

TANGODOWN

SECURITY

Developer Guide

Version 0.87.0 · 06 November 2026

Free hobby project by Lars Stahl – without warranty. Current version and download: https://tangodown.eu

Contents

1 Overview: which integration for what? 4

Decision guide 4
Direction of the data 4
Common principles 4

2 Plugins – development help 5

What is a plugin? 5
Security first: the plugin sandbox 5
Permissions as needed: network, keys, files 5
Where are plugins stored? 7
Step by step: your first plugin 7
plugin.json – reference 7
Input 8
Output 8
Template: Python 9
Template: JavaScript 9
Template: local web service with FastAPI 9
Where do plugins appear? 11
Testing and troubleshooting 11
Approval and logging 11
Good practice 11
Publishing plugins in the marketplace 11

3 Connecting registers & protected databases 12

The setup form (" + Set up source") 13
Example 1: agency system with client certificate (mTLS) 13
Example 2: POST with JSON body and token 13
Example 3: internal PostgreSQL database (read-only) 13
What TangoDown enforces 14

4 REST API 14

What does it do for you? 14
Turning it on 14
Address and authentication 15
Endpoints 15
Custom databases (named, cross-case lists) 16
Examples 16
Error codes 18
Protection 18

5 **MCP – connecting AI assistants** 18
What is MCP and what does it do for you? 18
Setting it up 18
Tools 20
Example requests to the assistant 20
Protection and data privacy 20

6 **Cyber intelligence: data source keys** 20

7 **Custom API connectors (without programming)** 21
What do they do for you? 21
Setting it up 21
Entering the key correctly 21
Connector or plugin? 21

8 **Other ways to extend** 21

9 **Updates** 22

10 **Security and law – short version for developers** 22

TangoDown Security – Developer Guide

This documentation describes how to extend and connect TangoDown with your own programs: **plugins** (your own transforms), the **REST API** (your own scripts and programs), **MCP** (AI assistants such as Claude Code) and **custom API connectors** (web services without programming). All paths go through the same safeguards as the user interface: case reference, purpose limitation, approvals and audit log.

> 🍼 **Still in development and testing:** TangoDown is a free hobby project developed by a single person in their spare time. The software is still being tested – there are certainly still bugs and a lot to improve, and some areas or functions may not yet work fully. Feedback is very welcome and helps to improve things in a targeted way: ideally with the version, the exact steps leading to the error (reproducible), what you expected and what happened instead, preferably with a screenshot and the diagnostic report ("Help & info → Contact → 🩺 Create diagnostic report") – to info@tangodown.eu or via the contact form on tangodown.eu. If you would like to support the work, you are welcome to do so via crypto or PayPal (tangodown.eu → Support).

Overview: which integration for what?

Decision guide

- **You want to enrich a value (domain, IP, name …) with your own logic and see the result in the graph? → Plugin.** Via right-click on an entity, in the terminal and in the investigation autopilot. Python or JavaScript in the isolated sandbox – or a local web service (FastAPI).
- **You want to query a third-party web service that returns JSON (e.g. a company API with a key), without programming? → Custom API connector.** URL template, key from the vault, fields via JSON path.
- **You want to access the open case from outside – read, create entities, link them – e.g. from a script, a pipeline or Excel/Power Query? → REST API.**
- **You want an AI assistant (Claude Code, Claude Desktop or another MCP-capable client) to be able to read the case and – if allowed – add to it? → MCP.**
- **You want to automate recurring workflows inside TangoDown? → Terminal** (pipes, variables, macros, `replay`) and **investigation autopilot**.

Direction of the data

- **Plugins and connectors:** TangoDown calls *your* program or *a* service → the result goes into the case.
- **REST API and MCP:** *Your* program or *an AI assistant* calls TangoDown → reads or adds to the open case.

Common principles

- Everything runs locally on this PC; plugin code only in the isolated sandbox, web services and API only via 127.0.0.1.
- Only with explicit approval: plugins individually (with checksum), the API with a token, write access separately.
- Every access is recorded in the case's audit log – what leaves the application or comes into it is traceable.
- VS-NfD (German classified – restricted) cases are blocked via the normal token. For programs and AI running in-house (on-prem) there is a separate **on-prem token** that can also query VS-NfD cases (see "REST API" →

on-prem access).

- Results from plugins, connectors and AI are leads, not findings. New entities get the classification F6 (cannot be judged) until someone assesses them.

Plugins – development help

What is a plugin?

A plugin is a folder with a description **plugin.json** and code in **Python** or **JavaScript** – or the address of a **local web service**. TangoDown passes in the value of an entity and gets back new entities, links and a note.

