VECTROLINK
TECHNOLOGIES

vMT-02

Industrial Modbus Translator & RTU/TCP Converter

API Reference

HTTP JSON API for System Integrators

Firmware version: 0.3.1

Document version: 1.4

November 12, 2026

Confidential — intended for the customer and integrators of this device.

Contents

1 Overview & Conventions **3**
- 1.1 Connecting . . . 3
- 1.2 Response Envelope . . . 3
- 1.3 HTTP Status Codes . . . 4
- 1.4 Field Naming and Types . . . 4

2 Authentication & Sessions **5**
- 2.1 Login Flow . . . 5
  - 2.1.1 POST /api/auth/login . . . 5
  - 2.1.2 Using the token . . . 5
  - 2.1.3 POST /api/auth/logout . . . 6
  - 2.1.4 GET /api/auth/session . . . 6
- 2.2 User Accounts . . . 6
  - 2.2.1 GET /api/auth/users . . . 8
  - 2.2.2 POST /api/auth/users – create a user . . . 8
  - 2.2.3 PUT /api/auth/users – edit a user . . . 9
  - 2.2.4 DELETE /api/auth/users . . . 9
  - 2.2.5 POST /api/auth/password – change a password . . . 9

3 Configuration & Status **10**
- 3.1 GET /api/config . . . 10
  - 3.1.1 Degraded fallback response . . . 10
- 3.2 POST /api/config . . . 11
- 3.3 GET /api/status . . . 11
- 3.4 GET /api/version . . . 12
- 3.5 POST /api/system/restart . . . 13
- 3.6 POST /api/system/firmware_update_allowed . . . 13

4 Mappings **14**
- 4.1 GET /api/mappings . . . 14
  - 4.1.1 Paging . . . 15
- 4.2 POST /api/mappings – create a mapping . . . 15
- 4.3 PUT /api/mappings – edit a mapping . . . 17
- 4.4 Deleting mappings . . . 18
  - 4.4.1 POST /api/mappings/delete . . . 18
  - 4.4.2 POST /api/mappings/clear . . . 18

5 Transparent Mappings **19**
- 5.1 GET /api/transparent_mappings . . . 19
- 5.2 POST /api/transparent_mappings – create . . . 20
- 5.3 PUT /api/transparent_mappings – edit . . . 20
- 5.4 DELETE /api/transparent_mappings . . . 21

6 Manual Read **22**
6.1 GET /api/manual_read . . . 22

7 Network & Translator Control **25**
7.1 GET /api/network . . . 25
7.2 POST /api/network . . . 25
7.3 GET /api/wifi/scan . . . 26
7.4 Saved Wi-Fi Networks . . . 26
7.4.1 GET /api/wifi/networks . . . 26
7.4.2 POST /api/wifi/networks – add . . . 27
7.4.3 DELETE /api/wifi/networks/{index} . . . 27
7.5 Translator Controls . . . 27
7.5.1 POST /api/translator/enable . . . 27
7.5.2 POST /api/translator/reload . . . 28
7.5.3 POST /api/translator/mode . . . 28
7.5.4 GET /api/translator/stats . . . 28

8 Enum Reference **29**
8.1 data_type . . . 29
8.2 byte_order . . . 29
8.3 modbus_function . . . 30
8.4 port (port_type_t) . . . 30
8.5 protocol (modbus_protocol_t) . . . 30
8.6 mapping_mode (mapping_mode_t) . . . 30
8.7 wifi_mode . . . 31
8.8 HTTP status vs. body `status` – quick reference . . . 31

Chapter 1

Overview & Conventions

Every function available in the vMT-02's web interface is also available as a JSON HTTP API on the device itself, so you can integrate the gateway directly with your own systems – a SCADA platform, a provisioning pipeline, a fleet-management tool, or a custom script – without going through Vectrolink Cloud at all.

This document is a precise, example-driven reference for that API: every endpoint, its exact request and response bodies, and the handful of gotchas you'll hit in practice. It does not cover firmware update (OTA) or any superadmin-only maintenance endpoints – those are internal/administrative and out of scope for this datasheet.

