundancy.
`license`	Contract `LicenseInfo` (mode, status, features, session). Present when licensing is active.
`snapshot`	`{ devices: [DeviceSnapshot], timelines: [TimelineState] }`.

Each `DeviceSnapshot` has `device` (the project's device), `status` (`connecting`, `online`, `degraded`, `offline`, `timeout` or `error`) and `state` (capability id, then state key, then value).

Client messages

TYPE	SCOPE	FIELDS	ACK RESULT
`subscribe`	read	`topics`: one or more topics	`{ "topics": [...] }`, the full list now subscribed

TYPE	SCOPE	FIELDS	ACK RESULT
`unsubscribe`	none	`topics`: one or more topics	`{ "topics": [...] }`, the list left
`intent`	control	`intent`: an `IntentRequest`	`IntentStatus` (first status)
`timeline`	control	`control`: a `TimelineControl`	`TimelineState`
`automation.run`	control	`automationId`, `force`, `params` (optional)	`AutomationRun`
`variable.set`	control	`name`, `value`	The variable's new state
`control.set`	control	`controlId`, and `value`, `add`, `label`, `visible`, `enabled` or `props`	The control's state
`control.event`	control	`controlId`, `event` (`press`, `release`, `click`, `hold`), `value` (optional)	`{ "published": true }`
`positions.push`	control	`positions`: 1 to 1000 positions, in map metres	`{ accepted, rejected }`
`snapshot`	read	none	none: core answers with a `snapshot` message instead of `ack`
`sentinel.status`	read	none	Sentinel overview
`sentinel.alerts`	read	`history` (optional boolean)	`{ alerts, history }`
`sentinel.ack`	control	`alertId`, `note` (optional)	The alert
`sentinel.silence`	control	`alertId` or `ruleId`, `until`	`{ until, alerts, ruleId }`
`sentinel.metrics`	control	`samples`: 1 to 1000 `MetricSample`	`{ "accepted": n }`
`license.status`	read	none	`LicenseInfo`
`license.resume`	control	none	`LicenseInfo`

Every client message needs `v`, `id` and `type`. Unknown fields are refused (`invalid_message`). Sending `hello` again gets `invalid_message`.

subscribe and unsubscribe

```JSON
{ "v": 1, "id": "2", "type": "subscribe", "topics": ["device.state:*", "timeline.state:main-show"] }
```

A topic is `<event type>:<subject id>` or `<event type>:*` (every subject). Subscriptions add up; `unsubscribe` removes the exact topics given.

intent

Makes something happen on devices. The `intent` object is the same as the REST body (see REST API intents).

```JSON
{
  "v": 1,
  "id": "3",
  "type": "intent",
  "intent": { "name": "scene.recall", "target": { "kind": "room", "id": "room-a" }, "params": { "scene": "presentation" } }
}
```

The `ack` carries the first `IntentStatus` (usually `accepted`, or `scheduled` or `rejected`). Subscribe to `intent.status:*`, or `intent.status:<intentId>`, to follow it to `completed`, `partial` or `failed`. Intents sent over the WebSocket are recorded with source `dashboard`.

timeline

```JSON
{ "v": 1, "id": "4", "type": "timeline", "control": { "op": "play", "timelineId": "main-show" } }
```

`CONTROL` FIELD	REQUIRED	DESCRIPTION
`op`	yes	`play`, `pause`, `stop`, `seek` or `go` (next cue of a cue list).
`timelineId`	yes	The timeline.
`positionMs`	for `seek`	Position from the start, in ms.

The `ack` carries the new `TimelineState`. See Timelines.

automation.run

```JSON
{ "v": 1, "id": "7", "type": "automation.run", "automationId": "close-room", "force": false }
```

Runs an automation now. Its conditions are checked unless `force` is true. The `ack` carries the run: `{ runId, automationId, state, trigger, reason, startedAt }`, with `state` `started`, or `skipped` and a `reason`. A disabled automation gets `error` with code `disabled`; on a backup node, `not_primary`. See Automations.

snapshot

```JSON
{ "v": 1, "id": "5", "type": "snapshot" }
```

Core answers with a `snapshot` message (same `id`) holding the full current state.

Sentinel messages

```JSON
{ "v": 1, "id": "9", "type": "sentinel.ack", "alertId": "offline/device:proj-1", "note": "on my way" }
```

```JSON
{ "v": 1, "id": "10", "type": "sentinel.metrics", "samples": [{ "name": "room.temperature", "value": 22.5, "labels": { "room": "lobby" } }] }
```

`until` is a timestamp in ms since the Unix epoch and must be in the future. `sentinel.alerts` returns up to 100 history entries. These work as on the REST API; monitoring is part of Tangle Pro and Ultimate. See Sentinel.

License messages

```JSON
{ "v": 1, "id": "r1", "type": "license.resume" }
```

In the free designer and demo modes, device connections stop and playing timelines pause every hour. `license.resume` resumes them for another hour. See Licensing.

Server messages

TYPE	WHEN	FIELDS
`welcome`	Answer to `hello`	See welcome.
`ack`	A request succeeded	`id`, `result`
`error`	A request failed, or the session ends	`id` (when it answers a request), `code`, `message`
`snapshot`	Answer to `snapshot`, or after the client fell behind	`id` (when it answers a request), `snapshot`

TYPE	WHEN	FIELDS
`event`	Something you subscribed to changed	`topic`, `event`

```json
{ "v": 1, "id": "3", "type": "ack", "result": { "intentId": "8c0e...", "state": "accepted",
"updatedAt": 1760000000000 } }
```

A client that reads too slowly to keep up with events gets a `snapshot` message (without `id`) holding the current state, in place of the events it missed.

Events

```json
{
  "v": 1,
  "type": "event",
  "topic": "device.state:proj-1",
  "event": { "type": "device.state", "deviceId": "proj-1", "capability": "tangle.power",
"values": { "on": false }, "at": 1760000000000 }
}
```

`event.at` is a timestamp in ms. The same topics and event bodies are available as Server-Sent Events at `GET /api/v1/events` (see REST API).

TOPIC	EVENT FIELDS	WHEN
`device.state:<deviceId>`	`deviceId`, `capability`, `values`, `at`	A device's state changed. `values` holds the changed keys of that capability.
`device.status:<deviceId>`	`deviceId`, `status`, `detail`, `at`	A device's status changed (`connecting`, `online`, `degraded`, `offline`, `timeout`, `error`).
`device.event:<deviceId>`	`deviceId`, `name`, `data`, `at`	A free-form device event, such as a button press.
`cluster.discovered:*`	the discovered core (contract `DiscoveredCore`)	A core not in this cluster appeared on the network. Sent by a primary or standalone core, once per core.
`intent.status:<intentId>`	`status` (`IntentStatus`), `at`	An intent's status changed.

TOPIC	EVENT FIELDS	WHEN
`timeline.state:<timelineId>`	`state` (`TimelineState`), `at`	A timeline's transport state changed.
`project.changed:<projectId>`	`projectId`, `revision`, `changes`, `at`	The project configuration changed (REST, MCP or a reload). `revision` increases by one per change; `changes` lists `{ op, kind, id, parent }`.
`module.status:<moduleId>`	`moduleId`, `status`, `restarts`, `retryInMs`, `detail`, `at`	A module process started (`running`), crashed (`restarting`), crashes too often (`unstable`) or stopped (`stopped`).
`module.health:<moduleId>`	`moduleId`, `cpuPercent`, `memoryBytes`, `avgLatencyMs`, `actions`, `warning`, `at`	Every 10 s while a module runs. `warning` is set when its average action latency is above 500 ms.
`automation.run:<automationId>`	`run` (`AutomationRun`), `at`	An automation run started, ended (`completed`, `partial`, `failed`, `cancelled`) or was skipped.
`device.action:<deviceId>`	the action, its source and its outcome (`sent`, `pending`, `progress`, `done`, `failed`)	An action was sent to a device, or progressed (a projector warming up).
`device.log:<deviceId>`	the log entry	A device log entry was added.
`variable.changed:<name>`	`name`, `value`, `previous`, `by`, `at`	A project variable changed.
`control.changed:<controlId>`	`id`, `state`, `previous`, `by`, `at`	A dashboard control's value or properties changed.
`control.event:<controlId>`	`id`, `event`, `value`, `by`, `at`	Someone pressed, released, clicked or held a control.
`listener.message:<listenerId>`	the log entry: message, matched rule, captures, runs, reply	A listener received a message.

TOPIC	EVENT FIELDS	WHEN
`listener.status:<listenerId>`	`listener`, `at`	A listener opened, closed, failed or its client count changed.
`dashboard.draft:<dashboardId>`	`id`, `by`, `at`	A dashboard draft was saved (builders and previews elsewhere refresh).
`cluster.placement:core`	`placement`, `reason`, `at`	Shared mode: the primary placed devices again (start, node down or back, rebalance).
`project.switching:<projectId>`	`projectId`, `nextProjectId`, `at`	The open project is about to close (another opens, or none). Clients disconnect for a moment and reload.
`sentinel.alert:<ruleId>`	`transition`, `alert`, `channelId`, `detail`, `at`	An alert changed (`pending`, `firing`, `resolved`, `dropped`, `repeat`, `acknowledged`, `silenced`), or a notification was sent (`notified`) or failed (`notifyFailed`).
`license.status:core`	`license` (`LicenseInfo`), `at`	The license changed, or a designer or demo session paused or resumed.

Topic subjects follow the id rules (lowercase letters, digits, `.`, `_`, `-`) or are a UUID (intents). Examples: `device.status:*`, `timeline.state:main-show`, `automation.run:*`, `license.status:core`.

Errors

```JSON
{ "v": 1, "id": "4", "type": "error", "code": "missing_scope", "message": "this token lacks the `control` scope" }
```

CODE	MEANING
`hello_required`	The first message was not `hello`. The socket closes.
`unsupported_version`	`v` is not 1. The socket closes.
`invalid_message`	Not JSON, missing or unknown fields, a bad topic, a second `hello`, or an invalid intent or timeline control (the message lists each problem).

CODE	MEANING
`unknown_type`	Unknown `type`.
`unauthorized`	No token from another machine, unknown or revoked token, expired dashboard session.
`missing_scope`	The token lacks the scope this message needs.
`not_licensed`	Monitoring is not part of this edition.
`not_primary`	This node is a redundancy backup: connect to the primary to control.
`timeline_not_found`, `automation_not_found`, `alert_not_found`, `alertRule_not_found`	Unknown id.
`unsupported`	The timeline does not support this operation.
`disabled`	The automation is disabled.
`not_paused`	`license.resume` when nothing is paused.
`unavailable`	A service is not ready.

The codes are the same as the REST API's (see REST API errors), except that validation problems use `invalid_message`.

Example: a minimal client

```JS
const ws = new WebSocket("ws://127.0.0.1:7340/ws");
let next = 1;
const send = (msg) ⇒ ws.send(JSON.stringify({ v: 1, id: String(next++), ...msg }));

ws.onopen = () ⇒ send({ type: "hello", client: "example/1.0.0" });
ws.onmessage = (e) ⇒ {
  const msg = JSON.parse(e.data);
  if (msg.type ≡≡ "welcome") {
    console.log("devices:", msg.snapshot.devices.map((d) ⇒ `${d.device.id} ${d.status}`));
    send({ type: "subscribe", topics: ["device.status:*", "timeline.state:*"] });
    send({ type: "intent", intent: { name: "power.on", target: { kind: "room", id: "room-a" }
} });
  } else if (msg.type ≡≡ "event") {
    console.log(msg.topic, msg.event);
  } else if (msg.type ≡≡ "error") {
    console.error(msg.code, msg.message);
  }
};
```

Run on the same machine as tangle-core, this needs no token. From another machine, add `token` to the `hello` and start tangle-core with remote access (see CLI).

REFERENCE

MCP

Let AI assistants read, configure and run a Tangle project through the Model Context Protocol, with show protection.

tangle-core includes an MCP (Model Context Protocol) server, so an AI assistant can read a project, build rooms and cue lists, control devices and check monitoring. Every tool goes through the same command layer as the REST API: the same validation, scopes and audit log, plus a protection that keeps assistants away from a running show.

Two ways to connect

TRANSPORT	WHERE	FOR
Streamable HTTP	`http://<host>:7340/mcp`	Assistants and tools that speak MCP over HTTP.
stdio bridge	`tangle-core mcp`	Desktop assistants that only start MCP servers as local commands.

Both need a running tangle-core and an API token. Create a token for the assistant on the machine that runs tangle-core:

```sh
tangle-core token create --name claude --scopes read,control,configure
```

The secret is printed once. Choose the scopes for what the assistant should be able to do (see Scopes).

> **Note:** MCP access is part of Tangle Pro and Ultimate (and the demo mode). On other editions, MCP requests get `403 not_licensed`. See Licensing.

Streamable HTTP

Send the token as a bearer token on every request:

```http
POST /mcp HTTP/1.1
Host: 127.0.0.1:7340
Authorization: Bearer tgl_...
Accept: application/json, text/event-stream
Content-Type: application/json
```

A request without a valid token gets `401` with a problem-details body. Unless tangle-core runs with `--allow-remote`, the MCP endpoint accepts only requests addressed to this machine (a loopback `Host`).

stdio bridge

`tangle-core mcp` reads JSON-RPC messages from stdin, forwards each one to the running core's `/mcp` endpoint, and writes every answer and notification to stdout.

```sh
TANGLE_TOKEN=tgl_... tangle-core mcp
tangle-core mcp --url http://10.0.0.5:7340 --token tgl_...
```

OPTION	DEFAULT	DESCRIPTION
`--url <url>`	`http://127.0.0.1:7340`	Address of the running tangle-core.
`--token <token>`	`TANGLE_TOKEN`	API token. Reads the `TANGLE_TOKEN` environment variable when not given.

If the core cannot be reached, the bridge answers each request with a JSON-RPC error that says so. When stdin closes, the bridge ends its MCP session and exits.

Configure an assistant

Most desktop assistants read a JSON file that lists MCP servers. Add Tangle with the bridge:

```json
{
  "mcpServers": {
    "tangle": {
      "command": "tangle-core",
      "args": ["mcp"],
      "env": { "TANGLE_TOKEN": "tgl_..." }
    }
  }
}
```

- If `tangle-core` is not on the assistant's `PATH`, give its full path in `command`.
- To reach a core on another machine, add `"--url", "http://10.0.0.5:7340"` to `args`. That core must run with remote access (`--listen 0.0.0.0:7340 --allow-remote --insecure-http`, see CLI).
- Restart the assistant after editing its configuration.

The server introduces itself as `tangle-core` and gives the assistant these instructions: read the project with `get_project` and live state with `get_snapshot`; change the configuration with the `create_*`, `update_*`

and `delete` tools (dry run first); make things happen with `send_intent`; drive shows with `control_timeline`; never interrupt a running show without being asked.

Scopes

Each tool needs one scope (shown in its description, which the assistant reads).