Security first: the plugin sandbox

Python and JavaScript plugins do not run as a separate program on your PC, but isolated:

- **no file system**, **no network**, **no programs**, no windows, no devices
- no access to TangoDown, the case, the vault, the local interface or other plugins
- every run starts fresh; the time limit also ends infinite loops
- TangoDown does not start any third-party programs
- the response is strictly checked: only known entity types, text of limited length; invalid field names and control characters are removed

Python is available with the standard library – without installation and without extra packages.

Permissions as needed: network, keys, files

If a plugin needs more – say, to query a web API or evaluate a file – it declares exactly that in `plugin.json` under `permissions`. When you approve it, you see the list and confirm it; if anything in it changes, the approval is gone.

```
"permissions": {
  "net": ["api.beispiel.org", "*.beispiel.net"],
  "secrets": ["shodan", { "name": "beispiel", "hosts": ["api.beispiel.org"] }],
  "files": "read"
}
```

- **net**: host names (without `https://` and path; `*.` for subdomains). The plugin itself still has no network – it asks TangoDown with `td.fetch(...)`. TangoDown checks the host, allows only `https`, follows no redirects, goes through the configured routing (Tor/proxy, kill switch) and writes every request to the audit log. At most 50 requests per run, 5 MB each.
- **secrets**: entries from "Vault & connectors". The code only contains the placeholder `{secret:name}` in the path, query, header or body; TangoDown inserts the key only when sending. The plugin never sees it, and it is redacted in responses too.
  - **Each key only goes to its own service.** Known entries are bound, e.g. `shodan` → `api.shodan.io`, `virustotal` → `www.virustotal.com` – even if the plugin declares other hosts. This way a plugin cannot send a key to its own server.
  - **Your own entries** need a host binding: `{ "name": "beispiel", "hosts": ["api.beispiel.org"] }` (the

hosts must also be listed under `net`). Without a binding TangoDown does not insert the key anywhere.
    - `{secret:…}` is not allowed in the host name.
    - When approving, the user sees for each key which hosts it goes to, plus a warning: the plugin uses the access on the user's behalf (quota, costs).

- **files**: `"read"` – the plugin may ask for a file with `td.readFile()`; you pick it each time in a dialog (up to 20 MB, at most 5 per run). The plugin cannot name paths and cannot write anything.
- **ID and name:** The `id` must be unique. If two plugins with the same ID are in the plugin folder, only the first runs; the second is shown as faulty. If two plugins have the same name, the right-click menu adds the author or ID.

Usage in code:

```
// JavaScript
async function transform(input) {
  const r = await td.fetch('https://api.beispiel.org/lookup?q=' + encodeURIComponent(input.value), {
headers: { Authorization: 'Bearer {secret:beispiel}' } })
  if (!r.ok) return { entities: [], message: 'HTTP ' + r.status }
  const d = r.json()
  return { entities: [], note: JSON.stringify(d).slice(0, 2000) }
}
```

```
# Python: transform as async def, then await td.fetch / td.fetch_json / td.read_file (results as
dict)
async def transform(inp):
    r = await td.fetch("https://api.beispiel.org/lookup?q=" + inp["value"], headers=
{"Authorization": "Bearer {secret:beispiel}"})
    f = await td.read_file()   # only with "files": "read" – {"name", "size", "text", "base64"}
    return {"entities": [], "note": r["text"][:2000]}
```

Without declared permissions, every request is refused. **Full permissions for plugins deliberately do not exist**: a program with full permissions would have access to all data on the workstation. If you really need that, run your code as a **local web service** (see below) – it runs visibly outside TangoDown under your responsibility and only exchanges JSON.

The isolation is designed so that a plugin can at most return a wrong answer – which you check like any lead. No software offers an absolute guarantee; so only approve plugins from a trustworthy source.

⚠ **Security boundary – local web services:** Local web services (e.g. FastAPI) are separate programs that you start yourself – outside TangoDown. TangoDown only sends them the value and reads JSON back; whatever else the service does on your PC is your responsibility. Only connect trustworthy services.

Where are plugins stored?

- In the app: **Tools** → **Plugins** → 📂 **Open plugin folder**.
- One subfolder per plugin.
- ＋ **Create examples** creates three templates and the quick guide `LIESMICH.md`:
  - `domain-teile` – JavaScript
  - `text-extrakt` – Python (finds email addresses, web addresses, IPs and phone numbers in text)
  - `fastapi-beispiel` – local web service with FastAPI

Step by step: your first plugin

1. Create a subfolder in the plugin folder, e.g. `mein-plugin`.
2. In it, create `plugin.json` and `main.py` (or `main.js`) – templates below.
3. In TangoDown, **Tools** → **Plugins** → ↻ **Rescan**.
4. ✓ **Review & approve**: check the kind, file and checksum, and confirm.
5. ▶ **Try it**: enter a value, view the result as JSON – or go straight to ⇄ **To case**.
6. After that, the plugin appears in the right-click menu of matching entities.