1.1 Connecting

All endpoints are served over HTTPS directly by the device, on whatever IP address it has on your network (Ethernet or Wi-Fi). The device uses a self-signed certificate, so your HTTP client or script needs to either trust that specific certificate or disable certificate verification for this host – for example, `curl -k` for quick testing, or your language's HTTP library equivalent for production code.

```
curl -k https://<device-ip>/api/version
```

`GET /api/version` (Section 3.4) is the one endpoint in this whole API that needs no session token, so it's the right quick check that you can reach the device at all before dealing with authentication (Chapter 2). `GET /api/status` looks similar but requires a token as of 0.1.21 – see Section 3.4 for why.

1.2 Response Envelope

There is no single universal response wrapper – read-only `GET` endpoints that return data (mappings, status, configuration) return that data directly as a JSON object or array. Write endpoints (`POST`/`PUT`/`DELETE`) that perform an action rather than return data respond with a small status object:

```
{"status": "ok"}
```

or, on failure:

```
{"status": "error", "message": "<human-readable description>"}
```

Always check the HTTP status code first, then the `status` field if present. A 200 response with `"status":"error"` in the body can occur for some endpoints – don't assume 200 alone means success.

1.3 HTTP Status Codes

Code	Meaning
200	Success.
400	Bad request – missing/invalid JSON, missing required field, or a value outside its valid range.
401	No `Authorization` header, or the session token is missing, invalid, or expired.
403	Authenticated, but this account isn't allowed to perform this action (e.g. a non-`superadmin` account calling a user-management endpoint, or trying to log in with an account that doesn't have the requested access mode – see Section 2.1.1).
404	The referenced resource (user, saved Wi-Fi network, . . .) doesn't exist.
413	Request body too large (the `/api/config` endpoint caps the body at 64 KB).
500	Internal error on the device.
503	Device temporarily out of memory – retry after a short delay.

1.4 Field Naming and Types

- All JSON keys are `snake_case`.
- Enumerated fields (data type, byte order, protocol, port, function code, . . .) are documented per-section as you encounter them, with the complete, authoritative mapping table collected in Appendix 8.
- Where an enumerated field accepts both an integer and a string form, both are shown – but note that **not every endpoint's response uses the same convention for the same-looking field**; see the warning in Section 4.1 for a concrete case where this matters.
- Timestamps are Unix epoch seconds unless a field name explicitly says otherwise (e.g. `uptime` is milliseconds).
- IP addresses are plain dotted-quad strings, e.g. `"192.168.1.50"`.

This reference was written directly against the firmware's source code, but the API is not formally versioned. If a response includes fields not documented here, ignore unknown fields rather than failing – this is the safest general integration practice against any embedded HTTP API.

Chapter 2

Authentication & Sessions

2.1 Login Flow

1. **Log in** to obtain a session token.
2. **Send the token** on every subsequent request in the `Authorization` header.
3. **Log out** (optional) to invalidate the token early, or just let it expire.

Sessions expire automatically after **1 hour of inactivity**; each successful authenticated request renews the timer. There's no refresh-token mechanism – simply log in again once a request comes back `401`.

2.1.1 POST /api/auth/login

Auth required: none.

```
POST /api/auth/login
Content-Type: application/json

{"username": "user", "password": "user"}
```

Success (200):

```
{"status": "ok", "token": "a1b2c3...64 hex chars", "username": "user"}
```

Errors: 400 (missing username/password), 401 (wrong credentials), 403 (see below).

Optional `access` field. Add `"access": "api"` to the login body if you only want to verify this account has *API access* (rather than *web* access, which is checked by default). This matters because an account can have one without the other – see the `web_access`/`api_access` flags in Section 2.2. If the account lacks the access mode being requested, login fails with `403` even though the password is correct.

2.1.2 Using the token

Attach it as a Bearer token on every request:

```
curl -k https://<device-ip>/api/status \
     -H "Authorization: Bearer a1b2c3...64 hex chars"
```

The device also accepts the raw token without the `Bearer` prefix, but sending it with the prefix is the conventional and recommended form.