SCOPE	ALLOWS
`read`	Every tool that only reads: `get_*`, `list*`, `search_fixture_library`, `export_project`, `device_references`.
`control`	Making things happen: `send_intent`, `control_timeline`, `run_automation`, `stop_automation`, `set_variable`, `set_control`, `press_control`, `call_function`, lighting playback and programmer, `rebalance_cluster`, `resume_session`, alerts and metrics, device log notes, positions.
`configure`	Changing the project and the core: `create_*`, `update_*`, `delete`, `remove_devices`, drivers, modules, projects, templates, dashboards drafts and publishing, tests, discovery, setup assistants, community.
`admin`	`manage_license`, `accept_discovered_core`, `get_pairing_code`.
`live`	Allows changes and control while a timeline plays (see below).

The tables below give each tool's scope.

Resources and live updates need `read`.

Show protection

While any timeline is playing, an assistant whose token lacks the `live` scope cannot change anything. Every tool that edits the project, sends an intent, drives a timeline or runs an automation is refused with code `show_running`:

```json
{
  "type": "https://tangle.dev/problems/show_running",
  "title": "Conflict",
  "status": 409,
  "detail": "the show is running (timeline main-show is playing); nothing was changed",
  "code": "show_running"
}
```

Reading always works. Give a token `live` only when an assistant should act during a show:

```sh
tangle-core token create --name show-assistant --scopes read,control,live
```

Other protections apply to every client, with or without `live`:

- **Live timelines:** editing a timeline (or its tracks and cues) while it plays is refused (`timeline_running`) unless the tool call sets `allowLive: true`.
- **Network scans:** `discover_devices` is refused while a timeline plays (`show_mode`).
- **Deleting:** `delete` only previews unless `confirm` is true.
- **Redundancy:** on a backup node, changes and control are refused with `not_primary`.

Results and errors

Tool results are JSON (structured content). A list is wrapped as `{ "result": [...] }`. When a tool fails, the result is an error whose content is a problem-details object with a stable `code`, `detail` and, for validation errors, `errors` listing each problem. The codes are the REST API's (see REST API errors); an unknown tool name gets `unknown_tool`.

Configuration tools (`create_*`, `update_*`, `delete`) return:

FIELD	DESCRIPTION
`dryRun`	True when nothing was applied.
`result`	The resource after the change (or as it would be). Absent after a delete.
`changes`	`[{ op, kind, id, parent }]`: what changed.
`revision`	Project revision after the change.
`note`	On a `delete` without `confirm`: says nothing was deleted.

Tools

Project and configuration

TOOL	SCOPE	PURPOSE	ARGUMENTS
`get_project`	read	Read the whole project.	none
`list_modules`	read	Installed device modules with their capabilities and the devices using each.	none

TOOL	SCOPE	PURPOSE	ARGUMENTS
`get_cluster_status`	read	Redundancy status: role, primary, every node's health, sync state and cue log.	none
`list`	read	List resources of one kind, the tracks of a timeline, or the cues of a track.	`kind` (required), `timelineId`, `trackId`
`get`	read	Read one resource by id.	`kind`, `id` (required), `timelineId`, `trackId`
`delete`	configure	Delete a resource. Without `confirm: true` it only previews. Refused if the project would become invalid (for example a scene still uses the device).	`kind`, `id` (required), `timelineId`, `trackId`, `confirm`, `dryRun`, `allowLive`
`create_<kind>`	configure	Create a resource. The whole project is validated first; a mistake changes nothing.	`<kind>` (the resource, required), `timelineId`, `trackId`, `dryRun`, `allowLive`
`update_<kind>`	configure	Replace a resource with a complete new version (read it first with `get`). Creates it if missing.	same as `create_<kind>`

`kind` for `list`, `get` and `delete` is one of: `room`, `device`, `scene`, `timeline`, `track`, `cue`, `automation`, `alertRule`, `notificationChannel`, `maintenanceWindow`, `probe`. Tracks need `timelineId`; cues need `timelineId` and `trackId`.

The create and update tools exist for every resource type; their input schema is the contract type, so the assistant sees every field:

TOOLS	ARGUMENT	CONTRACT TYPE
`create_room`, `update_room`	`room`	`Room`
`create_device`, `update_device`	`device`	`DeviceInstance`
`create_scene`, `update_scene`	`scene`	`Scene`
`create_timeline`, `update_timeline`	`timeline`	`Timeline`
`create_track`, `update_track`	`track`, `timelineId`	`Track`
`create_cue`, `update_cue`	`cue`, `timelineId`, `trackId`	`Cue`
`create_dashboard`, `update_dashboard`	`dashboard`	`Dashboard`

TOOLS	ARGUMENT	CONTRACT TYPE
`create_automation`, `update_automation`	`automation`	`Automation`
`create_alertRule`, `update_alertRule`	`alertRule`	`AlertRule`
`create_notificationChannel`, `update_notificationChannel`	`notificationChannel`	`NotificationChannel`
`create_maintenanceWindow`, `update_maintenanceWindow`	`maintenanceWindow`	`MaintenanceWindow`
`create_probe`, `update_probe`	`probe`	`Probe`
`create_listener`, `update_listener`	`listener`	`Listener`
`create_map`, `update_map`	`map`	`Map`
`create_view`, `update_view`	`view`	`EndUserView`
`create_sequence`, `update_sequence`	`sequence`	`Sequence`
`create_preset`, `update_preset`	`preset`	`LightingPreset`
`create_lightingGroup`, `update_lightingGroup`	`lightingGroup`	`LightingGroup`
`create_executor`, `update_executor`	`executor`	`Executor`

The fields of each type are in Project file. `dryRun: true` validates and returns the changes without applying them. Device settings are checked against the module's settings schema, and secret settings go to the OS credential store.

TOOL	SCOPE	PURPOSE	ARGUMENTS
`list_available_modules`	read	Every module this core knows: installed, update available, or bundled and ready to install.	none
`install_module`	configure	Install or update a module. Devices using it restart; refused while a timeline plays if any would.	`moduleId`
`device_references`	read	What uses these devices and their sub-devices. Call it before `remove_devices`.	`ids`
`remove_devices`	configure	Delete devices, their sub-devices and their references in one change. Without `confirm: true` it only previews.	`ids`, `references` (`remove` or `keep`), `confirm`, `allowLive`

TOOL	SCOPE	PURPOSE	ARGUMENTS
`run_device_setup`	configure	Run one step of a device's setup assistant through its module. Settings it returns are saved on the device.	`deviceId`, `configurator` (required), `step`, `answers`, `config`
`list_units`, `set_project_units`	read, configure	The quantities and units Tangle converts between; the project's metric or imperial system and overrides.	`system`, `overrides`, `autoScale`
`set_project_controls`	configure	The project's default dashboard control style and highlight colours.	`controls`, `colors`
`get_variables`, `set_project_variables`	read, configure	Project variables with their values; declare the whole list.	`name`; `variables`
`list_project_templates`, `apply_project_template`	read, configure	The kinds of project the wizard starts from; add one to the project (rooms, planned devices, timelines, automations, dashboards, alert rules).	`templateId`, `answers`, `devices`, `acknowledged`, `dryRun`

Project library

TOOL	SCOPE	PURPOSE	ARGUMENTS
`list_projects`	read	The open project and every project kept on this core.	none
`create_project`	configure	A new, empty project; `open: true` opens it.	`name` (required), `id`, `open`
`open_project`, `close_project`	configure	Open another project, or close the open one. Refused while a timeline plays unless `force` (needs `live`).	`projectId`, `force`
`duplicate_project`	configure	Copy a project with its dashboards and plans; `open: true` is Save as.	`projectId`, `name`, `open`
`delete_project`	configure	Delete a project that is not open. Cannot be undone.	`projectId`

TOOL	SCOPE	PURPOSE	ARGUMENTS
`export_project`, `import_project`	read, configure	A `.tangle` file as base64, without device passwords; import one, or a `Project` object, as a new project.	`projectId`; `dataBase64` or `project`, `fileName`, `open`
`save_project`	configure	Write the open project's file now.	none

Control and live state

TOOL	SCOPE	PURPOSE	ARGUMENTS
`send_intent`	control	Make something happen (`power.on`, `level.set`, `input.select`, `scene.recall` with `params.scene`, ...) on a device, room, tag or everything. Waits for the result and returns the final `IntentStatus` with each device's outcome.	`intent` (an `IntentRequest`, required), `waitMs` (0 to 60000, default 5000)
`get_intent_status`	read	Current status of an intent sent earlier.	`intentId`
`control_timeline`	control	Play, pause, stop, seek (`positionMs`) or GO (cue lists). Returns the `TimelineState`.	`control`: `{ op, timelineId, positionMs }`
`get_timeline_states`	read	Transport state of every timeline (status, position, last and next cue).	none
`get_snapshot`	read	Live status and state of every device, and every timeline's state.	none
`get_device_state`	read	Live status and capability state of one device; use it to diagnose a device.	`deviceId`
`test_device`	configure	Test a device's connection in a separate module process (the running device is not touched). Pass `config` to try settings before saving them.	`deviceId` (required), `config`
`discover_devices`	configure	Scan the local network (mDNS, SSDP) for devices, with suggested modules. Refused while a timeline plays.	`seconds` (1 to 30, default 5)
`get_device_log`	read	A device's log, newest first: actions and who sent them, their outcome, status and state changes, events. Several devices are merged.	`deviceIds` (required), `limit`, `before`, `kinds`, `level`

TOOL	SCOPE	PURPOSE	ARGUMENTS
`add_device_log_note`, `clear_device_log`	control, configure	Write a note in a device's log; empty it.	`deviceId`, `text`, `level`
`set_variable`	control	Set a project variable; variable triggers fire.	`name`, `value`
`list_controls`	read	Dashboard controls with a kept value or override.	`id`
`set_control`	control	Set a dashboard control: value (or `add`), label, visible, enabled, other properties in `props`. Every screen follows.	`id` (required), `value`, `add`, `label`, `visible`, `enabled`, `props`
`press_control`	control	Report a control event as if someone used it: click, press, release or hold.	`id` (required), `event`, `value`
`rename_control`	configure	Rename a control; every reference follows.	`id`, `to`
`list_functions`, `call_function`	read, control	Library functions for expressions and Function steps; call one to try it.	`namespace`; `name`, `args`

Intents sent over MCP are recorded with source `api` and the token name. See REST API intents for intent names and targets.

Dashboards

TOOL	SCOPE	PURPOSE	ARGUMENTS
`get_dashboard_draft`	configure	Read a dashboard's draft, or its published version when there is no draft.	`dashboardId`
`save_dashboard_draft`	configure	Save a draft (contract `Dashboard`). Screens keep the published version until it is published.	`dashboardId`, `dashboard`
`publish_dashboard`	configure	Publish the draft: every screen showing the dashboard reloads.	`dashboardId`

TOOL	SCOPE	PURPOSE	ARGUMENTS
`list_dashboard_versions`	read	Published versions, newest first.	`dashboardId`
`get_dashboard_version`	read	One published version.	`dashboardId`, `version`
`restore_dashboard_version`	configure	Publish an old version again, as a new version.	`dashboardId`, `version`

`create_dashboard` and `update_dashboard` publish directly. See Dashboards.

Automations

TOOL	SCOPE	PURPOSE	ARGUMENTS
`run_automation`	control	Run an automation now. Conditions are checked unless `force` is true. Returns the run: started, or skipped with the reason.	`automationId` (required), `force`
`get_automation_runs`	read	Recent runs, newest first: what triggered each, and whether it completed, failed (with the reason) or was skipped.	`automationId`
`stop_automation`	control	Stop every run of an automation in progress.	`automationId`
`run_macro`, `get_macro_runs`, `stop_macro`	control, read	Deprecated aliases: macros are automations since contract 0.2.30.	`macroId`

Listeners

TOOL	SCOPE	PURPOSE	ARGUMENTS
`get_listeners`	read	Every listener and how it is doing; with `listenerId` and `log: true`, the messages it received with the matched rule, captures and runs.	`listenerId`, `log`, `since`
`get_listener_runs`	read	Recent runs started by a listener's messages.	`listenerId`
`test_listener`	configure	Check a message against a listener's rules. A dry run unless `run: true`.	`listenerId` (required), `text`, `http`, `osc`, `run`

Device creator and community

TOOL	SCOPE	PURPOSE	ARGUMENTS
`list_drivers`, `get_driver`, `get_driver_files`	read	Drivers made with the device creator: the list, one definition and code, or every file of its module.	`driverId`
`check_driver`	configure	Check a definition and its code without saving: errors and warnings with paths.	`definition` (required), `script`
`save_driver`	configure	Build and install a driver's module; devices using it restart. The description explains the definition format.	`definition` (required), `script`
`delete_driver`	configure	Delete a driver's source; its module keeps working.	`driverId`
`community`	read, configure	The community driver repository: `status`, `search`, `install`, `publish`, sign-in and sign-out.	`action` (required), and the action's arguments

Cluster

TOOL	SCOPE	PURPOSE	ARGUMENTS
`list_discovered_cores`	read	Other cores on the local network, not in this cluster.	none
`accept_discovered_core`	admin	Keep a discovered core separate, or pair it as a backup with its pairing code (ask the user for it).	`instanceId`, `mode` (required), `pairingCode`
`get_pairing_code`	admin	This core's pairing code.	none
`get_cluster_placement`	read	Shared mode: which node runs each device.	none
`rebalance_cluster`	control	Shared mode: spread the devices again. Ask the user first during a show; `dryRun` previews.	`dryRun`
`set_cluster_settings`	configure	Standby or shared mode, folder placement, auto return, the cluster address, health limits.	`mode`, `folders`, `autoReturn`, `address`, `limits`

Maps, plans and images

TOOL	SCOPE	PURPOSE	ARGUMENTS
`import_plan`, `get_plan`	configure, read	Import a floor plan (SVG text, or a DXF, DWG, PNG or JPEG file as base64) for a map; read one back.	`name`, `svg` or `dataBase64`; `file`
`import_image`, `get_image`	configure, read	Store a dashboard image; read one back.	`name`, `svg` or `dataBase64`; `file`
`get_map_positions`	read	What is where on a map, in metres: placed devices and tracked positions.	`mapId`
`push_positions`	control	Report positions as a tracking source does.	`positions`

Lighting

TOOL	SCOPE	PURPOSE	ARGUMENTS
`get_lighting_state`	read	Every sequence, the grand master and your programmer.	none
`lighting_command`	control	Go, back, goto, off, fader, flash, pause, grand master.	`command`
`lighting_programmer`	control	Select lights, set values, fan, blind, highlight, clear.	`update`
`lighting_store`	configure	Store the programmer into a cue, a preset or a group.	`store`
`search_fixture_library`	read	Search the bundled fixture library by manufacturer or model.	`query`, `limit`

Monitoring (Sentinel)

TOOL	SCOPE	PURPOSE	ARGUMENTS
`get_sentinel_status`	read	Overview: firing alert counts by severity, the firing alerts, the health of every device, core node and probe.	none
`list_alerts`	read	Pending and firing alerts; with `history: true` also recent transitions, newest first.	`history`, `limit` (1 to 500)
`ack_alert`	control	Acknowledge an alert: stops its repeat notifications.	`alertId` (required), `note`

TOOL	SCOPE	PURPOSE	ARGUMENTS
`silence_alerts`	control	Stop notifications for one alert or every alert of a rule until a time.	`alertId` or `ruleId`, `until` (ms since epoch, required)
`push_metrics`	control	Record custom metric values that metric alert rules can watch.	`samples`: `[{ name, value, labels, at }]`
`get_metrics`	read	Current custom and probe metric values; `samples: true` adds 24 h of per-minute min/max/last.	`samples`
`test_channel`	configure	Send a test notification through a channel and return the provider's answer.	`channelId`

Monitoring is part of Tangle Pro and Ultimate. See Sentinel.