Every change to a file in the folder revokes the approval – after editing, review and approve again.

plugin.json – reference

```
{
  "id": "mein-plugin",
  "name": "My Plugin",
  "version": "1.0",
  "description": "What it does – appears in the plugin list",
  "author": "Name or team",
  "inputs": ["domain", "ip"],
  "kind": "python",
  "main": "main.py",
  "timeout": 30,
  "icon": "⚙"
}
```

- **id** (required): unique; letters, digits, `.`, `_`, `-`.
- **name**, **version**, **description**, **author**: display; name and version appear as the source on every result.
- **inputs**: entity types for which the plugin is offered, e.g. `domain`, `ip`, `email`, `person`, `username`, `phone`, `company`, `location`, `vehicle`, `btc`, `webcam`, `note`, custom types – or `"*"` for all.
- **kind**: `"python"` or `"js"` (sandbox) or `"http"` (local web service).
- **main** (python/js): the code file in the plugin folder (default `main.py` or `main.js`).
- **url** (http): e.g. `http://127.0.0.1:8765/transform` – only `127.0.0.1`, `localhost` or `[::1]`.
- **network** (http only): `true` if your service itself goes online; it is then blocked in air-gap mode.
- **timeout**: seconds (2–300, default 30). The first Python run takes about 1–2 seconds to start the runtime.
- **icon**: one character for the menu.
- **permissions** (python/js, optional): permissions for network, keys, files – see "Permissions as needed".

Input

Your code receives an object `input` (Python: a `dict`):

```
{
  "value": "mail.shop.example.org",
  "type": "domain",
  "label": "mail.shop.example.org",
  "props": { "registrar": "Example Registrar", "url": "https://mail.shop.example.org/" }
}
```

- **value**: the value (for entities with a `url` property the URL, otherwise the label).
- **type**: entity type; **label**: label; **props**: the entity's properties (field name → text).
- Deliberately not included: vault keys, other entities, case metadata.

Output

Return an object:

```
{
  "entities": [
    { "type": "domain", "label": "example.org", "props": { "art": "registrable domain" }, "link":
"belongs to" },
    { "type": "location", "label": "Hamburg", "geo": [53.55, 9.99], "link": "Location" }
  ],
  "note": "Free text – attached to the entity as a note with timestamp and source",
  "message": "short message for the status bar"
}
```

- **entities**: at most 200. `type` (must be an existing entity type) and `label` are required; `props` (text), `geo` ([latitude, longitude]) and `link` (label of the link) are optional.
- If an entity with the same type and the same label already exists in the case, it is only linked, not created twice.
- **note** (up to 5000 characters) and **message** are optional.

Template: Python

```
import re

def transform(inp):
    wert = inp["value"]
    ergebnisse = []
    if "." in wert:
        ergebnisse.append({"type": "domain", "label": wert.split(".", 1)[1], "link": "parent"})
    return {"entities": ergebnisse, "note": f"checked: {wert}", "message": f"{len(ergebnisse)}
matches"}
```

The Python standard library is available (re, json, datetime, hashlib, ipaddress, urllib.parse, …). Extra packages and network access are not available in the sandbox – use a local web service for that. As an alternative to `transform`, the script may also read `json.load(sys.stdin)` and output the result with `print(json.dumps(...))`.

Template: JavaScript

```
function transform(input) {
  const teile = String(input.value).split('.')
  return { entities: [], note: `Labels: ${teile.length}`, message: 'done' }
}
```

`transform` may also be `async`. `print(...)` or `console.log(...)` write to the output. No `require`, no `fetch` to the network, no files – plain JavaScript.

Template: local web service with FastAPI

```
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()

class Eingabe(BaseModel):
    value: str
    type: str
    label: str | None = None
    props: dict = {}

@app.post("/transform")
def transform(e: Eingabe):
    return {"entities": [{"type": "note", "label": f"checked: {e.value}", "link": "Result"}],
"message": "ok"}
```

Start: `py -3 -m pip install fastapi uvicorn` (once), then `py -3 -m uvicorn server:app --host 127.0.0.1 --port 8765`. plugin.json for it: `"kind": "http", "url": "http://127.0.0.1:8765/transform"`.

When to use a web service instead of sandbox code? When the plugin has to go online (its own queries), needs extra packages or models, has to keep state, or already exists as a service in the team. The service can be written in any language – as long as it accepts JSON via POST and responds with JSON.

Where do plugins appear?

- **Right-click** on matching entities (graph, inspector, tables) – with icon and name.
- **Terminal:** `plugin` lists all, `plugin <id> <wert> [--type domain]` runs one; the new entities come back as a table and can be processed further, e.g. `plugin domain-teile mail.example.org | graph new "Plugin result"`.
- **Investigation autopilot:** as a step you can deselect for matching start values (not preselected).
- **Plugin section:** "▶ Try it" for testing, "⇄ To case" for adding to the case.