The `web_access`/`api_access` flags on an account are only checked at *login* time (which mode you request via the `access` field). Once you hold a valid token, it works identically against every endpoint in this document regardless of those flags – they gate *how you can log in*, not what a given session can subsequently call.

2.1.3 POST /api/auth/logout

Auth required: any authenticated session.

```
POST /api/auth/logout
Authorization: Bearer <token>
```

Success (200): `{"status": "ok"}`. Invalidates the token immediately; no request body needed.

2.1.4 GET /api/auth/session

Auth required: any authenticated session. Use this to check whether a token you're holding is still valid, and to find out what the account can do.

```
GET /api/auth/session
Authorization: Bearer <token>
```

Success (200):

```
{
  "status": "ok",
  "authenticated": true,
  "username": "user",
  "is_admin": true,
  "is_superadmin": false
}
```

Expired/invalid (401):

```
{"status": "error", "message": "Invalid or expired session", "
   authenticated": false}
```

2.2 User Accounts

The device ships with a small, fixed set of accounts you can use directly or extend:

Username	Default password	Web access	API access	Notes
superadmin	Not documented here	Yes	Yes	Full control; hidden from GET /api/auth/users entirely; can't be edited/deleted via the API at all. Reserved for Vectrolink support/administrative use – not intended for integrators, and its credentials are out of scope for this reference.
user	user	Yes	Yes	General-purpose account for day-to-day web UI use.
api	api	No	Yes	Meant for integrations – change its password before relying on it.

Change the default passwords before deploying the device. user/user and api/api are well-known defaults – identical on every device of this model until changed, not device-specific.

You can create up to **5 additional managed accounts** beyond these built-in ones (see the X-Max-Managed-Users / X-Managed-Users-Count response headers on GET /api/auth/users to check how many you have left).

Auth required: any authenticated session for all four endpoints below, but what you can actually do depends on whether you're superadmin:

- **Creating/editing a user with is_admin: true requires superadmin.** A non-superadmin caller can still call POST/PUT, but is_admin in the request body is silently ignored and forced to false – this does *not* fail with an error, it just doesn't grant what you asked for. Check the is_admin field in the response of a follow-up GET if you need to confirm what actually landed.
- **Only superadmin can touch the built-in accounts** – superadmin, cloud, user, and api themselves. A non-superadmin session gets 403 Forbidden ("Cannot manage/delete system account") if it tries to PUT/DELETE any of these four usernames. Creating a user with one of these names always fails anyway (they already exist) regardless of caller.
- Anything else – creating, editing, or deleting one of the up to **5 additional managed accounts** – works for any authenticated session, superadmin or not.

In short: any account can manage plain managed users, but only superadmin can grant admin privileges or touch the four built-in accounts. is_admin itself is otherwise informational – shown back to you in GET /api/auth/users/GET /api/auth/session, but nothing else in this API reads it.

2.2.1 GET /api/auth/users

```
GET /api/auth/users
Authorization: Bearer <admin-token>
```

Success (200) – a JSON array, plus two response headers:

```
X-Max-Managed-Users: 5
X-Managed-Users-Count: 2

[
  {
    "username": "user",
    "is_admin": true,
    "web_access": true,
    "api_access": true,
    "manageable": true,
    "created_at": 1720000000
  },
  {
    "username": "api",
    "is_admin": false,
    "web_access": false,
    "api_access": true,
    "manageable": true,
    "created_at": 1720000001
  }
]
```

created_at is a Unix timestamp. The built-in `superadmin` account never appears in this list under any circumstances. `manageable` reflects what *you*, the caller, are allowed to do with that entry: `true` for any of the 5 additional managed accounts, and for the `user`/`api`/`cloud` built-ins only if you're calling as `superadmin` – for a non-superadmin caller, `user` and `api` still appear in the list (unlike `cloud`, which is hidden entirely unless you're `superadmin`) but with `manageable: false`, since `PUT`/`DELETE` against them will 403.