License

TOOL	SCOPE	PURPOSE	ARGUMENTS
`get_license`	read	State, mode (licensed, designer, demo), edition, features, limits, expiry dates, and whether a designer or demo session is paused.	none
`resume_session`	control	Resume device connections and timelines after the hourly pause of the free designer or demo mode.	none
`manage_license`	admin	Activate, refresh, deactivate, import a license file, offline mode, sign in with email, password reset, request file.	`action` (required), `key`, `email`, `password`, `licenseId`, `name`, `file`, `offline`

Resources

URI	CONTENT
`tangle://project`	The project configuration.
`tangle://snapshot`	Every device's status and state, and every timeline's state.
`tangle://devices/{deviceId}/state`	Live status and state of one device (one resource per device in the project).

URI	CONTENT
`tangle://timelines/{timelineId}/state`	Transport state of one timeline (one resource per timeline).

All resources are `application/json`. The two `{...}` forms are also offered as resource templates.

Assistants can subscribe to resources to be notified when they change:

CHANGE	RESOURCES NOTIFIED
A device's state or status	`tangle://devices/<id>/state`, `tangle://snapshot`
A timeline's state	`tangle://timelines/<id>/state`, `tangle://snapshot`
The project	`tangle://project`

Prompts

PROMPT	ARGUMENT	WHAT IT ASKS THE ASSISTANT TO DO
`set_up_room`	`room` (room name)	List the installed modules, ask which devices are in the room and how to reach them, create the room and each device (dry run first), propose a "presentation" and an "off" scene, then test with `scene.recall` and report each device's outcome.
`build_cue_list`	`show` (what happens, in order)	Read the project for real device ids, create a `cuelist` timeline with one track and one cue per step, validate with a dry run, show the cue list, and apply it only after you agree.
`diagnose_device`	`deviceId`	Read the device's status and configuration, check its module is installed, and explain the likely cause and what to check on site, without changing anything.

Audit

Every change and control action made through MCP is recorded in `audit.log` (in the Tangle log folder) with surface `mcp` and the token name, like any other client. Use one token per assistant so the log shows who did what.

REFERENCE

Command line

Every command and option of tangle-core, tangle and tangle-sim, the environment variables they read, and where they keep their files.

Tangle has three command-line programs: `tangle-core` runs projects and manages tokens, licenses and the boot service; `tangle` installs device modules and edits the devices of a project file; `tangle-sim` runs virtual devices for testing. Each one prints its help with `--help` and its version with `--version`.

---

tangle-core

```sh
tangle-core <command> [options]
```

COMMAND	WHAT IT DOES
`run`	Run a project: start its device modules and serve clients.
`check`	Check a project file and exit.
`modules`	List the device modules found.
`token`	Create, list and revoke API tokens.
`license`	Activate, check or move the license of this computer.
`service`	Install or remove the boot service.
`cluster`	The cluster this computer joined by pairing.
`community`	The community driver repository.
`projects`	The project library: the projects kept on this computer.
`mcp`	MCP over stdio for desktop assistants, connected to a running tangle-core.

On error, every command prints `error: <message>` and exits with code 1.

tangle-core run

```sh
tangle-core run [--project show.yaml] [options]
```

Loads the project (without `--project`: the project open last, from the library, or a new one), starts the modules of its devices, and serves the WebSocket, REST API, MCP endpoint, designer and dashboards on one port. It stops cleanly on Ctrl+C (and SIGTERM on macOS and Linux): modules get a shutdown request and core exits within about 5 seconds.

OPTION	DEFAULT	DESCRIPTION
`-p`, `--project <file>`	the library	Project file (`.json`, `.yaml` or `.yml`), edited in place: changes made through the API are saved back to it. Without it, core runs the project open last from its library, or a new one.
`-m`, `--modules <folder>`		Extra folder of modules (one sub-folder per module). Repeatable. The installed modules folder is always searched first; later folders win on the same module id.
`--listen <addr:port>`	`127.0.0.1:7340`	Address to serve clients on.
`--allow-remote`	off	Allow listening on a network address, so other machines can connect.
`--insecure-http`	off	Accept unencrypted remote connections. Required with a network address until TLS is available: trusted networks only.
`--node-id <id>`	`core-1`	Name of this node, shown to clients. With `--peer`, required and unique per node.
`--catalog <folder>`	`modules` next to the program	Folder of modules offered for install in the designer (one sub-folder per module), such as the modules bundled with the installer. Repeatable. Also read from `TANGLE_MODULE_CATALOG`.
`--device-creator <folder>`	`device-creator` next to the program	The device creator's builder. Also read from `TANGLE_DEVICE_CREATOR`.
`--dwg-converter <program>`	ODA File Converter, `ODAFileConverter` or `dwg2dxf`	Converts DWG map plans to DXF. Also read from `TANGLE_DWG_CONVERTER`.
`--ui <folder>`	`ui` next to the program	Folder of the built web apps (dashboards, and the designer in its `designer/` sub-folder). Also read from `TANGLE_UI`.
`--demo`	off	Without a license, run in demo mode (every feature, up to 10 devices, sessions of one hour) instead of the free designer (this computer only, pauses every hour).
`--log-json`	off	Write console logs as JSON lines.

OPTION	DEFAULT	DESCRIPTION
`--log-dir <folder>`	platform log folder	Where log files go.

Redundancy options (all need `--peer`):

OPTION	DEFAULT	DESCRIPTION
`--peer <host:port>`		Another node's redundancy address. Repeat it, or separate addresses with commas, for each other node. Turns redundancy on.
`--role primary|backup`	`primary`	`primary`: this node goes first when choosing a primary. Others follow by node id.
`--cluster-listen <addr:port>`	`0.0.0.0:7341`	Address for heartbeats (UDP) and state mirroring (TCP).
`--cluster-key <secret>`	`TANGLE_CLUSTER_KEY`	Shared secret, identical on every node, at least 16 characters.
`--cluster-key-file <file>`		File holding the shared secret (what installed services use). Cannot be combined with `--cluster-key`.
`--no-project-sync`	off	Do not copy the primary's project to the backups (update them by hand).

`--no-discovery` (no `--peer` needed) stops this core from announcing itself on the local network and from listing discovered cores. A core that was paired from the designer rejoins its cluster from the saved `cluster.json` without any of these options. See Redundancy.

Examples:

```sh
# This machine only
tangle-core run -p show.yaml

# Accept other machines on a trusted network
tangle-core run -p show.yaml --listen 0.0.0.0:7340 --allow-remote --insecure-http

# Extra module folder
tangle-core run -p show.yaml -m ./my-modules

# Redundant pair (run the matching command on 10.0.0.2 with --role backup)
export TANGLE_CLUSTER_KEY=<same secret on every node>
tangle-core run -p show.yaml --node-id core-a --peer 10.0.0.2:7341
```

Notes:

- Core refuses to listen on a network address without `--allow-remote`, and without `--insecure-http` (no TLS yet).
- The license is checked once, at start. If the project has more devices, DMX universes or timelines than the edition allows, `run` stops with a message naming the limit. In the free designer mode, remote access is turned off. See Licensing.
- If a `tangle.lock` file sits next to the project, the locked module versions are made active at start (see `tangle module lock`).
- Logs go to the console (stderr) and to a daily JSON log file `tangle-core.<date>.log` in the log folder; 14 files are kept. Set `TANGLE_LOG` to change the level.
- See Redundancy for how the cluster chooses its primary.

tangle-core check

```sh
tangle-core check --project show.yaml
```

Validates the file against the contract schema and its cross-references (a scene pointing at a missing device, a device in a missing room...). On success it prints a summary and exits with 0:

```text
show.yaml: valid (2 rooms, 8 devices, 3 scenes, 1 timelines)
```

On failure it lists each problem and exits with 1.

OPTION	DESCRIPTION
`-p`, `--project <file>`	Project file (required).

tangle-core modules

```sh
tangle-core modules [-m <folder>]...
```

Lists the modules found in the installed modules folder and in the extra folders: id, version and folder. Folders that are not valid modules are reported as `skipped: ...` on stderr.

OPTION	DESCRIPTION
`-m`, `--modules <folder>`	Extra folder of modules. Repeatable.

tangle-core token

API tokens authenticate REST, MCP and WebSocket clients from other machines. See REST API.

```sh
tangle-core token create --name stage-ipad --scopes read,control
tangle-core token list
tangle-core token revoke stage-ipad
```

COMMAND	OPTIONS	DESCRIPTION
`token create`	`--name <name>` (required), `--scopes <list>` (default `read,control`)	Create a token and print its secret once. Scopes, comma-separated: `read`, `control`, `configure`, `admin`, `live`. Names must be unique among active tokens.
`token list`		Id, name, scopes and state (`active` or `revoked`) of every token. Secrets are never shown.
`token revoke <name-or-id>`		Revoke a token. It stops working at once, even on a running core.

Tokens are stored hashed in `tokens.json` in the data folder. A core installed as a boot service uses the machine-wide data folder: manage its tokens with `sudo tangle-core token ...` on macOS and Linux.

tangle-core license

```sh
tangle-core license <command>
```

COMMAND	OPTIONS	DESCRIPTION
`activate <key>`	`--name <name>`	Activate this computer. The first host creates the installation; the next ones join it as redundant hosts. `--name` defaults to this computer's name.
`status`	`--json`	Show the license, edition, limits and what the next start will run as. `--json` prints `{ status, nextStart }`.
`refresh`		Fetch a fresh license file now (renewals, new limits).
`deactivate`		Remove this computer from its installation. It runs as the free designer (or `--demo`) from its next start.
`move <key>`	`--name <name>`	Move a floating license to this computer (once every 7 days).
`fingerprint`		Print this computer's id, for offline activation.

COMMAND	OPTIONS	DESCRIPTION
`issue <key>`	`--fingerprint <id>` (required), `--name <name>`, `-o`, `--out <file>` (required)	Offline activation, step 1, on a computer with internet: write the license file for the offline computer whose fingerprint is given.
`import <file>`		Offline activation, step 2, on the offline computer: install the license file. Also installs the answer to a license request.
`sign-in <email>`	`--license <id>`, `--name <name>`	Sign in with the license owner's email and activate their license. The password comes from `TANGLE_LICENSE_PASSWORD` or is typed on standard input. With several licenses and no `--license`, their ids are listed.
`request`	`--key <key>` or `--email <email>`, `--name <name>`, `-o`, `--out <file>` (required)	Write a license request for this computer, to email to licensing@tangle.systems. The answer is a license file for `import`.

A license change applies at the next start of tangle-core. See Licensing.

tangle-core service

Runs tangle-core at boot, restarts it after a crash, and stops it cleanly.

```sh
sudo tangle-core service install -- --project /shows/main.yaml --node-id core-a   # macOS, Linux
tangle-core service install -- --project C:\Shows\main.yaml                        # Windows, administrator terminal
sudo tangle-core service uninstall
```

COMMAND	OPTIONS	DESCRIPTION
`service install [--user] -- <run options>`	`--user`	Install and start the service. Everything after `--` is passed to `run`. With `--user` (macOS, Linux) it runs at login for the current user instead of at boot.
`service uninstall [--user]`	`--user`	Stop and remove the service.

What `install` does:

- Checks the `run` options and loads the project first, so mistakes show now and not at boot.
- Makes paths absolute (`--project`, `--modules`, `--log-dir`, `--cluster-key-file`, `--ui`).

- Keeps `PATH`, `TANGLE_HOME`, `TANGLE_PYTHON`, `TANGLE_NODE` and `TANGLE_LOG` from your shell, so `node` and `python3` are found.
- With `--peer`, writes the cluster key (from `TANGLE_CLUSTER_KEY` or `--cluster-key`) to `cluster.key` in the data folder, readable only by the service, and passes `--cluster-key-file`. The key never appears in the service definition.

	MACOS	WINDOWS	LINUX
Service	launchd daemon `/Library/LaunchDaemons/tangle-core.plist` (`--user`: `~/Library/LaunchAgents`)	Windows Service `tangle-core` ("Tangle core"), automatic start	systemd unit `/etc/systemd/system/tangle-core.service` (`--user`: `~/.config/systemd/user`)
Needs	`sudo` (not with `--user`)	Administrator terminal; `--user` is not available	`sudo` (not with `--user`)

See Operations.

tangle-core cluster

```sh
tangle-core cluster show    # this machine's saved cluster: id, node id, members
tangle-core cluster leave
```

`leave` forgets this machine's saved cluster membership and key (`cluster.json` in the data folder), so the next start runs standalone. Stop core first. The other members are not changed. Run it with `sudo` for a core installed as a boot service.

tangle-core projects

```sh
tangle-core projects list               # the open project is marked *
tangle-core projects import show.tangle # or a project .json / .yaml
```

The library keeps each project in `projects/<id>/project.json` in the core data folder, with its dashboards and plans. The designer's project menu creates, opens, duplicates, exports and deletes them.

tangle-core community

```sh
tangle-core community status            # which server, who is signed in
tangle-core community search [words] [--mine]
```

Signing in and publishing are done in the designer (**Settings > Community**, **Publish to community**). `TANGLE_COMMUNITY_URL` points core at another community server. See Device creator.

tangle-core mcp

```sh
tangle-core mcp [--url <url>] [--token <token>]
```

Bridges MCP over stdio to a running tangle-core, for desktop assistants.

OPTION	DEFAULT	DESCRIPTION
`--url <url>`	`http://127.0.0.1:7340`	Address of the running tangle-core.
`--token <token>`	`TANGLE_TOKEN`	API token.

See MCP.

---

tangle

```sh
tangle <command> [options]
```

`tangle` manages device modules and devices on this machine. Device commands edit a project file. While tangle-core runs that project they refuse, because the running core would not see the change: use its REST API, MCP or the designer then.

COMMAND	WHAT IT DOES
`module`	Install, update, roll back and lock device modules.
`device`	Add, test and bulk-edit the devices of a project file.
`discover`	Scan the local network for devices and suggest modules.

tangle module

COMMAND	OPTIONS	DESCRIPTION
`module add <folder>`	`-y`, `--yes`	Install (or update) a module from its folder. Asks before granting the permissions it requests; `--yes` grants them without asking (needed when not run in a terminal). Reinstalling a version that a running project uses is refused.
`module list`	`--json`	Installed modules: id, version, channel. `--json` adds source, previous version and folder.

COMMAND	OPTIONS	DESCRIPTION
`module remove <id>`		Remove a module and all its versions. Refused while a running project uses it.
`module rollback <id>`		Make the previously active version active again.
`module lock`	`-p`, `--project <file>` (required)	Write `tangle.lock` next to the project with the module versions it uses.
`module sync`	`-p`, `--project <file>` (required)	Make the versions in the project's `tangle.lock` active. Reports modules that are not installed or whose files differ, and exits with 1 if any.

Each module keeps its active and previous versions. Module ids look like `misc.pjlink.projector`. See Devices and Module SDK.

```sh
tangle module add ./misc.pjlink.projector --yes
tangle module lock -p show.yaml          # on the programming machine
tangle module sync -p show.yaml          # on the show machine
```

tangle device

All device commands take `-p`, `--project <file>` (required), except `secret`.