Testing and troubleshooting

- **Fastest:** in the plugin section, "▶ Try it" – error messages from Python (traceback) and JavaScript appear directly.
- **Test Python locally** (if Python is installed): in a file `test.py` `from main import transform; print(transform({"value": "example.org", "type": "domain", "props": {}}))`.
- **Test FastAPI locally:** `curl -X POST http://127.0.0.1:8765/transform -H "Content-Type: application/json" -d "{\"value\":\"example.org\",\"type\":\"domain\"}"`
- **"no valid JSON":** `transform` does not return an object, or the script does not print JSON.
- **"... discarded with unknown type":** the entity type does not exist (check the spelling; create custom types first under Types & symbols).
- **"time limit exceeded":** increase `timeout` or make the work faster.
- **"service not reachable":** Is the web service running at exactly the address in `url`?
- **"plugin was changed":** approve again after every file change.

Approval and logging

- The approval applies to the exact state of all of the plugin's files (checksum). Any change → new approval.
- A plugin folder may contain at most **200 files** (not counting `__pycache__`, `.venv`, `node_modules`, `.git`) – otherwise it cannot be approved. Large dependencies do not belong in the plugin folder.
- **Audit log** per run: plugin, version, checksum, entity type and value, number of results, SHA-256 of the response, run time.
- Approving and revoking requires the administrator role (Agency edition).

Good practice

- One plugin, one task; clear names and versions (increase the version when you change something).
- Enrich results with their origin (`props.quelle`, link to the match location).
- Do not invent or "estimate" personal data; when in doubt, return nothing.
- Only use lawfully accessible sources; respect terms of use.
- Never write secrets into plugin.json or the code. If a query needs a key, use a **custom API connector** (key in the vault) or a local service with its own configuration.

Publishing plugins in the marketplace

The marketplace on **tangodown.eu** distributes plugins as `.tdplugin` – a JSON file with `plugin.json` and the

plugin's text files (only `.js`, `.py`, `.json`, `.md`, `.txt`, `.csv`, flat, at most 50 files and 2 MB):

```
{
  "format": "tdplugin/1",
  "manifest": { "id": "example.whois", "name": "Whois summary", "version": "1.0", "kind": "js",
"main": "main.js", "inputs": ["domain"] },
  "files": { "main.js": "async function transform(i) { … }", "README.md": "# Whois summary" }
}
```

This is how a plugin can get into the marketplace:

1. Finish the plugin folder, **check it yourself** (code, declared permissions, function) and test it in TangoDown.
2. Send the folder with a short description and instructions (German, English welcome) to **info@tangodown.eu**. By sending it, you confirm that you have checked the plugin yourself, are responsible for it, hold the rights to it and provide it **free of charge**.
3. Whether and when a plugin is included is at the publisher's sole discretion – there is **no entitlement** to inclusion. Included plugins are packed and **signed**.

Responsibility: The plugin's **author** alone is responsible for its code, permissions, function and consequences. The publisher does not review submitted plugins and accepts no responsibility for them – not even when they are included in the marketplace. The signature only confirms that the plugin comes from the marketplace and is unchanged; it says nothing about security or function.

Free only: TangoDown is free, and so are its plugins: plugins for TangoDown may only be passed on **free of charge** – no sale, no paywall, no paid licence. The marketplace only includes free plugins.

Open code: Users can view every plugin's code in TangoDown ("⟨⟩ Code") and in the marketplace. State in the README under which licence others may improve and share your plugin (e.g. MIT) – so the community can review and develop it further.

Blocking: The publisher can remove plugins from the marketplace at any time and block them in the application if he considers it necessary.

Offer it yourself: You can also share your plugin without the marketplace, e.g. on your own website or on GitHub – as a folder or `.tdplugin` file, likewise free of charge and at your own responsibility.

TangoDown only installs marketplace plugins whose **checksum** matches and whose **signature** matches the built-in publisher key. As always, every new version needs a new approval. A `.tdplugin` file can also be installed without the marketplace under "Plugins" → "🛒 Marketplace" → "📄 Install from file …" – for example to share your own plugin within your team.

Connecting registers & protected databases

For non-public sources your organization is authorized to use (agency systems, internal databases). Set up in the ⛨ **Registers & databases** module – without programming. Passwords, tokens and certificates are stored encrypted in the vault, never in the configuration.