2.2.2 POST /api/auth/users – create a user

```
POST /api/auth/users
Authorization: Bearer <admin-token>
Content-Type: application/json

{
  "username": "integration1",
  "password": "a-strong-password",
  "is_admin": false,
  "web_access": false,
  "api_access": true
}
```

`is_admin` defaults to `false`, `web_access` and `api_access` both default to `true` if omitted. Sending `"is_admin": true` only has an effect if you're calling as `superadmin` – see the note above.

Success (200): `{"status": "ok"}`. **Errors:** 400 (missing username/password, username already taken, or the 5-account managed-user limit reached).

For a pure machine-to-machine integration account with no interactive web login, set `"web_access": false, "api_access": true` as shown above.

2.2.3 PUT /api/auth/users – edit a user

```
PUT /api/auth/users
Authorization: Bearer <admin-token>
Content-Type: application/json

{
  "username": "integration1",
  "is_admin": false,
  "web_access": false,
  "api_access": true,
  "password": "only-set-this-field-to-change-the-password"
}
```

`username`, `is_admin`, `web_access`, and `api_access` are all required in every request (this call replaces the full permission set, it doesn't patch individual fields). `password` is optional – omit it or send an empty string to leave the password unchanged.

Success (200): `{"status": "ok"}`. **Errors:** 400 (missing required fields), 403 (attempting to edit a protected system account you don't have rights to).

2.2.4 DELETE /api/auth/users

```
DELETE /api/auth/users
Authorization: Bearer <admin-token>
Content-Type: application/json

{"username": "integration1"}
```

Success (200): `{"status": "ok"}`. **Errors:** 400 (missing username), 403 (`"Cannot delete system account"` – protected account), 404 (user not found).

2.2.5 POST /api/auth/password – change a password

Auth required: any authenticated session. There's no separate admin/non-admin distinction on this endpoint – `username` can be any account, not just your own, and any authenticated session can target any account. What actually gates this is `old_password`: the request only succeeds if it matches the *target* account's current password, which is what makes this safe to leave open rather than restricted to `superadmin` like the four endpoints above – proving you already know an account's password is treated as sufficient grounds to change it, regardless of which account you're currently logged in as.

```
POST /api/auth/password
Authorization: Bearer <token>
Content-Type: application/json

{
  "username": "user",
  "old_password": "current-password",
  "new_password": "new-password"
}
```

Success (200): `{"status": "ok"}`. **Errors:** 400 (missing fields), 401 (`old_password` incorrect for the target account).

Chapter 3

Configuration & Status

3.1 GET /api/config

Auth required: any authenticated session.

Returns the device's complete configuration – the same data you'd get from the Dashboard's *Export* feature – as a single JSON object: master/slave port setup, transparent-mode and translator enable flags, the global mapping mode, and the full arrays of mappings and transparent mappings.

```
GET /api/config
Authorization: Bearer <token>
```

> The exact shape of this response is large and mirrors the mapping/transparent-mapping structures documented in Chapters 4 and 5 – rather than duplicate it here, use `GET /api/mappings` and `GET /api/transparent_mappings` directly if you only need those pieces; use this endpoint when you want to snapshot or back up the entire configuration in one call.

3.1.1 Degraded fallback response

If the device cannot build the full configuration JSON (for example, under memory pressure), it falls back to a smaller, flat summary instead of failing outright:

```
{
  "status": "degraded",
  "message": "Full configuration unavailable, returning minimal config
     ",
  "device_name": "vMT-02-A1B2",
  "serial_number": 123456,
  "system_version": 256,
  "config_timestamp": 1720000000,
  "master_port": "UART1",
  "slave_port": "UART2",
  "master_port_type": 0,
  "slave_port_type": 1,
  "transparent_mode_enabled": false,
  "translator_enabled": true,
  "firmware_update_allowed": true,
  "mapping_count": 12,
  "transparent_mapping_count": 0
}
```

Always check the `status` field on this endpoint. `"status": "degraded"` (still returned with HTTP 200) means you got this reduced summary, not the full configuration – the mapping and transparent-mapping *arrays* are absent, only their *counts* are present. `master_port_type`/`slave_port_type` are the integer `port_type_t` values – see Appendix 8.