COMMAND	OPTIONS	DESCRIPTION
`device list`		Id, name, module and room of each device.
`device add <id>`	`-m`, `--module <id>` (required), `--name <name>` (default: the id), `--room <room>`, `--set KEY=VALUE` (repeatable)	Add one device. Settings are typed from the module's settings schema (numbers, booleans, objects and arrays as JSON). Secret settings go to the credential store. The whole project is validated before saving.
`device remove <id>`		Remove a device.
`device test <id>`		Start the device's module on its own and check that the device answers (10 s at most). Prints its state on success, or the status and detail on failure (exit code 1).

COMMAND	OPTIONS	DESCRIPTION
`device import <file>`	`--skip-invalid`, `--dry-run`	Add devices from a `.csv` or `.json` file. Nothing is added if a row is invalid, unless `--skip-invalid`. `--dry-run` checks only.
`device export`	`--format csv|json` (default `csv`), `-o`, `--output <file>`	Write the devices as CSV or JSON, to stdout without `--output`.
`device secret <device> <setting>`	`--delete`	Store a secret setting (a password) for a device on this machine. The value is read without echo, or from stdin when piped. `--delete` removes it.

CSV columns: `id`, `name`, `moduleId`, `room`, `tags` (separated by `;`), and one `config.<setting>` column per setting. Cells are typed from the module's settings schema; errors give the line number.

```sh
tangle device add -p show.yaml pj-1 -m misc.pjlink.projector --room hall --set host=10.0.0.5
--set password=admin
tangle device import -p show.yaml displays.csv
tangle device export -p show.yaml --format csv -o displays.csv
tangle device test -p show.yaml pj-1
echo "s3cret" | tangle device secret pj-1 password
```

Secrets are stored in the OS credential store (Keychain on macOS, Credential Manager on Windows, Secret Service on Linux) under service `tangle`. Where no store answers, they go to `secrets.json` in the data folder, readable by its owner only.

tangle discover

```sh
tangle discover [--seconds 5] [--json]
```

Scans the local network with mDNS, SSDP and the module searches (vendor broadcasts) at the same time, and suggests installed modules for each device found.

OPTION	DEFAULT	DESCRIPTION
`--seconds <n>`	`5`	How long to listen (1 to 30).
`--json`	off	Print the results as JSON.

Refused while tangle-core runs a project on this machine: scan through the core then (`POST /api/v1/discovery`), which refuses only while a timeline plays.

tangle-sim

```sh
tangle-sim <command> [options]
```

`tangle-sim` runs virtual devices that speak their real protocols on real ports, so Tangle's modules connect to them as to the real devices. It needs Node.js 20 or newer (Node 22.6 or newer when run from source). See Simulator.

Run devices

COMMAND	DESCRIPTION
`tangle-sim run [config.json]`	Start the devices. Default config `tangle-sim.json`; without one, it starts with no devices and creates the file when you add the first device in the web UI. `run` is the default command.
`tangle-sim init [file]`	Write an example config (default `tangle-sim.json`): two PJLink projectors, a video matrix (ASCII over TCP), a DMX receiver and three scenarios. Refuses to overwrite a file.
`tangle-sim types`	List the device types and their options.

`run` options:

OPTION	DEFAULT	DESCRIPTION
`--ui-host <ip>`	`127.0.0.1`	Address of the control API and web UI.
`--ui-port <n>`	`4380`	Port of the control API and web UI.
`--no-mdns`		Do not advertise devices over mDNS.
`--no-ssdp`		Do not advertise devices over SSDP.
`-q`, `--quiet`		Do not print protocol lines.

Device types: `pjlink` (PJLink projector, TCP 4352), `ascii` (any line-based device over TCP or UDP, port 5000, behaviour from rules) and `dmx` (Art-Net receiver on UDP 6454 and sACN on UDP 5568).

Control a running simulator

These commands talk to a running simulator at `--url` (default `http://127.0.0.1:4380`, or `TANGLE_SIM_URL`).

COMMAND	DESCRIPTION
`tangle-sim status`	Devices, ports, status and faults.
`tangle-sim show <device>`	Full state, faults and the last 20 log lines.
`tangle-sim set <device> key=value...`	Change state, for example `set pj-1 lampHours=1990 power=on`. Values are JSON when they parse, text otherwise.
`tangle-sim fault <device> key=value...`	Set faults: `offline`, `silent`, `fail`, `rejectAuth` (`on` or `off`; a bare name means on), `latency=<ms>`.
`tangle-sim clear <device>`	Remove every fault.
`tangle-sim drop <device>`	Cut its connections.
`tangle-sim reboot <device> [seconds]`	Take it offline for N seconds (default 5); it comes back with its initial state.
`tangle-sim add <type> <id> [key=value...]`	Add a device, for example `add pjlink pj-3 name="Lobby PJ"`. Saved to the config file.
`tangle-sim remove <device>`	Remove a device. Saved to the config file.
`tangle-sim scenario [name] [stop]`	List scenarios, or start or stop one.

```sh
tangle-sim init
tangle-sim run
tangle-sim fault pj-1 offline
tangle-sim clear pj-1
```

Environment variables

VARIABLE	READ BY	DESCRIPTION
`TANGLE_HOME`	tangle-core, tangle	Puts every Tangle folder (modules, data, tokens, logs in `logs/`) in this one folder. Handy for portable installs.
`TANGLE_LOG`	tangle-core	Log filter, for example `debug` or `warn`. Default `info`.
`TANGLE_PYTHON`	tangle-core, tangle	Python interpreter for Python modules, for example a virtualenv's `python`. Default `python3` (`python` on Windows). A module with its own virtual environment uses that one.

VARIABLE	READ BY	DESCRIPTION
`TANGLE_NODE`	tangle-core, tangle	Node.js executable for JavaScript modules. Default `node`.
`TANGLE_NATIVE_DIR`	tangle-core, tangle	Extra folders where native module executables are looked for (path list).
`TANGLE_SECRETS`	tangle-core, tangle	`file` keeps device secrets in `secrets.json` in the data folder instead of the OS credential store.
`TANGLE_TOKEN`	`tangle-core mcp`	API token for the MCP bridge (instead of `--token`).
`TANGLE_CLUSTER_KEY`	`tangle-core run`, `tangle-core service install`	Redundancy shared secret (instead of `--cluster-key`), at least 16 characters, the same on every node.
`TANGLE_UI`	`tangle-core run`	Folder of the built web apps (instead of `--ui`).
`TANGLE_MODULE_CATALOG`	`tangle-core run`	Folder of modules offered for install (instead of `--catalog`).
`TANGLE_DEVICE_CREATOR`	`tangle-core run`	Folder of the device creator's builder (instead of `--device-creator`).
`TANGLE_DWG_CONVERTER`	`tangle-core run`	Program that converts DWG plans to DXF (instead of `--dwg-converter`).
`TANGLE_COMMUNITY_URL`	tangle-core	Address of another community driver server.
`TANGLE_LICENSE_PASSWORD`	`tangle-core license sign-in`	The license owner's password, instead of typing it.
`TANGLE_SIM_URL`	`tangle-sim`	Address of the running simulator for control commands (instead of `--url`).

Files and folders

	MACOS	WINDOWS	LINUX
Data folder	`~/Library/Application Support/Tangle`	`%ProgramData%\Tangle`	`~/.local/share/tangle`
Log folder	`~/Library/Logs/Tangle`	`%ProgramData%\Tangle\logs`	`~/.local/state/tangle/logs`

	MACOS	WINDOWS	LINUX
Data folder, as root (boot service)	`/Library/Application Support/Tangle`	same as above	`/var/lib/tangle`
Log folder, as root (boot service)	`/Library/Logs/Tangle`	same as above	`/var/log/tangle`

`TANGLE_HOME` replaces all of them: data in that folder, logs in its `logs` sub-folder.

IN THE DATA FOLDER	WHAT IT HOLDS
`modules/`	Installed modules, one folder per module id with its active and previous versions.
`module-data/`	Writable data of each module.
`tokens.json`	API tokens (hashes only).
`secrets.json`	Device and channel secrets, when no OS credential store answers.
`license/`	The license of this computer.
`cluster.key`	Redundancy key written by `service install`.
`cluster.json`	Cluster membership and key saved at pairing (mode 600).
`dashboards/<project>/`	Dashboard drafts and published versions.
`sentinel/<project>/`	Monitoring alert history (`history.jsonl`).

IN THE LOG FOLDER	WHAT IT HOLDS
`tangle-core.<date>.log`	Daily JSON log, 14 days kept.
`audit.log`	One JSON line per configuration change or control action: time, surface (`rest`, `mcp`, `ws`, `cli`), caller, action, target, outcome.

REFERENCE

Project file

The Tangle project file format in JSON or YAML - rooms, devices, scenes, timelines, dashboards, automations, variables, listeners and Sentinel monitoring.

A Tangle project is one file that holds everything tangle-core runs: rooms, devices, scenes, timelines, dashboards, automations and monitoring. The designer, the REST API, MCP and the `tangle` command all edit this file; you can also write it by hand. This page describes every section and field.

Format and loading

- The file is JSON (`.json`) or YAML (`.yaml`, `.yml`). The format follows the extension. Changes made through the API are saved back in the same format.
- Field names are camelCase. Unknown fields are refused.
- Ids (`id`, `deviceId`, `room` and other references) are lowercase: letters, digits, `.`, `_` and `-`, starting with a letter or digit, at most 128 characters. For example `room-a`, `room-a.projector-1`.
- Timestamps are milliseconds since the Unix epoch (UTC). Times of day are `"HH:MM"` (24 h) in the core machine's local time. Weekdays are `mon`, `tue`, `wed`, `thu`, `fri`, `sat`, `sun`.

Core checks a project in two steps: first against the contract JSON Schema (`Project`), then the cross-references the schema cannot express. Every error names the place to fix. Check a file without running it:

```sh
tangle-core check --project show.yaml
```

```text
error: project is invalid:
  scene presentation action 2: unknown device light-3
```

Cross-reference checks:

- Ids are unique within rooms, devices, scenes, timelines, automations, alert rules, notification channels, maintenance windows and probes.
- A device's `room` exists.
- Scene actions point at existing devices.
- Cues point at existing devices (action cues) or existing rooms and devices (intent cue targets).
- Automation triggers, conditions and actions point at existing devices, timelines and cues; a `scene.recall` intent names an existing scene; a cue trigger has a `cueId`.
- Alert rules point at existing devices, rooms and notification channels; a `stale` condition has `capability` and `key`, or `metric`.

- Maintenance windows point at existing alert rules, devices and rooms, and have `start` (once) or `from` and `to` (weekly).
- A probe's `room` exists.

The license also limits what a project may contain (devices, DMX universes, timelines). See Licensing.

Top level

FIELD	TYPE	REQUIRED	DESCRIPTION
`id`	id	yes	Project id. Also names the project's dashboard and monitoring folders.
`name`	string	yes	Project name.
`contract`	string	no	Contract version the project was written for, for example `"0.2.43"`.
`rooms`	array of Room	no	Rooms and areas.
`devices`	array of Device	no	Controlled devices.
`scenes`	array of Scene	no	Named sets of device actions.
`timelines`	array of Timeline	no	Timecode shows and cue lists.
`dashboards`	array of Dashboard	no	Control surfaces.
`automations`	array of Automation	no	Step lists run by triggers, buttons, listeners or by hand.
`variables`	array of project variables	no	Values core keeps for the whole project: `name`, optional `label`, `description`, `default`, `persist` (default true). See Automations.
`listeners`	array of listeners	no	Ports other systems send messages to. See Listeners.
`maps`	array of maps	no	Floor plans with devices placed on them.
`views`	array of views	no	End-user views. See End-user views.
`lighting`	object	no	Lighting engine: groups, presets, sequences and executors.

FIELD	TYPE	REQUIRED	DESCRIPTION
`cluster`	object	no	How a cluster shares the work: `mode` (`standby` or `shared`), folder placement, node health limits. See Redundancy.
`units`	object	no	`system` (`metric` or `imperial`) and per-quantity overrides for how values are shown.
`controls`, `controlColors`		no	The default dashboard control style and the highlight colours of each style.
`macros`	array	no	Deprecated since contract 0.2.30: read and moved into `automations` when the project loads.
`alertRules`	array of AlertRule	no	Monitoring rules.
`notificationChannels`	array of NotificationChannel	no	Where alerts are sent.
`maintenanceWindows`	array of MaintenanceWindow	no	Periods when notifications are muted.
`probes`	array of Probe	no	Network checks for assets without a device module.

Complete example

A meeting room with a projector and two lights, a scene, a short cue list, a dashboard, an automation and a monitoring rule:

JSON

```
{
  "id": "demo",
  "name": "Demo building",
  "contract": "0.2.43",
  "rooms": [{ "id": "room-a", "name": "Room A" }],
  "devices": [
    { "id": "proj-1", "name": "Projector", "moduleId": "misc.pjlink.projector", "room": "room-
a", "config": { "host": "10.0.0.20" } },
    { "id": "light-1", "name": "Ceiling light", "moduleId": "tangle.virtual-light-js", "room":
"room-a" },
    { "id": "light-2", "name": "Wall light", "moduleId": "tangle.virtual-light-py", "room":
"room-a", "tags": ["wall"] }
  ],
  "scenes": [
    {
      "id": "presentation",
      "name": "Presentation",
      "actions": [
        { "deviceId": "proj-1", "capability": "tangle.power", "action": "on" },
        { "deviceId": "light-1", "capability": "tangle.level", "action": "set", "params": {
"level": 0.3 } },
        { "deviceId": "light-2", "capability": "tangle.power", "action": "off" }
      ]
    }
  ],
  "timelines": [
    {
      "id": "welcome",
      "name": "Welcome",
      "mode": "cuelist",
      "tracks": [
        {
          "id": "main",
          "name": "Main",
          "cues": [
            { "id": "c1", "name": "House to half", "atMs": 0, "payload": { "kind": "intent",
"intent": { "name": "level.set", "target": { "kind": "room", "id": "room-a" }, "params": {
"level": 0.5 } } } },
            { "id": "c2", "name": "Presentation", "atMs": 5000, "payload": { "kind": "intent",
"intent": { "name": "scene.recall", "target": { "kind": "room", "id": "room-a" }, "params": {
"scene": "presentation" } } } }
          ]
        }
      ]
    }
  ],
  "dashboards": [
    {
      "id": "room-a",
      "name": "Room A",
      "access": { "mode": "open" },
      "pages": [
        {
          "id": "main",
          "name": "Main",
```

```
          "widgets": [
            {
              "id": "light",
              "type": "toggle",
              "label": "Ceiling light",
              "props": { "value": { "kind": "state", "device": "light-1", "capability":
"tangle.power", "key": "on" } },
              "action": { "kind": "intent", "name": "power.on", "target": { "kind": "device",
"device": "light-1" } },
              "offAction": { "kind": "intent", "name": "power.off", "target": { "kind":
"device", "device": "light-1" } }
            }
          ],
          "layouts": { "tablet": [{ "widget": "light", "x": 0, "y": 0, "w": 3, "h": 2 }] }
        }
      ]
    }
  ],
  "automations": [
    {
      "id": "evening-off",
      "name": "Lights off at night",
      "triggers": [{ "kind": "time", "at": "22:00" }],
      "steps": [{ "kind": "intent", "intent": { "name": "power.off", "target": { "kind":
"room", "id": "room-a" } } }]
    }
  ],
  "alertRules": [
    { "id": "offline", "name": "Device offline", "severity": "critical", "condition": {
"kind": "status", "rooms": ["room-a"] }, "forSeconds": 30, "channels": ["techs"] }
  ],
  "notificationChannels": [
    { "id": "techs", "name": "Technicians", "kind": "webhook", "settings": { "url":
"https://ops.example.com/tangle" } }
  ],
  "maintenanceWindows": [{ "id": "sunday", "name": "Sunday", "days": ["sun"], "from": "02:00",
"to": "04:00" }],
  "probes": [{ "id": "switch-1", "name": "Core switch", "kind": "tcp", "host": "10.0.0.2",
"port": 22 }]
}
```

The same project start in YAML:

```yaml
id: demo
name: Demo building
rooms:
  - id: room-a
    name: Room A
devices:
  - id: proj-1
    name: Projector
    moduleId: misc.pjlink.projector
    room: room-a
    config:
      host: 10.0.0.20
scenes:
  - id: presentation
    name: Presentation
    actions:
      - deviceId: proj-1
        capability: tangle.power
        action: "on"
```

Note: in YAML, quote values such as `"on"`, `"off"`, `"yes"` and `"no"`, or YAML reads them as booleans.