The setup form (" + Set up source")

- **Type:** agency system (REST/HTTPS) or database (SQL: PostgreSQL, Microsoft SQL Server, MySQL/MariaDB).
- **Address** with `{q}` for the search value; `http://` only in your own network.
- **Login:** client certificate (mTLS), token, user/password or none. Certificate (PFX/P12) and password via "📜 Certificate …" – they go into the vault, never into the configuration.
- **SQL:** one read-only query with `:q`; write commands, multiple statements and system procedures are rejected.
- **Apply fields:** one line each `Label = path` – for REST a JSON path (`treffer[0].halter.name`), for SQL the column name.
- **For entity types:** for which entities the source appears in the right-click menu.
- **Test connection:** checks reachability, TLS and certificate without a search value – no personal data is queried.

Example 1: agency system with client certificate (mTLS)

- Address: `GET https://fachverfahren.intern/api/v2/halter?kennzeichen={q}`
- Login: "Client certificate (mTLS)" → "📜 Certificate …" (PFX/P12) and "Cert. password …"
- Headers (optional): `X-Dienststelle: Musterstadt`
- Apply fields:

```
Owner = treffer[0].halter.name
Address = treffer[0].halter.anschrift
Vehicle = treffer[0].fahrzeug.typ
```

Expected response (example): `{"treffer": [{"halter": {"name": "…", "anschrift": "…"}, "fahrzeug": {"typ": "…"}}]}`. TangoDown takes the first array in the response as the rows (or the object itself).

Example 2: POST with JSON body and token

- Address: `POST https://register.beispiel.org/suche`, login "Token (Bearer)" → "🔑 Token …"
- Body: `{"suchwert": "{q}", "art": "person"}` – {q} is inserted JSON-safely

Example 3: internal PostgreSQL database (read-only)

```
SELECT p.name, p.geburtsdatum, a.strasse || ', ' || a.ort AS anschrift
FROM personen p JOIN anschriften a ON a.person_id = p.id
WHERE p.kennung = :q
```

Fields: `Name = name`, `Date of birth = geburtsdatum`, `Address = anschrift`. Create a database account with read-only permissions, e.g.:

```
CREATE ROLE tangodown_leser LOGIN PASSWORD '…';
GRANT CONNECT ON DATABASE register TO tangodown_leser;
GRANT SELECT ON personen, anschriften TO tangodown_leser;
```

Microsoft SQL Server: the same query with :q (TangoDown inserts it as parameter @q, at most 200 rows); MySQL/MariaDB likewise.

What TangoDown enforces

- Encrypted database connections verify the **server certificate** (default). Only for servers in your own network (e.g. `db.intern`, 10.x, 192.168.x) can you tick "Do not verify certificate (internal)" – for instance with self-signed certificates. For servers on the internet, this is rejected.
- Search values only go into the query as parameters (no SQL injection), only SELECT/WITH, one statement, read-only transaction, 20 s time limit, 200 rows.
- Not allowed (rejected when saving): comments (`--`, `/* */`, `#`), `INTO`, system procedures and functions (e.g. `xp_…`, `pg_read_file`, `OPENROWSET`, `dblink`), `SLEEP` / `WAITFOR`, `SET`, `LOCK`, `FOR UPDATE`. Text in quotes is not checked for this – `WHERE notiz = 'a;b'` works.
- Example that works: `SELECT halter, marke FROM fahrzeuge WHERE kennzeichen = :q` · Example that is rejected: `SELECT * INTO kopie FROM fahrzeuge WHERE kennzeichen = :q`.
- Every query: legal basis + justification, stored in the case with a checksum, in the audit log; optional four-eyes approval per source (Agency edition).
- Plain-text keys in headers are rejected (use `{key}` or the login type).

REST API

What does it do for you?

Your own scripts, pipelines or tools can **read the open case** (case info, entities, links, audit log) and – if approved – **create entities and links**. Typical uses: pushing results from your own tools into the case, exporting case data for analyses, generating reports in other systems, bulk imports.

Turning it on

1. **Tools → Vault & connectors → Local interface · REST & MCP**.
2. Turn on "active"; change the port (default **47821**) if needed.
3. ⧉ **curl example** copies a ready-made call with token to the clipboard.
4. Only turn on **Write access** if programs are supposed to create data (separate confirmation).

The token is stored encrypted and can be renewed there ("Renew"). It does not belong in scripts that are passed on – better read it from an environment variable.

Protection of the interface:

- Reachable only on this computer (`127.0.0.1`), not from the network. Other users and programs on the same computer can reach the port, though – so the **token** is what matters, not the port number.
- Every request needs the token (random, 192 bits); it is compared in constant time.
- Requests with a foreign host name (protection against DNS rebinding) and from web pages in the browser (`Origin` header) are rejected.
- **Lock after failed attempts:** After 10 requests with a wrong or missing token within one minute, the interface rejects **all** requests for one minute (HTTP 429) – even with the correct token. TangoDown reports

this, and it is recorded in the audit log of the open case. Restarting the interface in the settings lifts the lock. If in doubt, renew the token.