Rooms

FIELD	TYPE	REQUIRED	DESCRIPTION
`id`	id	yes	Room id.
`name`	string	yes	Display name.
`tags`	array of strings	no	Free labels.

An intent with target `{ "kind": "room", "id": "room-a" }` reaches every device whose `room` is `room-a`.

Devices

Each device is one module plus one configuration (contract `DeviceInstance`).

FIELD	TYPE	REQUIRED	DESCRIPTION
`id`	id	yes	Device id.
`name`	string	yes	Display name.

FIELD	TYPE	REQUIRED	DESCRIPTION
`moduleId`	id	yes	The module that drives it, for example `misc.pjlink.projector`.
`config`	object	no	Settings, checked against the module's settings schema (`configSchema`) when changed through the API or `tangle`.
`room`	id	no	Room it belongs to.
`tags`	array of strings	no	Labels; intents can target a tag (`{ "kind": "tag", "tag": "wall" }`).
`node`	id	no	Redundancy: the core node its hardware is attached to (serial port, USB DMX, LTC output). Only that node runs its module. Leave it out for networked devices. See Redundancy.
`runsOn`	id	no	Shared mode: the node a networked device prefers. It moves to another node while this one is down.
`folder`	string	no	Folder in the device list, `/`-separated (`Stage/Truss 1`).
`planned`	boolean	no	`true`: a placeholder whose address may not be known yet. Core does not start it.
`parent`	id	no	The device this one is a sub-device of (a light under its DMX output).
`setup`	object	no	Progress of the device's setup assistant. Never holds secrets.

- Secret settings (marked `x-secret` in the module) are not stored in the project file: they go to the OS credential store when set through the designer, the API or `tangle device add`. Set one by hand with `tangle device secret <device> <setting>`.
- A device whose module is not installed is accepted; its status shows the problem.
- DMX modules put the universe a device uses in a `universe` setting; licensing counts distinct universes.

Every device reports a status: `connecting`, `online`, `degraded`, `offline`, `timeout` or `error`. See Devices.

Scenes

A scene is recalled with the `scene.recall` intent (`params.scene`). Only the actions for devices inside the intent's target run.

FIELD	TYPE	REQUIRED	DESCRIPTION
`id`	id	yes	Scene id.
`name`	string	yes	Display name.
`actions`	array of scene actions	yes	What to do.

Scene action:

FIELD	TYPE	REQUIRED	DESCRIPTION
`deviceId`	id	yes	Device.
`capability`	id	yes	Capability, for example `tangle.power`, `tangle.level`.
`action`	string	yes	Action of that capability, for example `on`, `set`.
`params`	object	no	Action parameters, for example `{ "level": 0.3 }`.

Timelines

FIELD	TYPE	REQUIRED	DESCRIPTION
`id`	id	yes	Timeline id.
`name`	string	yes	Display name.
`mode`	`timecode` | `cuelist`	yes	`timecode`: runs freely and fires cues at their time. `cuelist`: waits for GO at each cue.
`tracks`	array of tracks	yes	Tracks of cues.
`clock`	clock	no	Time source. Default internal. See Clocks.
`frameRate`	`24` | `25` | `29.97df` | `30` | `50` | `60`	no	Frame rate (a string).
`durationMs`	number > 0	no	Length. At the end the timeline stops, or starts again when `loop` is true. Without it, the timeline keeps playing after its last cue until stopped.
`loop`	boolean	no	Start again at the end.

FIELD	TYPE	REQUIRED	DESCRIPTION
`timecodeOutputs`	array	no	Devices that generate timecode following this timeline: `[{ "deviceId", "offset" }]`. `offset` is the timecode sent at position 0 (default `00:00:00:00`).

Tracks

FIELD	TYPE	REQUIRED	DESCRIPTION
`id`	id	yes	Track id (unique in its timeline).
`name`	string	yes	Display name.
`cues`	array of cues	yes	Cues.
`muted`	boolean	no	A muted track fires no cues.

Cues

FIELD	TYPE	REQUIRED	DESCRIPTION
`id`	id	yes	Cue id.
`atMs`	number >= 0	yes	Position from the timeline start, in ms.
`payload`	object	yes	What the cue does (below).
`name`	string	no	Display name.
`preRollMs`	number >= 0	no	Send this many ms early to absorb device latency.
`enabled`	boolean	no	Default true. A disabled cue does not fire.

Payloads:

KIND	FIELDS	DESCRIPTION
`intent`	`intent`: an intent request (`name`, `target`, `params`, `priority`)	Sends an intent, for example `scene.recall`.
`action`	`deviceId`, `capability`, `action`, `params`	Runs one device action directly.

Clocks

KIND	FIELDS	STATUS
`internal`	none	The default: core's own clock.
`timecode`	`deviceId` (required), `offset` (timecode at position 0, default `00:00:00:00`), `freewheelMs` (0 to 10000, default 500)	Chases a timecode input device (any device with the `tangle.timecode` capability: LTC, MTC, Art-Net timecode).
`ltc`	`input`	Older form, not played: use `timecode` with an LTC input device.
`mtc`	`port`	Older form, not played: use `timecode` with an MTC input device.
`ptp`	`deviceId` (PTP clock reference device), `domain` (0 to 127, default 0, used when `deviceId` is not set)	Plays like `internal`, at the rate of the PTP grandmaster.
`wallclock`	`timezone` (IANA zone, empty: the computer's zone)	The position is the time of day; restarts from 0 at midnight.

Timecode values are `HH:MM:SS:FF`, with `;` before the frames for drop-frame. See Timelines.

Dashboards

The project holds the published version of each dashboard. Drafts and older versions are kept by tangle-core outside the project file. See Dashboards.

FIELD	TYPE	REQUIRED	DESCRIPTION
`id`	id	yes	Dashboard id; opened at `/d/<id>`.
`name`	string	yes	Display name.
`pages`	array of pages	yes	At least one page.
`description`	string	no	Description.
`version`	integer >= 0	no	Published version, increased by core on every publish (0 = never published).

FIELD	TYPE	REQUIRED	DESCRIPTION
`theme`	`light` | `dark` | `auto`	no	Color theme.
`skin`	`console` | `lobby`	no	Visual skin. Missing or unknown values render as `console`.
`access`	object	no	Who may open it (below). Default `{ "mode": "open" }`.
`roomParam`	boolean	no	Room-relative: device references with room `"$room"` use the room the dashboard is opened with.
`room`	id	no	Room used when none is given (room-relative dashboards).
`generated`	object	no	Set by the generator: `{ "kind": "room" | "show" | "overview", "source" }`.

Access:

FIELD	DESCRIPTION
`mode`	`open`: anyone on the network. `pin`: a PIN unlocks it. `panels`: enrolled panels (API tokens) only.
`pin`	Write-only, 4 to 12 digits. Core stores it as `pinHash` and never returns it.
`pinHash`	Set by core.

Page:

FIELD	TYPE	REQUIRED	DESCRIPTION
`id`	id	yes	Page id.
`name`	string	yes	Display name.
`widgets`	array of widgets	yes	In reading order (used to reflow when a screen class has no layout).
`layouts`	object	yes	Grid positions per screen class: `phone`, `tablet`, `panel`, `desktop` (may be `{}`).

Screen classes: `phone` below 600 px wide, `tablet` below 1024, `panel` below 1440, `desktop` above. A layout is a list of `{ "widget", "x", "y", "w", "h" }` on a grid of 12 columns (4 on phone).

Widget:

FIELD	TYPE	REQUIRED	DESCRIPTION
`id`	id	yes	Widget id, unique on its page.
`type`	string	yes	`button`, `toggle`, `slider`, `meter`, `value`, `source`, `scenes`, `status`, `transport`, `cue`, `clock`, `label`, `image` or `group`.
`label`	string	no	Label.
`props`	object	no	Property name to binding. Which properties a widget reads depends on its type.
`action`	widget action	no	What it does when used.
`offAction`	widget action	no	Toggles: sent when turning off (`action` is sent when turning on).
`confirm`	`none` | `confirm` | `hold`	no	Critical actions: ask first, or require press-and-hold.
`parent`	id	no	The `group` widget on the same page it belongs to.
`generated`	object	no	Set by the generator: `{ "source", "removed" }`.

Bindings (`props` values):

KIND	FIELDS	VALUE
`const`	`value`	A fixed value.
`state`	`device`, `capability`, `key`	A device state value, for example `tangle.level` / `level`.
`status`	`device`	A device's status.
`timeline`	`timeline`, `key` (`status`, `positionMs`, `lastCue`, `nextCue`)	Timeline state.
`system`	`key` (`connection`, `role`, `nodeId`, `license`, `time`)	Core and connection information.

A `device` reference is a device id, or a selector among a room's devices: `{ "room": "<room id>" | "$room", "capability", "tag", "index" }` (`index`: which match in project order, default 0).

Widget actions:

KIND	FIELDS	DESCRIPTION
`intent`	`name`, `target`, `params`	Sends an intent. A param value `"{{value}}"` (or the older `"$value"`) is replaced by the widget's value (slider level, selected input…); `"Level {{value}}"` fills it in as text. `target` is `{ "kind": "device", "device" }`, `{ "kind": "room", "id" }` (`id` may be `"$room"`), `{ "kind": "tag", "tag" }` or `{ "kind": "all" }`.
`timeline`	`timeline`, `op` (`play`, `pause`, `stop`, `go`)	Drives a timeline.

Automations

Run by tangle-core on the primary (or a standalone node). The Automations guide lists every trigger, condition and step kind with its fields.

FIELD	TYPE	REQUIRED	DESCRIPTION
`id`	id	yes	Automation id.
`name`	string	yes	Display name.
`steps`	array	no	Run in order. `actions`, the name before contract 0.2.30, is still read; core saves `steps`.
`triggers`	array	no	Any one starts a run. Without triggers it runs only when something runs it (a button, a listener, another automation, the API).
`conditions`	array	no	All must hold when triggered or run by hand (unless run with `force`).
`params`	array	no	Inputs (`name`, `label`, `default`), read as `{{name}}`.
`description`	string	no	Description.
`folder`	string	no	Folder in the automation list.
`enabled`	boolean	no	Default true. `false`: triggers do not start it; it still runs by hand and from other steps.

FIELD	TYPE	REQUIRED	DESCRIPTION
`mode`	`single` | `restart` | `queued` | `parallel`	no	What a trigger does while a run is in progress. `single` (default): ignored. `restart`: cancels the run and starts again. `queued`: runs after it (at most 10 waiting). `parallel`: runs alongside (at most 10).
`continueOnError`	boolean	no	Keep running the next steps when one fails (the run ends as `partial`).
`priority`	integer 0-100	no	Priority of the intents it sends.

Trigger kinds: `deviceState`, `deviceEvent`, `deviceStatus`, `time`, `timeline`, `variable`, `control`, `listener`, `startup`, `manual`. Condition kinds: `state`, `status`, `time`, `timeline`, `variable`, `control`. Step kinds: `intent`, `action`, `timeline`, `delay`, `if`, `repeat`, `while`, `waitUntil`, `set`, `control`, `automation`, `call`, `stop`, `comment`, `log`; every step takes `disabled`.

Alert rules

Monitoring (Sentinel) is part of Tangle Pro and Ultimate. See Sentinel.

FIELD	TYPE	REQUIRED	DESCRIPTION
`id`	id	yes	Rule id.
`name`	string	yes	Display name.
`severity`	`info` | `warning` | `critical`	yes	Severity of its alerts.
`condition`	object	yes	What to watch (below).
`description`	string	no	Description.
`enabled`	boolean	no	Default true.
`forSeconds`	integer >= 0	no	How long the condition must hold before the alert fires. Default 0.
`clearForSeconds`	integer >= 0	no	How long the condition must be false before the alert resolves. Default 0.

FIELD	TYPE	REQUIRED	DESCRIPTION
`schedule`	object	no	Active only `{ "days", "from", "to" }` (local time; `from` later than `to` spans midnight).
`channels`	array of ids	no	Channels to notify. Empty or absent: every channel whose `minSeverity` allows it.
`repeatMinutes`	integer >= 1	no	Notify again while firing and not acknowledged.
`notifyOnResolve`	boolean	no	Default true.
`message`	string	no	Template with `{name}`, `{subject}`, `{value}`, `{severity}`.

Conditions:

KIND	FIELDS	DESCRIPTION
`status`	`devices`, `rooms`, `modules`, `statuses`	Devices whose status is one of `statuses` (default `offline` and `error`). Devices are chosen by id, room or module; all devices when none is given.
`state`	`capability`, `key` (required); `devices`, `rooms`, `modules`, `above`, `below`, `equals`, `notEquals`, `clear`	A device state value compared to a threshold. `clear` is the value that resolves the alert (hysteresis).
`metric`	`name` (required); `labels`, `above`, `below`, `equals`, `notEquals`, `clear`	A custom or probe metric compared to a threshold, per label set.
`stale`	`seconds` (required); `devices`, `rooms`, `modules`, `capability`, `key`, `metric`, `labels`	No update for `seconds`: a device state key (`capability` and `key`) or a metric (`metric` and `labels`).
`coreNode`	`on` (`down`, `unsynced`, `noPrimary`); `nodes`	A core node is not heard, is out of step, or the cluster has no primary. `nodes` limits it to some node ids.

Metric names use letters, digits, `.`, `_` and `-`, for example `room.temperature`. Labels are string pairs, for example `{ "room": "server" }`.

Notification channels

FIELD	TYPE	REQUIRED	DESCRIPTION
`id`	id	yes	Channel id.
`name`	string	yes	Display name.
`kind`	string	yes	`email`, `sms`, `push`, `webhook`, `slack` or `teams`.
`settings`	object	yes	Depends on `kind` (below).
`enabled`	boolean	no	Turn the channel off without deleting it.
`minSeverity`	`info` | `warning` | `critical`	no	Lowest severity sent through this channel.
`quietHours`	object	no	`{ "days", "from", "to", "includeCritical" }`: non-critical notifications are dropped in this window; critical ones too when `includeCritical` is true.

Secret settings (marked below) go to the OS credential store when set through the designer or the API; the project keeps `********`.

KIND	SETTINGS
`email`	`host`, `from`, `to` (list) required; `port`; `security` (`starttls` default on port 587, `tls` on 465, `none` on 25 for trusted networks only); `username`; `password` (secret).
`sms`	`provider` (`twilio`), `accountSid`, `from` (E.164 number or messaging service id), `to` (list) required; `authToken` (secret).
`push`	`provider` (`ntfy` or `pushover`) required. ntfy: `server` (default `https://ntfy.sh`), `topic`, `token` (secret). Pushover: `userKey`, `appToken` (secret).
`webhook`	`url` required; `signingKey` (secret: signs the body, header `X-Tangle-Signature: sha256=<hex HMAC>`); `headers`.
`slack`	`webhookUrl` (secret, required).
`teams`	`webhookUrl` (secret, required): a Teams workflow (Power Automate) webhook.

Maintenance windows

A maintenance window mutes notifications, once or every week.

FIELD	TYPE	REQUIRED	DESCRIPTION
`id`	id	yes	Window id.
`name`	string	yes	Display name.
`start`, `end`	timestamps	once	A single period.
`days`, `from`, `to`	weekdays, `HH:MM`	weekly	A weekly period (local time).
`rules`	array of ids	no	Alert rules it applies to.
`devices`	array of ids	no	Devices it applies to.
`rooms`	array of ids	no	Rooms it applies to.

With no `rules`, `devices` or `rooms`, it applies to everything.

Probes

A probe checks a network asset that has no device module. It produces the metrics `probe.up`, `probe.latencyMs` and, for HTTP, `probe.httpStatus`, labelled `probe=<id>`.

FIELD	TYPE	REQUIRED	DESCRIPTION
`id`	id	yes	Probe id.
`name`	string	yes	Display name.
`kind`	`tcp` | `http`	yes	Check type.
`host`, `port`	string, 1-65535	tcp	Address to connect to.
`url`	string	http	Address to request.
`expectStatus`	array of status codes	no	HTTP only: accepted status codes. Default 200 to 399.
`room`	id	no	Room it belongs to.
`enabled`	boolean	no	Turn it off without deleting it.

FIELD	TYPE	REQUIRED	DESCRIPTION
`intervalSeconds`	integer >= 5	no	Default 30.
`timeoutMs`	100 to 60000	no	Default 5000.

Related files

- `tangle.lock` next to the project: module versions written by `tangle module lock`, made active when tangle-core starts. See CLI.
- Secrets, dashboard drafts and history, and alert history live in the Tangle data folder, not in the project. See CLI.

EXTEND

Module SDK

Write a Tangle device module in JavaScript, TypeScript or Python: manifest, settings, controls, protocol, SDK APIs, testing and installing.

A device module teaches Tangle to control one kind of device. It is a folder with a manifest (`tangle-module.json`) and a program that tangle-core starts as a child process and talks to over stdin and stdout. This page shows how to write one with the JavaScript/TypeScript SDK (`@tangle/module-sdk`) or the Python SDK (`tangle-module-sdk`), test it, and install it.

What a module is

- One folder per module, named after the module id.
- `tangle-module.json`: the manifest. It says how to start the module, which capabilities it uses, which settings a device needs (`configSchema`), and which ready-made dashboard controls it offers.
- An entry file: `index.js`, `main.py` or a native executable.
- tangle-core starts **one process per module** and sends it every device that uses it. The module owns the connection to each device (TCP, UDP, HTTP, WebSocket…). The SDK handles the protocol, reconnects, polling, timeouts and status reporting, so you write only the device logic.

Start from one of the reference modules in the Tangle source tree: `tangle-contract/examples/virtual-light-js`, `virtual-light-ts` and `virtual-light-py` (the same simulated light in three languages), or `tangle-connect/modules/misc.pjlink.projector`, a complete networked module with password, polling and discovery.

Naming and folder

Module ids follow `<manufacturer or misc>.<product or protocol>.<category>[.<version>]`:

- **manufacturer**: lowercase brand without spaces (`barco`, `qsc`), or `misc` for open protocols and generic devices.
- **product or protocol**: the product line (`watchout`, `pulse`) or the protocol (`pjlink`, `artnet`).
- **category**: one of `timecode`, `projector`, `display`, `mediaserver`, `engine`, `player`, `computer`, `io`, `dsp`, `lighting`, `console`, `switcher`, `control`.
- **version** (optional): a protocol or product generation when they are not compatible (`v6`, `v7`, `ma2`). It is not the module's own version.

Use only `a-z`, `0-9` and `-` inside a part. When the manifest declares a `category`, the id's third part must equal it and the folder must be named exactly like the id: `tangle module add` refuses a folder `my-projector` holding `barco.pulse.projector`. A manifest without a category installs with a warning.

Python package names cannot contain dots, so put Python code in a normally named package inside the folder (`misc.ssh.computer/ssh_computer/`). Only the folder name matters.

The manifest

`tangle-module.json` is validated against `ModuleManifest` in `tangle-contract/schema/module.schema.json`.

FIELD	REQUIRED	MEANING
`id`	yes	Module id, as above
`name`	yes	Display name
`version`	yes	Module version, `1.2.3` (optional `-suffix`)
`publisher`	yes	`{ "name": "...", "url": "https://..." }`
`channel`	yes	`official` or `community`
`contract`	yes	Semver range of the Tangle contract supported, for example `"^0.2.0"`
`runtime`	yes	`{ "kind": "node", "minVersion": "20.0.0" }`, `{ "kind": "python", "minVersion": "3.11" }` or `{ "kind": "native" }`
`entry`	yes	Entry file, relative to the folder
`description`		One or two sentences
`category`		Device category (see above)
`devices`		Supported models, `[{ "manufacturer": "Epson", "model": "EB-L1100U" }]`. Discovery uses this list to suggest the module
`capabilities`		Standard capabilities used, for example `["tangle.power", "tangle.input"]`
`customCapabilities`		Capabilities this module defines (see below)
`configSchema`		JSON Schema for one device's settings

FIELD	REQUIRED	MEANING
`permissions`		Any of `network`, `serial`, `usb`, `midi`, `filesystem`. Shown to the operator for consent at install
`discovery`		`true` if the module implements `discover`
`controls`, `controlLayouts`		Ready-made dashboard controls and arrangements
`configurators`		Device-side setup steps
`process`		Native modules only: modules with the same process name share one process

While the contract is 0.x, `"^0.2.0"` does not match 0.3.0. Tangle refuses to install a module whose `contract` range does not include its own contract version.

Capabilities

A capability is a set of state keys and actions that lets dashboards, scenes and timelines control any device the same way. Standard capabilities:

CAPABILITY	STATE	ACTIONS
`tangle.power`	`on`	`on`, `off`, `toggle`
`tangle.input`	`input`	`select` (`input`)
`tangle.level`	`level` (0..1)	`set` (`level`, `fadeMs`)
`tangle.mute`	`muted`	`mute`, `unmute`, `toggle`
`tangle.preset`	`current`	`recall` (`preset`)
`tangle.transport`	`status`, `positionMs`, `clip`	`play`, `pause`, `stop`, `seek` (`positionMs`), `load` (`clip`)
`tangle.timecode`	`timecode`, `fps`, `running`, `signal`	`start`, `stop`, `locate` (`timecode`)

Define your own in `customCapabilities`, with an id `<module prefix>.<name>`:

```json
{
  "id": "pjlink.lamp",
  "version": 1,
  "description": "Lamp use (first lamp).",
  "state": { "hours": { "type": "number" }, "on": { "type": "boolean" } },
  "actions": {}
}
```

State values and action parameters are typed: `boolean`, `number` (`min`, `max`, `step`, `unit`), `string`, `enum` (`values`) or `json`.

Settings: configSchema

`configSchema` is a JSON Schema for one device's `config`. Tangle validates device settings against it, and the designer builds the device's settings form from it: strings, numbers, integers, booleans and enums become fields, `default` becomes a placeholder, `required` fields are marked, and anything else is edited as JSON.

Mark passwords, PINs, tokens and keys with `"x-secret": true`. Tangle stores them in the OS credential store, never in the project file, and fills them in before sending the device to your module.

```json
"configSchema": {
  "type": "object",
  "properties": {
    "host": { "type": "string", "minLength": 1, "description": "IP address or host name of the projector." },
    "port": { "type": "integer", "minimum": 1, "maximum": 65535, "default": 4352 },
    "password": { "type": "string", "x-secret": true, "description": "PJLink password, if the projector has one." }
  },
  "required": ["host"],
  "additionalProperties": false
}
```

DMX modules put the universe a device uses in an integer `universe` setting. Licensing counts distinct `universe` values to apply the edition's universe limit.

Controls and layouts

`controls` are ready-made widgets for this module's devices, offered by the dashboard builder. Bindings and actions apply to whichever device the control is used with.

FIELD	MEANING
`id`, `label`, `description`	Identity and text
`widget`	`button`, `toggle`, `slider`, `meter`, `value`, `source`, `scenes`, `status`, `transport`, `cue`, `clock`, `label`, `image` or `group`

FIELD	MEANING
`props`	Widget properties, each bound to a constant `{ "kind": "const", "value": ... }`, a state value `{ "kind": "state", "capability": "...", "key": "..." }` or the device status `{ "kind": "status" }`
`action`	Intent sent when used: `{ "kind": "intent", "name": "level.set", "params": { "level": "{{value}}" } }`. `{{value}}` (or the older `$value`) is replaced by the widget's value
`offAction`	Intent for the off position of a toggle
`confirm`	`none`, `confirm` or `hold`
`size`	Default size in grid cells, `{ "w": 3, "h": 2 }`

`controlLayouts` arrange controls into named sets (`compact`, `full`...) that the builder offers when a device is dragged onto a dashboard. Each placement gives `control`, `x`, `y`, `w`, `h` in grid cells. See Dashboards.

```JSON
"controls": [
  {
    "id": "power", "label": "Power", "widget": "toggle",
    "props": { "value": { "kind": "state", "capability": "tangle.power", "key": "on" } },
    "action": { "kind": "intent", "name": "power.on" },
    "offAction": { "kind": "intent", "name": "power.off" },
    "size": { "w": 3, "h": 2 }
  },
  { "id": "status", "label": "Status", "widget": "status", "props": { "value": { "kind":
"status" } }, "size": { "w": 3, "h": 1 } }
],
"controlLayouts": [
  { "id": "compact", "label": "Compact", "controls": [
    { "control": "power", "x": 0, "y": 0, "w": 3, "h": 2 },
    { "control": "status", "x": 0, "y": 2, "w": 3, "h": 1 }
  ] }
]
```

Configurators

`configurators` describe setup needed on the device side, such as a script to copy to a player or remote access to enable on a PC. Each has an `id`, a `name`, optional `platforms` (`macos`, `windows`, `linux`, `device`) and ordered `steps` of kind:

- `info`: text only (Markdown in `text`);
- `download`: files your module generates, to copy to the device;
- `script`: a script your module generates, for the operator to run on the target;
- `check`: runs an action (`capability`, `action`) to confirm the setup worked.

Your module builds the files and script from a device's configuration when the host calls `configurator.run` (see the SDK APIs below).

> **Note:** The designer wizard that shows configurator steps is not available yet.

A minimal module in JavaScript

A simulated dimmable light, as in `tangle-contract/examples/virtual-light-js`. Folder `misc.virtual-light.lighting/`:

`tangle-module.json`:

```json
{
  "id": "misc.virtual-light.lighting",
  "name": "Virtual light",
  "version": "0.1.0",
  "description": "Simulated dimmable light.",
  "category": "lighting",
  "publisher": { "name": "Your name" },
  "channel": "community",
  "contract": "^0.2.0",
  "runtime": { "kind": "node", "minVersion": "20.0.0" },
  "entry": "index.js",
  "devices": [{ "manufacturer": "Tangle", "model": "Virtual light" }],
  "capabilities": ["tangle.power", "tangle.level"],
  "configSchema": {
    "type": "object",
    "properties": { "startOn": { "type": "boolean", "default": false } },
    "additionalProperties": false
  },
  "permissions": []
}
```

`package.json`:

```json
{
  "name": "misc.virtual-light.lighting",
  "version": "0.1.0",
  "private": true,
  "type": "module",
  "dependencies": { "@tangle/module-sdk": "workspace:*" }
}
```

`index.js`:

```JS
import { runModule } from "@tangle/module-sdk";

runModule({
  version: "0.1.0",
  createDriver(ctx) {
    let on = ctx.config.startOn === true;
    let level = 1;

    return {
      async connect() {
        // A real module opens its connection here.
        ctx.setState("tangle.power", { on });
        ctx.setState("tangle.level", { level });
      },

      async execute(cmd) {
        const key = `${cmd.capability}.${cmd.action}`;
        switch (key) {
          case "tangle.power.on":
          case "tangle.power.off":
          case "tangle.power.toggle":
            on = cmd.action === "toggle" ? !on : cmd.action === "on";
            ctx.setState("tangle.power", { on });
            return;
          case "tangle.level.set":
            level = Math.min(1, Math.max(0, Number(cmd.params?.level)));
            ctx.setState("tangle.level", { level });
            return;
          default:
            throw new Error(`unsupported action: ${key}`);
        }
      },
    };
  },
});
```

The SDK lives in the Tangle source tree at `tangle-contract/sdk/js`. The `workspace:*` dependency works when the module folder is inside the Tangle pnpm workspace, for example in `tangle-connect/modules/`. Build the SDK once:

```SH
pnpm install
pnpm --filter @tangle/module-sdk build
```

In TypeScript, import the types too: `import { runModule, type ActionCommand, type DeviceContext, type DeviceDriver } from "@tangle/module-sdk";` and compile to the entry file named in the manifest (see `examples/virtual-light-ts`).

A minimal module in Python

The same light in Python, as in `tangle-contract/examples/virtual-light-py`. Folder `misc.virtual-light-py.lighting/`, with the same manifest except `"id": "misc.virtual-light-py.lighting"`, `"runtime": { "kind": "python", "minVersion": "3.11" }` and `"entry": "main.py"`.

`main.py`:

```python
"""A simulated dimmable light, written in Python."""

from tangle_module_sdk import DeviceDriver, run_module


class VirtualLight(DeviceDriver):
    async def connect(self):
        # A real module opens its connection here.
        self.on = self.ctx.config.get("startOn", False)
        self.level = 1.0
        self.ctx.set_state("tangle.power", {"on": self.on})
        self.ctx.set_state("tangle.level", {"level": self.level})

    async def execute(self, cmd):
        key = f"{cmd['capability']}.{cmd['action']}"
        if key in ("tangle.power.on", "tangle.power.off", "tangle.power.toggle"):
            self.on = (not self.on) if cmd["action"] == "toggle" else cmd["action"] == "on"
            self.ctx.set_state("tangle.power", {"on": self.on})
        elif key == "tangle.level.set":
            self.level = min(1.0, max(0.0, float(cmd.get("params", {}).get("level", 0))))
            self.ctx.set_state("tangle.level", {"level": self.level})
        else:
            raise ValueError(f"unsupported action: {key}")


if __name__ == "__main__":
    run_module(VirtualLight, version="0.1.0")
```

The Python SDK uses the standard library only and needs Python 3.11 or later. For development, install it from the source tree:

```sh
python3 -m venv .venv && source .venv/bin/activate     # Windows: .venv\Scripts\activate
pip install -e tangle-contract/sdk/python
```

For an installed module, the SDK must be importable when Tangle runs `main.py`: copy `tangle-contract/sdk/python/tangle_module_sdk` next to the entry file, which is how the official Python modules are packaged.