- The settings show how many requests were rejected.
- "⧉ copy" puts the token into the clipboard and clears it again after 60 seconds if the same text is still there.
- Reading only with the token; writing additionally only with its own approval; VS-NfD cases only with the on-prem token. Every query is recorded in the audit log.
- The connection is unencrypted HTTP but does not leave the computer (loopback).

Address and authentication

- Base: `http://127.0.0.1:47821/api/v1`
- Header: `Authorization: Bearer <TOKEN>`
- Responses: JSON, UTF-8.

Endpoints

- `GET /api/v1/case` – case number, title, purpose, legal basis, scope of the open case.
- `GET /api/v1/entities?type=domain&q=suchtext&limit=200` – list entities (type and search optional, at most 500).
- `GET /api/v1/entities/<id oder Bezeichnung>` – one entity with properties, source, classification, notes, links.
- `GET /api/v1/links` – all links.
- `GET /api/v1/audit?limit=50` – latest audit log entries.
- `GET /api/v1/types` – all entity types (built-in and custom) with field templates (key, label, field kind).
- `GET /api/v1/search?q=suchtext&limit=50` – search in the case (label, properties, source, notes), with a short excerpt.
- `GET /api/v1/export` – **Give data:** the whole case as a package (all entities with properties/coordinates and all links) for further processing.
- `POST /api/v1/entities` (write access) – create an entity. Body: `{ "type": "domain", "label": "example.org", "rel": "C3", "source": "my script", "props": { "notiz": "…" }, "notes": "…" }`
- `PATCH /api/v1/entities/<id>` (write access) – change an entity. Body (all optional): `{ "label": "…", "rel": "B2", "props": { "port": "443", "altesFeld": null }, "geo": [53.55, 9.99], "notes": "…" }`. In `props`, a value sets the field, `null` removes it; `geo: null` removes the coordinates. (Also via `POST` to the same path.)
- `POST /api/v1/links` (write access) – Body: `{ "source": "<id>", "target": "<id>", "label": "belongs to" }`
- `POST /api/v1/ingest` (write access) – **Receive data:** several entities and links at once (bulk import from another tool). Links refer to the newly created entities via `"#0"`, `"#1"` … (in order) or via an existing ID or exact label.

Created entities carry the source "API · <program>" and, if not specified, the classification F6; every change appears immediately in the user interface and in the audit log.

> **Where data is imported:** `POST /entities`, `POST /links` and `POST /ingest` write to the **open case** (the entity network in the graph). You address the cross-case **custom databases** (e.g. a named vehicle list) separately via the `databases` endpoints (below) – there you target a specific database as the destination/source. You connect protected sources via **Registers & databases** (connector). During matching (e.g. vehicle), the custom databases are searched for the search value automatically.

Custom databases (named, cross-case lists)

Separate from the case graph – ideal for filling e.g. a vehicle or person list by script and querying it specifically. The database is addressed **by name or ID** (name not case-sensitive).

- `GET /api/v1/databases` – all custom databases (name, kind, fields, count).
- `GET /api/v1/databases/<Name|id>/entries?q=suchtext&limit=200` – entries of a specific database, optionally a substring search across key and fields.
- `POST /api/v1/databases/<Name|id>/entries` (write access) – add an entry. Body: `{ "key": "HH-AB 000", "fields": { "Owner": "Max Muster" }, "note": "…" }` (`key` = key value: license plate, name, phone number …).
- `GET /api/v1/databases/search?q=wert` – search for a value across **all** custom databases (like matching), stating the database and exact/partial match.

Example (Python): write to the vehicle list and query it specifically

```
BASE = "http://127.0.0.1:47821/api/v1"
requests.post(f"{BASE}/databases/Vehicles/entries", headers=H,
    json={"key": "HH-AB 000", "fields": {"Owner": "Max Muster", "Type": "Car"}})
treffer = requests.get(f"{BASE}/databases/Vehicles/entries", headers=H, params={"q": "HH-
AB"}).json()
print(treffer["db"]["name"], len(treffer["entries"]), "matches")
```

Examples

PowerShell:

```
$h = @{ Authorization = "Bearer $env:TD_TOKEN" }
Invoke-RestMethod -Uri http://127.0.0.1:47821/api/v1/entities?type=domain -Headers $h
```