Talking to a real device

Open the connection in `connect()` with a transport from the device context. The SDK watches it: if it closes, the device goes `offline` and `connect()` runs again with backoff. Use `poll()` to read state regularly.

This module drives output 1 of a 4x2 video matrix that answers `3*1!` with `Out1 In3 All` and `1!` with the current input. The simulator's example config has such a matrix on TCP port 5023, so you can try it without hardware.

```JS
import { runModule } from "@tangle/module-sdk";

runModule({
  version: "0.1.0",
  createDriver(ctx) {
    const output = Number(ctx.config.output ?? 1);
    let conn;

    const readInput = async () => {
      const answer = await conn.request(`${output}!\r\n`, { match: (f) => /^\d+$/.test(f) });
      ctx.setState("tangle.input", { input: answer });
    };

    return {
      async connect() {
        conn = await ctx.tcp({ host: ctx.config.host, port: Number(ctx.config.port ?? 5023),
delimiter: "\r\n" });
        await readInput();
      },
      poll: readInput,
      async execute(cmd) {
        if (cmd.capability !== "tangle.input" || cmd.action !== "select") {
          throw new Error(`unsupported action: ${cmd.capability}.${cmd.action}`);
        }
        const input = String(cmd.params?.input ?? "");
        if (!/^[1-4]$/.test(input)) throw new Error("input must be 1 to 4");
        await conn.request(`${input}*${output}!\r\n`, { match: (f) =>
f.startsWith(`Out${output} `) });
        ctx.setState("tangle.input", { input });
      },
    };
  },
});
```

The same in Python:

```python
import re
from tangle_module_sdk import DeviceDriver, run_module


class Matrix(DeviceDriver):
    async def connect(self):
        cfg = self.ctx.config
        self.output = int(cfg.get("output", 1))
        self.conn = await self.ctx.tcp(cfg["host"], int(cfg.get("port", 5023)),
delimiter="\r\n")
        await self.poll()

    async def poll(self):
        answer = await self.conn.request(f"{self.output}!\r\n", match=lambda f: f.isdigit())
        self.ctx.set_state("tangle.input", {"input": answer})

    async def execute(self, cmd):
        if (cmd["capability"], cmd["action"]) != ("tangle.input", "select"):
            raise ValueError(f"unsupported action: {cmd['capability']}.{cmd['action']}")
        value = str(cmd.get("params", {}).get("input", ""))
        if not re.fullmatch(r"[1-4]", value):
            raise ValueError("input must be 1 to 4")
        await self.conn.request(f"{value}*{self.output}!\r\n", match=lambda f:
f.startswith(f"Out{self.output} "))
        self.ctx.set_state("tangle.input", {"input": value})


if __name__ == "__main__":
    run_module(Matrix, version="0.1.0")
```

Its manifest lists `"capabilities": ["tangle.input"]`, `"category": "switcher"`, `"permissions": ["network"]`, and a `configSchema` with `host` (required), `port` and `output`.

SDK APIs

Module definition

JAVASCRIPT (`runModule({...})`)	PYTHON (`run_module(driver, version, ...)`)	PURPOSE
`version`	`version`	Module version reported to the host; use the manifest's
`createDriver(ctx)`	the `DeviceDriver` subclass	One driver instance per device
`discover({ timeoutMs, log })`	`discover=async fn(timeout_seconds)`	Return devices found on the network

JAVASCRIPT (`RUNMODULE({...})`)	PYTHON (`RUN_MODULE(DRIVER, VERSION, ...)`)	PURPOSE
`configurators: { id: fn }`	`configurators={"id": async fn}`	Build configurator files and scripts
`timing: { ... }`	`timing=Timing(...)`	Change timeouts and delays

Driver methods

METHOD	WHEN IT RUNS
`connect()`	When the device is added, and again after a lost connection. Open the connection and send the initial state. Throwing marks the device offline and retries with backoff
`execute(cmd)`	For each action. `cmd` has `deviceId`, `capability`, `action`, optional `params` and `at` (epoch ms, for timeline pre-roll; the SDK waits until then). Return nothing for success, or throw
`poll()`	Every 10 s while online, never at the same time as an action. Read the device and set what changed
`disconnect()`	When the device is removed or reconnected
`pollIntervalMs` (JS), `poll_interval` (Python, seconds)	Per-device poll interval

Errors to throw:

- `RetryableError`: the action may work later (device busy, warming up). Core may retry it.
- `ConfigError`: from `connect()`, when retrying cannot help (password rejected, setting missing). The device shows `error` and is not retried.
- `TimeoutError` (JS: exported by the SDK; Python: the built-in one): from `connect()` or `poll()` when the device did not answer. The device shows `timeout` instead of `offline`.
- Anything else from `execute()` fails the action, not retryable.

Device context

JAVASCRIPT (`CTX`)	PYTHON (`SELF.CTX`)	PURPOSE
`ctx.device`, `ctx.config`	`ctx.device`, `ctx.config`, `ctx.device_id`	The device and its settings, secrets filled in

JAVASCRIPT (`ctx`)	PYTHON (`self.ctx`)	PURPOSE
`ctx.dataDir`	`ctx.data_dir`	Folder the module may write to
`ctx.setState(capability, values)`	`ctx.set_state(capability, values)`	Report changed state. Send the initial state after connecting, then every change, including changes made outside Tangle
`ctx.setStatus(status, detail)`	`ctx.set_status(status, detail)`	Report a status yourself, for example `degraded` with an alarm text
`ctx.status`	`ctx.status`	Current status
`ctx.emit(name, data)`	`ctx.emit(name, data)`	Device event (a button press, an alarm) for automations
`ctx.log.info(msg, data)` (also `debug`, `warn`, `error`)	`ctx.log("info", msg, **data)`	Logs, written to stderr
`ctx.connectionLost(reason, "offline" or "timeout")`	`ctx.connection_lost(reason, status)`	For transports you manage yourself: reconnect with backoff
`ctx.signal`	`ctx.stopped`	Aborted (JS) or set (Python) when the device is removed

Transports

JAVASCRIPT	PYTHON	WHAT IT IS
`ctx.tcp({ host, port, delimiter })`	`ctx.tcp(host, port, delimiter=...)`	TCP with framing (default delimiter `\r`), one request at a time with a timeout (default 3 s), answer matching with `match`, unsolicited frames to `onFrame` / `on_frame`. A closed connection reconnects the device
`udpSearch({ port, message, timeoutMs })`, `udpSend(...)`	`udp_search(port, message, timeout)`, `udp_send(...)`	Broadcast a search and collect answers (for `discover`), or send one datagram
`ctx.udp({ host, port })`		UDP socket for one device, with `request` and `onMessage`
`ctx.http({ baseUrl })`		REST client: `get`, `post`, `put`, `patch`, `delete`. Timeouts become `TimeoutError`, 503 and 429 are retryable

JAVASCRIPT	PYTHON	WHAT IT IS
`ctx.websocket({ url })`		WebSocket (Node.js 22+). A closed socket reconnects the device
`ctx.osc({ host, port })`		OSC over UDP, with `encodeOsc` / `decodeOsc`
`JsonRpcClient`		JSON-RPC over any of the above
`ctx.watch(transport)`		Watch a transport you opened yourself

TCP options also include `frameLength` (custom framing for binary or length-prefixed protocols such as Modbus TCP), `jsonFrameLength` (JSON objects sent back to back), `encoding` (`"binary"` for Buffers), `requestTimeoutMs` and `sendIntervalMs` (minimum time between writes). UDP cannot see a device go away: implement `poll()` so a silent device becomes `timeout`. `@tangle/module-sdk/testing` has a small WebSocket server for writing simulators.

> **Note:** A serial (RS-232) transport is not available yet in the SDKs. Use a serial-to-IP adapter with the TCP helper.

Timing

JAVASCRIPT (MS)	PYTHON (`TIMING`, SECONDS)	DEFAULT
`actionTimeoutMs`	`action_timeout`	10 s
`connectTimeoutMs`	`connect_timeout`	10 s
`reconnectMinMs`	`reconnect_min`	1 s, doubling up to the maximum
`reconnectMaxMs`	`reconnect_max`	30 s
`pollIntervalMs`	`poll_interval`	10 s

Discovery

Set `"discovery": true` in the manifest and implement `discover`. The host calls it during network scans, in a separate short-lived process, never the project's running one. Search for at most the given time, then return a list of `{ address, name, manufacturer, model, config }` (only `address` is required). `config` pre-fills the device settings when the operator adds it.

```JS
import { runModule, udpSearch } from "@tangle/module-sdk";

runModule({
  version: "0.1.0",
  createDriver: (ctx) => ({ /* ... */ }),
  async discover({ timeoutMs }) {
    const answers = await udpSearch({ port: 4352, message: "%2SRCH\r", timeoutMs: timeoutMs -
500 });
    return answers.map((a) => ({ address: a.address, config: { host: a.address } }));
  },
});
```

Configurators

```JS
runModule({
  version: "0.1.0",
  createDriver: (ctx) => ({ /* ... */ }),
  configurators: {
    "enable-remoting": ({ config }) => ({
      files: [],
      script: { language: "powershell", content: "Enable-PSRemoting -Force" },
    }),
  },
});
```

In Python: `run_module(Driver, version="0.1.0", configurators={"enable-remoting": build})` where `build` is `async def build(config, device_id)` returning a dict with `files` (each `name`, `content` as text or bytes, optional `executable`), `script` (`language`, `content`) and `instructions`. Script languages are `powershell`, `bash`, `shell`, `brightscript` and `text`.

Statuses

The SDK reports device status for you:

SITUATION	STATUS
First connection in progress	`connecting`
`connect()` succeeded	`online`
One failed poll	`degraded`
`connect()` failed, or the connection closed	`offline`, then retry after 1 s, 2 s, 4 s… up to 30 s
`TimeoutError`, or two failed polls without an answer	`timeout`, then retry

SITUATION	STATUS
`ConfigError`	`error`, no retry

Every module must report at least `online`, `offline` and `timeout`. Actions for an `offline` or `timeout` device fail at once with `retryable: true`. Actions sent while the first connection is being made wait for it, up to the connect timeout. Report `degraded` yourself with `setStatus` when the device answers but has a problem (a lamp warning, an alarm).

The module protocol

The SDKs implement the protocol, so you only need this to debug or to write a module in another language. The full reference is `tangle-contract/docs/module-protocol.md`; message shapes are in `tangle-contract/schema/module-protocol.schema.json`.

- The host starts the module with the runtime from the manifest (`node <entry>`, `python <entry>`, or the native executable), with the module folder as working directory.
- **stdin** carries host-to-module messages, **stdout** module-to-host messages: JSON-RPC 2.0, one JSON object per line, UTF-8. **stderr** is for logs. Never print logs to stdout. The SDKs redirect `console.log` to stderr.
- When stdin closes, the module must clean up and exit.

Methods (host to module):

METHOD	PURPOSE
`initialize`	First call, with `contractVersion`, `host`, `moduleId`, `dataDir`. Answer within 5 s with `moduleId`, `version`, `sdk`
`device.add`	A device to drive, secrets filled in. Answer as soon as it is registered; report progress with `device.status`
`device.remove`	Disconnect and forget a device
`device.execute`	Run an action. The result is `{ "ok": true }` or `{ "ok": false, "error": "...", "retryable": true }`; failures are results, not JSON-RPC errors. Actions for one device run one at a time, in order
`discover`	Search the network for `timeoutMs` (only with `"discovery": true`)
`configurator.run`	Build a configurator's files and script for one device configuration
`ping`	Answer `{ "now": <epoch ms> }`. 3 missed pings and the host restarts the module

METHOD	PURPOSE
`shutdown`	Disconnect every device, answer, exit with code 0

Notifications (module to host): `device.status`, `device.state` (changed values for one capability) and `device.event`.

```JSON
{"jsonrpc":"2.0","id":4,"method":"device.execute","params":{"deviceId":"proj-
1","capability":"tangle.power","action":"on"}}
{"jsonrpc":"2.0","id":4,"result":{"ok":false,"error":"lamp warming up","retryable":true}}
{"jsonrpc":"2.0","method":"device.state","params":{"deviceId":"proj-
1","capability":"tangle.power","values":{"on":true}}}
```

Error codes: `-32700` invalid JSON, `-32601` unknown method (also `discover` in a module without it), `-32602` bad params (unknown configurator), `-32603` internal error. The host fails an action not answered within 15 s, without restarting the module.

Test your module

Conformance kit

The conformance kit starts your module the way Tangle does and checks it: answers `initialize` within 5 s, answers `ping`, rejects unknown methods and invalid JSON, handles actions for unknown devices, answers `discover` in time (when declared), runs a device scenario, validates every message against the contract, keeps stdout clean, shuts down within 5 s and exits when stdin closes.

```SH
pnpm exec tangle-conformance path/to/module           # inside the Tangle workspace
node tangle-contract/conformance/run.mjs path/to/module
```

For Python modules, set `PYTHON` to choose the interpreter. Add a `conformance.json` to the module folder to run a device scenario:

```JSON
{
  "device": { "id": "light-1", "name": "Test light", "moduleId": "misc.virtual-
light.lighting", "config": {} },
  "steps": [
    {
      "execute": { "capability": "tangle.power", "action": "on" },
      "expect": { "ok": true, "state": { "capability": "tangle.power", "values": { "on": true
} } }
    },
    { "execute": { "capability": "tangle.power", "action": "explode" }, "expect": { "ok":
false } }
  ]
}
```

The device must come online within 5 s, so point it at something that answers (a simulator).

Against a simulated device

Test against a virtual device before you have hardware. The simulator runs PJLink projectors, any line-based TCP or UDP device described by rules (`ascii` type), and Art-Net/sACN receivers, with faults to switch on: `offline`, `silent`, `latencyMs`, `fail`, `rejectAuth`, `drop` and `reboot`. Check that your module reports `offline` for an offline device, `timeout` for a silent one, and `online` again when the fault is cleared. Most official modules also ship their own `sim.js` or `sim.py` that you can use as a model.

In a running project

Run tangle-core with an extra modules folder (one sub-folder per module), without installing:

```SH
tangle-core run --project show.yaml -m ./my-modules
tangle-core modules -m ./my-modules        # list the modules found
```

Then test a device with `tangle device test` or **Test connection** in the designer.

Package and install

To install a module on a Tangle machine:

```SH
tangle module add ./misc.virtual-light.lighting
```

The folder is copied into the Tangle modules folder (`tangle module list` shows it as local and unsigned) and becomes the active version. Before installing, make sure:

- **JavaScript:** the folder runs on its own. Either keep `node_modules` with the runtime dependencies (symlinks are followed when copying; development dependencies are skipped), or bundle the entry into one file. The official packages bundle with esbuild.

- **TypeScript:** compile first; `entry` points to the compiled JavaScript.
- **Python:** list third-party dependencies in `requirements.txt`. At install, Tangle creates a virtual environment for the module (`.venv`) and installs them, from a `wheels/` folder when present (so it works offline). Ship the `tangle_module_sdk` package next to the entry file.
- **Permissions:** declare what the module needs. The operator confirms them at install and again when an update adds one.

See Devices for updates, rollback and per-project version locks.

> **Note:** Publishing modules to a marketplace, signed packages and managed Node.js/Python runtimes are not available yet.

Native modules

Timing-critical modules (timecode, MIDI, DMX) are native Rust programs built with the Rust SDK (`tangle-native-sdk`, in `tangle-connect/crates/native-sdk`), which behaves like the JS and Python SDKs. Their manifest uses `"runtime": { "kind": "native" }`, and `entry` names the executable, looked up in the module folder (with `.exe` on Windows), then in `TANGLE_NATIVE_DIR`, then next to the host executable. Modules with the same `process` name share one process, which receives the devices of all of them; each `device.add` carries the device's own `moduleId`.

EXTEND

Engine plugins

Control Unreal Engine, Unity and Godot projects from Tangle with triggers, properties and game events.

Tangle can drive real-time engines in installations, kiosks, virtual production and interactive shows: fire named triggers, set and watch named properties, and react to game events. Unity and Godot need a small Tangle plugin in the game. Unreal needs no Tangle plugin: it uses Epic's built-in Remote Control API.

ENGINE	TANGLE MODULE	WHAT GOES IN THE GAME	PORT
Unreal Engine 5 (and 4.26+)	`epic.unreal.engine`	Epic's Remote Control API plugin	HTTP 30010, WebSocket 30020
Unity 2021.3+	`unity.unity.engine`	Tangle Unity package (`dev.tangle.engine`)	TCP 9420
Godot 4.1+	`misc.godot.engine`	Tangle Godot addon	TCP 9420

The Unity package and the Godot addon are MIT licensed and live in the Tangle source tree under `tangle-connect/plugins/unity` and `tangle-connect/plugins/godot`.

What you get in Tangle

All three modules work the same way from the show's side:

- **Triggers** (in Unreal: exposed Blueprint functions) become buttons. The buttons are one `scenes` widget whose options are the game's list, so they follow what the game exposes without editing the dashboard.
- **Properties** can be set and watched. Numeric properties get sliders, with the range set in the game. Four slider slots are available; the `sliders` setting picks which properties go on them (default: the first numeric properties).
- **Game events** become device events named `game.<name>`, with their data, so automations can react to them.
- **Layouts:** `compact` (status, last event, trigger buttons), `operator` (adds the 4 sliders) and `full` (adds the game or preset name, slider property names and a reload button).

Statuses:

SITUATION	STATUS
Game or editor not running (port closed)	`offline`, reconnects on its own (1 s, 2 s… 30 s)
Running but not answering (a long load, stuck, paused in the debugger, a firewall dropping packets)	`timeout`
Wrong token (Unity, Godot) or passphrase (Unreal)	`error`, not retried until you change the device
Unreal preset not found (its level is not loaded)	`degraded`, retried at every poll

Each module has a setup step (configurator) with the install instructions and a final check that the game answers.

Unity

Install the package

1. In Unity: **Window > Package Manager > + > Add package from disk…**, and pick `tangle-connect/plugins/unity/package.json`. Or use **Add package from git URL…** with the repository URL and `?path=/tangle-connect/plugins/unity`.
2. Create an empty GameObject in your first scene, name it `Tangle`, and add **Tangle > Tangle Server**. It keeps itself across scene loads.
3. Press Play. The console shows `[Tangle] Listening on 0.0.0.0:9420`.
4. In Tangle, add a device with the module **Unity** (`unity.unity.engine`), `host` = this computer's IP address. Run the device's **Install the Tangle Unity package** check: it should pass.

Windows asks to allow the game (or the Unity editor) through the firewall the first time. Allow it on private networks, or Tangle cannot reach it.

The package works on any platform with TCP sockets (Windows, macOS, Linux, Android, iOS), but not WebGL: browsers cannot listen on a port.

Triggers

Add **Tangle > Tangle Trigger** to any GameObject, set **Trigger Name** (for example `intro`), and wire **On Trigger** to what should happen: play a Timeline, start an Animator state, call your method. **On Trigger With Argument** gets the first argument Tangle sends, as text. Several components with the same name all run.

From code:

```csharp
GetComponent<Tangle.TangleTrigger>().Triggered += args => Debug.Log($"intro with {args.Count} args");
Tangle.TangleServer.TriggerReceived += (name, args) => Debug.Log($"Tangle fired {name}");
```

Properties

Add **Tangle > Tangle Property** and set **Property Name** (`speed`), then either:

- set **Target** to a component and **Member** to one of its fields or properties (`intensity` on a Light, `speed` on your script). Types: float, double, int, long, bool, string, enum (as text);
- or leave **Target** empty: the component's own **Value** (float) is the property. Wire **On Value Changed** to whatever should follow it.

Tick **Use Range** and set **Min** and **Max** to clamp numbers and give Tangle's sliders their range (the default is 0 to 1). Untick **Writable** for values Tangle should only watch, such as a score. Changes made by the game are pushed to Tangle about 10 times per second.

IL2CPP builds with managed code stripping may remove engine properties nothing else uses. If a property works in the editor but not in a build, add a `link.xml` that preserves that type.

Game events

Call `Tangle.TangleServer.SendEvent("door-opened", 2);` from code, or add **Tangle > Tangle Event** and call its `Send()` from any UnityEvent (a UI button, an animation event, a Timeline signal). Tangle receives the device event `game.door-opened`.

Server settings

FIELD	DEFAULT	NOTES
Port	9420	Must match the device's `port` in Tangle
Bind Address	`0.0.0.0`	`127.0.0.1` accepts only Tangle on the same computer
Token	empty	When set, the device's `token` setting must match
Keep Across Scenes	on	`DontDestroyOnLoad`
Run In Background	on	The game answers while unfocused
Property Checks Per Second	10	How often values are compared for changes
Verbose	off	Logs connections and requests

Requests are handled on the main thread, once per frame, so triggers and properties can use any Unity API. A game at 60 fps answers within about 16 ms.

Actions

ACTION	PARAMETERS
`unity.trigger.fire`	`name`, optional `args` (array)
`unity.property.set`	`name`, `value` (converted to the property's type; the game may clamp it)
`unity.property.setSlot`	`slot` (1 to 4), `value`
`unity.property.get`	`name`
`unity.game.refresh`	Read the trigger and property list again
`unity.game.ping`	The setup check

State `unity.property.values` holds every property, for example `{ "speed": 4, "lights": true }`.

Godot

Install the addon

1. Copy `addons/tangle` into your project's `addons/` folder, keeping the path `res://addons/tangle/`.
2. **Project > Project Settings > Plugins**: enable **Tangle**. This adds the `Tangle` autoload and its settings.
3. Run the project. The output shows `[Tangle] Listening on *:9420`.
4. In Tangle, add a device with the module **Godot** (`misc.godot.engine`), `host` = this computer's IP address. Run the device's **Install the Tangle Godot addon** check: it should pass.

The addon works in desktop and mobile exports, not in the web export.

Triggers

Add a **TangleTrigger** node anywhere in a scene, set **Trigger Name** (`intro`), and connect its `triggered(args)` signal to what should happen (an AnimationPlayer, a cutscene). Several nodes with the same name all emit. From code, without a node:

```gdscript
func _ready():
    Tangle.add_trigger("intro", "Starts the intro")
    Tangle.trigger_received.connect(_on_tangle_trigger)

func _on_tangle_trigger(trigger_name: String, args: Array):
    if trigger_name == "intro":
        $AnimationPlayer.play("intro")
```

Properties

Add a **TangleProperty** node and set **Property Name** (`speed`), then either:

- set **Target** to a node and **Property** to one of its properties (`speed` on your script, `light_energy` on a light, `position:x` for a sub-property). Types: float, int, bool, String; other types are shown as text and read-only;
- or leave **Target** empty: the node's own **Value** (float) is the property. Follow it with the `value_changed_by_tangle(new_value)` signal.

Use Range with **Min Value** and **Max Value** clamps numbers and sets the slider range (default 0 to 1). Untick **Writable** for watch-only values.

Game events

```gdscript
Tangle.send_event("door-opened", {"door": 2})
```

Settings

In **Project Settings > General > Tangle > Server** (turn on Advanced Settings to see them):

SETTING	DEFAULT	NOTES
`tangle/server/port`	9420	Must match the device's `port` in Tangle
`tangle/server/bind_address`	`*`	`127.0.0.1` accepts only Tangle on the same computer
`tangle/server/token`	empty	When set, the device's `token` setting must match
`tangle/server/property_checks_per_second`	10	How often values are compared for changes
`tangle/server/verbose`	false	Prints connections and requests

Exported builds also take overrides on the command line after `--`:

```sh
game.exe -- --tangle-port=9500 --tangle-token=secret --tangle-bind=127.0.0.1
```

The addon runs on the main thread once per frame and keeps running while the scene tree is paused.

Actions

The Godot module has the same actions as Unity, under `godot.`: `godot.trigger.fire`, `godot.property.set`, `godot.property.setSlot`, `godot.property.get`, `godot.game.refresh` and `godot.game.ping`. Its state is `godot.property.values`.

Unreal Engine

The `epic.unreal.engine` module uses the Remote Control API that ships with Unreal Engine, in the editor or in a packaged game.

Enable the Remote Control API

1. **Edit > Plugins**, search "Remote Control", enable **Remote Control API** (and **Remote Control Web Interface** if you want Epic's browser control page). Restart the editor.
2. **Project Settings > Plugins > Remote Control**: check that the web server starts automatically, and note the HTTP and WebSocket ports. In UE 5.1 and later, look at the passphrase and allowed-clients settings: either allow the Tangle computer, or set a passphrase and put it in the device's `passphrase` setting.
3. In the editor, the console command `WebControl.StartServer` starts the server if it is not running. Test it from the Tangle computer:

   ```sh
   curl http://<unreal-pc>:30010/remote/info
   ```

4. **Packaged games** (nDisplay, kiosks): keep both plugins enabled for the build, start the game with `-RCWebControlEnable -RCWebInterfaceEnable`, and allow ports 30010 and 30020 in the Windows firewall.

Expose what the show needs

Recommended: a Remote Control Preset.

1. Content Browser > **Add > Remote Control > Remote Control Preset**, and name it (for example `Show`).
2. Open it and expose:
   - **functions**: Blueprint functions of level actors ("Play Intro", "Reset"). Each becomes a button in Tangle, under its display name;
   - **properties**: numbers, booleans and text (a light's intensity, a speed variable). Numbers become sliders; set their UI min and max so the sliders get the right range.
3. In Tangle, set the device's `preset` setting to the preset's name. Keep the preset's level loaded.

With a preset, the module opens the WebSocket and Unreal pushes value changes. If the WebSocket server is off (or `websocket` is false), values are polled every 5 s.

Without a preset, list objects directly in the device settings:

- `functions`: `{ name, objectPath, function, parameters }`
- `properties`: `{ name, objectPath, property, type, min, max }`

The object path is what the editor's right-click **Copy Reference** gives for a level actor, for example `/Game/Maps/Main.Main:PersistentLevel.BP_Show_C_1`. Both ways can be combined.

Game events from Unreal

The Remote Control WebSocket pushes changes of preset properties. To tell Tangle that something happened:

1. Add a String variable `LastEvent` to a level actor and expose it in the preset (display name, for example, "Last Event").
2. In Blueprints, set it to the event name (`intro-finished`) when the thing happens. Set it to an empty string first if the same event can happen twice in a row.
3. Set the device's `eventProperty` setting to `Last Event`. Each new value becomes the device event `game.<value>`.

Settings and actions

SETTING	DEFAULT
`host`	required
`port`	30010 (HTTP)
`wsPort`	30020 (WebSocket)
`websocket`	true
`passphrase`	secret, stored in the OS credential store
`preset`, `eventProperty`, `functions`, `properties`, `sliders`	

ACTION	PARAMETERS
`unreal.function.call`	`name`, optional `parameters`. The return value is in `unreal.function.lastResult`
`unreal.property.set`	`name`, `value` (converted to the property's type and clamped to its range)
`unreal.property.setSlot`	`slot`, `value`
`unreal.property.get`	`name`
`unreal.function.callObject`	`objectPath`, `function`, `parameters` (any object, for scripts)
`unreal.property.setObject`	`objectPath`, `property`, `value`
`unreal.info.refresh`	Read the preset again
`unreal.info.ping`	The setup check

The module checks `GET /remote/info` every 5 s to tell whether Unreal is there.

Tangle engine protocol

The Unity package and Godot addon speak the Tangle engine protocol, version 1. You only need it to write a plugin for another engine or to debug.

- **TCP**: the game is the server, Tangle connects to it. Default port 9420. Several clients may connect at once (a main and a backup Tangle).
- Messages are UTF-8 JSON objects, one per line, ended by `\n`.
- Each client message is a request with a numeric `id` and an `op`; the game answers each with one reply carrying the same `id`: `{"id": 3, "ok": true, ...}` or `{"id": 3, "ok": false, "error": "unknown trigger: intro2"}`.
- The game also sends pushes: messages with an `event` field and no `id`.

OP	FIELDS	REPLY	PURPOSE
`hello`	`protocol` (1), `client`, `clientVersion`, optional `token`	`protocol`, `engine`, `engineVersion`, `app`, `appVersion`	Must be first. A wrong token answers `bad token` and closes the connection
`list`		`triggers`, `properties`	What the game exposes now
`trigger`	`name`, optional `args`		Fire a trigger
`set`	`name`, `value`	`value` (after clamping and conversion)	Set a property
`get`	`name`	`value`	Read a property
`ping`		`time` (game clock, seconds)	Keep-alive; Tangle sends it every few seconds

Properties in a `list` reply have `name`, `type` (`number`, `integer`, `boolean` or `string`), `value`, optional `min` and `max`, and `writable`.

Pushes:

```json
{"event": "property", "name": "speed", "value": 2.5}
{"event": "game", "name": "level-complete", "data": {"level": 3}}
{"event": "list"}
```

`list` means triggers or properties appeared or disappeared (a scene was loaded); Tangle reads the list again.

Example session:

```
> {"id":1,"op":"hello","protocol":1,"client":"tangle","clientVersion":"1.0.0"}
< {"id":1,"ok":true,"protocol":1,"engine":"unity","engineVersion":"2022.3.20f1","app":"Lobby
Show","appVersion":"1.2"}
> {"id":2,"op":"list"}
< {"id":2,"ok":true,"triggers":[{"name":"intro"}],"properties":
[{"name":"speed","type":"number","value":1,"min":0,"max":10,"writable":true}]}
> {"id":3,"op":"trigger","name":"intro"}
< {"id":3,"ok":true}
```
TEXT

Warning: The engine protocol has no encryption. The optional token only prevents accidental control from another Tangle or a stray tool. Use it on a show network only, never expose the port to the internet, and set the bind address to `127.0.0.1` when Tangle runs on the same computer.