Python:

```
import os, requests

H = {"Authorization": f"Bearer {os.environ['TD_TOKEN']}"}
BASE = "http://127.0.0.1:47821/api/v1"

print(requests.get(f"{BASE}/case", headers=H).json())

# Create an entity and add a property afterwards
neu = requests.post(f"{BASE}/entities", headers=H,
    json={"type": "ip", "label": "192.0.2.10", "source": "Import"}).json()
requests.patch(f"{BASE}/entities/" + neu["id"], headers=H,
    json={"props": {"offene ports": "443, 8443"}, "geo": [53.55, 9.99]})

# Receive data: take over matches from your own tool as a package
requests.post(f"{BASE}/ingest", headers=H, json={
    "entities": [
        {"type": "company", "label": "Example Inc.", "rel": "B2"},
        {"type": "domain",  "label": "beispiel.example", "props": {"quelle": "my crawler"}},
    ],
    "links": [{"source": "#0", "target": "#1", "label": "Website"}],  # #0 -> #1
})

# Give data: export the whole case
fall = requests.get(f"{BASE}/export", headers=H).json()
print(len(fall["entities"]), "entities,", len(fall["links"]), "links")
```

Bulk import with curl:

```
curl -X POST http://127.0.0.1:47821/api/v1/ingest \
  -H "Authorization: Bearer $TD_TOKEN" -H "content-type: application/json" \
  -d '{"entities":[{"type":"ip","label":"198.51.100.7","props":{"asn":"AS64500"}}]}'
```

curl:

```
curl -H "Authorization: Bearer <TOKEN>" http://127.0.0.1:47821/api/v1/case
```

Error codes

- **401** token missing or wrong · **403** host not allowed, browser access, VS-NfD case without on-prem token or no write access · **404** unknown path · **409** no case open.

Protection

- Only reachable on this PC (`127.0.0.1`).
- Requests from web browsers are rejected – no website can read out the case.
- Off by default; write access separate; VS-NfD cases only with the on-prem token; every access in the audit log.

Existing elements (never overwrite silently): `create_entity` or `POST /entities` and the import only create a new element if none with the same type and name exists. Otherwise the existing one is reused: empty fields are filled, differing values are **not** overwritten. The response reports this with `existing: true` , `filled` (filled fields) and `conflicts` (`field` , `old` , `next`); the import reports `entitiesExisting` and `conflicts` . Differences are also recorded in the audit log.

On-prem access (VS-NfD): Under Vault & connectors → Local interface, switch on "On-prem access". TangoDown creates a **second token**. Whoever queries with this token may also read VS-NfD cases (and write, if write access is enabled). The normal token stays blocked for VS-NfD – assistants that send data to the cloud are often connected to it.

- Only enter the on-prem token in programs that run entirely in-house, e.g. a local language model (Ollama, vLLM) with its own MCP client, or your own scripts.
- "⧉ MCP JSON (on-prem)" and "⧉ curl (on-prem)" copy the configuration with this token.
- Every access is recorded in the audit log with the note `[On-Prem-Token]` . Switching it off invalidates the token immediately.

```
# Example: query a VS-NfD case with the on-prem token
curl -H "Authorization: Bearer <on-prem token>" http://127.0.0.1:47821/api/v1/case
```

MCP – connecting AI assistants

What is MCP and what does it do for you?

The **Model Context Protocol (MCP)** is an open standard that lets AI assistants use tools. Via the MCP interface, for example **Claude Code** or **Claude Desktop** can read the open case, answer questions about it, find connections, draft reports – and, if approved, create entities, links and notes. The assistant always works through TangoDown: with purpose limitation and audit log.

Setting it up

1. Turn on the local interface (see REST API).
2. ⧉ **Claude Code command** copies the ready-made command, e.g.:

```
claude mcp add --transport http tangodown http://127.0.0.1:47821/mcp --header "Authorization: Bearer
<TOKEN>"
```

1. For other clients, copy **MCP JSON** and enter it in their configuration:

```
{
  "mcpServers": {
    "tangodown": {
      "type": "http",
      "url": "http://127.0.0.1:47821/mcp",
      "headers": { "Authorization": "Bearer <TOKEN>" }
    }
  }
}
```

Transport: Streamable HTTP (JSON responses).

Tools

Read-only (always):

- `case_info` – case, purpose, legal basis (the assistant should call it first: the purpose limits the research).
- `list_entities` – entities, optionally by type and search text.
- `list_entity_types` – types with field templates.
- `get_entity` – one entity in full.
- `list_links` – links.
- `audit_log` – latest entries.
- `search` – search in the case (label, properties, source, notes), short matches with excerpt.
- `export_case` – **Give data:** the whole case as a package (entities + links) for further processing.
- `list_databases` – list custom (cross-case) databases.
- `query_database` – search a **named** custom database (e.g. "Vehicles").
- `search_databases` – search for a value across **all** custom databases.

Writing (only with write access):

- `create_entity` – new entity (classification F6, source "MCP").
- `create_link` – link two entities.
- `add_note` – attach a note.
- `update_entity` – change an entity: set/remove properties (`props` , value `null` removes a field), label, classification, coordinates `[lat, lng]` (`null` removes them), notes.
- `ingest` – **Receive data:** several entities and links at once. Links refer to the newly created entities via `"#0"` , `"#1"` … or via an existing ID/label.
- `add_database_entry` – add an entry to a **named** custom database (e.g. a vehicle to "Vehicles"): `key` = key value, `fields` = further columns.

Example requests to the assistant

- "Summarize the case and list open questions."
- "Which domains and IPs are connected? Draw me the paths."
- "Check the notes for contradictions."
- "Create entities for the three companies mentioned and link them to the person." (write access)
- "Take this JSON into the case as entities and links." (ingest, write access)
- "Export the case and make me a table of the IPs with open ports from it." (export_case)

Protection and data privacy

- What the assistant reads goes to its provider (with cloud AI). Every MCP access is recorded in the audit log.
- VS-NfD cases only with the on-prem token – give it only to an assistant that runs entirely in-house (local model, no cloud).
- AI statements are leads; check the evidence.

Cyber intelligence: data source keys

The ◎ **Cyber intelligence** module passively queries several services about an IP/domain. The keys are stored in

the vault (Tools → Vault & connectors), one per provider; "Test" checks it:

- **GreyNoise** – Community API (free, ~50/week): key from your GreyNoise account.
- **AbuseIPDB** – free up to 1000 checks/day: API key from your account.
- **VirusTotal** – Public API (4/min, 500/day, non-commercial): API key.
- **urlscan.io** – search free (public scans are visible); key optional for more quota.
- **Netlas** – free with limit; for security integrations, Netlas offers a **Developer License** (on request).
- **Censys** – store the key as "API-ID:Secret" (TangoDown builds the Basic Auth from it).
- **Shodan InternetDB** – no key (IPv4 only).

All queries are purely read-only (the services have already scanned); TangoDown sends nothing to the target systems. Results are merged, contradictions are marked; every query is recorded in the audit log with a checksum. Notes on redistribution: check for each provider whether the free API may be used only personally or also in software that is passed on.

Custom API connectors (without programming)

What do they do for you?

A connector connects a web service that returns JSON – e.g. a company, phone or threat-data API. It appears as a right-click transform for the selected entity types; the fields from the response are taken over as properties.

Setting it up

- **Tools → Vault & connectors → Custom connectors → + New connector**.
- **URL template** with `{q}` for the value, e.g. `https://api.example.org/v1/lookup?q={q}`.
- **Headers** may contain `{key}` – the key is stored in the vault, never in the connector.
- **Fields:** label = JSON path, e.g. `Land = data.country`, `Ports = ports`.
- "Test" shows the response before the connector becomes active.

Entering the key correctly

The header only contains the placeholder, e.g. `Authorization: Bearer {key}`; the actual key goes into the "Key" field and is stored encrypted in the vault. A plain-text key in the header is rejected (it would otherwise appear in the configuration and screenshots) – in that case, renew the key with the provider.

Connector or plugin?

- **Connector:** one web service, one query, take over fields – without code, key in the vault, network route and kill switch apply.
- **Plugin:** your own logic, several steps, new entities and links, offline analyses, your own libraries or models.

Other ways to extend

- **Terminal:** commands with pipes over typed objects, variables, background jobs, macros, `replay`, redacted

transcript. Ideal for recurring workflows inside TangoDown.

- **Investigation autopilot:** fixed workflow per start value, steps after approval – including matching plugins.
- **Custom types and fields:** Tools → Types & symbols; custom types can be listed in the `inputs` of plugins.
- **Custom databases:** named, cross-case lists with their own fields, value search; automatically included in matching (e.g. vehicle, person). Fill them in the module (also CSV import) or via the `databases` endpoints of the REST API or the MCP tools.
- **Case package (.tdpack):** encrypted export and import of whole cases between workstations.

Updates

TangoDown only checks for updates on request ("Help & info" → "⇪ Check for updates") and never installs anything itself. The version file comes from **tangodown.eu** and is **signed** by the publisher; TangoDown verifies the signature with its built-in public key. If it does not match, no download is offered.

After downloading, "✓ Check download" (SHA-256) and only start it if it says "Checksum matches". Portable: put the new EXE in the same place – the data folder next to it stays.

Why a signature instead of just a checksum: whoever takes over a website could swap file and checksum together. Without the publisher's private key they cannot create a valid signature. MD5 is deliberately not used (considered broken).

Security and law – short version for developers

- Only lawfully publicly accessible sources, and data within the scope of the case purpose.
- No secrets in code or configuration; keys belong in the vault.
- Nothing that circumvents access protection (no login bypassing, no scraping against terms of use, no unsecured private cameras).
- Results are leads, not findings – label where they come from.
- Everything that comes into the case or leaves it is recorded in the audit log – including access via plugins, REST API and MCP. The only exception: cases that were explicitly created "without logging" in the PI edition; they are marked and have no audit trail. In the Agency edition, this exception does not exist.