3.2 POST /api/config

Auth required: any authenticated session.

Uploads and applies a complete configuration, replacing the device's current one. This is the same operation the Dashboard's *Import* feature performs.

```
POST /api/config
Authorization: Bearer <token>
Content-Type: application/json

{ ... full configuration object, as returned by GET /api/config ... }
```

Success (200):

```
{"status": "ok", "message": "Configuration applied successfully"}
```

Errors: 400 (empty body, or the body isn't valid/parseable configuration JSON), 413 (body larger than 512 KB).

This endpoint applies the configuration immediately – it does *not* restart the device on its own. If your workflow expects a restart after importing configuration (some earlier tooling assumed this), call `POST /api/system/restart` explicitly afterwards.

3.3 GET /api/status

Auth required: any authenticated session. If all you need is a lightweight liveness/version check without a session, use `GET /api/version` below.

Returns a large snapshot of runtime health: firmware version, uptime, last reset reason, Modbus master/slave counters, register-cache hit ratio, network state, and mapping capacity. This is the endpoint to poll for a dashboard or health check.

```
GET /api/status
```

Success (200):

```
{
  "status": "running",
  "version": "0.3.1",
  "build_date": "Nov 12 2026 10:41:06",
  "uptime": 8453210,
  "last_reset_reason": "poweron",
  "crash_count": 0,
  "security": {
    "secure_boot": true,
    "flash_encryption": "release",
    "uart_download_disabled": true,
    "jtag_disabled": true
  },
  "master_running": true,
  "slave_running": true,
  "modbus_master": {
```

```
    "total_requests": 48211,
    "successful_requests": 48190,
    "timeout_errors": 18,
    "crc_errors": 3,
    "protocol_errors": 0
  },
  "modbus_slave": {
    "total_requests": 512,
    "read_requests": 498,
    "write_requests": 14,
    "error_responses": 2,
    "client_connections": 1
  },
  "cache": {
    "hits": 47850,
    "misses": 361,
    "hit_ratio": 0.9925
  },
  "wifi_connected": false,
  "wifi_ap": false,
  "wifi_sta": true,
  "wifi_mode": "STA",
  "wifi_ip": "N/A",
  "eth_connected": true,
  "eth_ip": "192.168.10.111",
  "internet_access": true,
  "config_timestamp": 1720000000,
  "translator_enabled": true,
  "firmware_update_allowed": true,
  "global_mapping_mode": 0,
  "mapping_count": 12,
  "max_mappings": 512,
  "transparent_mapping_count": 0,
  "max_transparent_mappings": 256
}
```

`uptime` is in **milliseconds**, unlike every timestamp field elsewhere in this API (which are Unix epoch seconds) – see Section 1. `wifi_ip`/`eth_ip` are `"N/A"` when that interface has no address. `global_mapping_mode` is the integer `mapping_mode_t` value – see Appendix 8.

This response does *not* include `device_name` or `serial_number` – those only appear in the `GET /api/config` degraded fallback shown above. If you need the device's name/serial as part of a status poll, you currently have to read it from `GET /api/config` instead.

3.4 GET /api/version

Auth required: none. Only exposes the two fields that aren't sensitive on their own; everything else (IP addresses, uptime, traffic counters) requires a session token.

```
GET /api/version
```

Success (200):

```
{
  "version": "0.3.1",
  "build_date": "Nov 12 2026 10:41:06"
```

```
}
```

Use this for a simple "is the device up, and what firmware is it running" check that doesn't need a session token. For anything beyond that (uptime, connectivity, traffic counters), authenticate and use `GET /api/status`.

3.5 POST /api/system/restart

Auth required: any authenticated session.

Reboots the device. Use this after applying configuration changes that require a restart, or as part of a remote recovery routine.

```
POST /api/system/restart
Authorization: Bearer <token>
```

Success (200):

```
{"status": "ok", "message": "System restarting..."}
```

The response is sent *before* the device actually restarts (there's a fixed 500 ms delay to guarantee delivery), so a 200 here is a promise that the restart is imminent, not a confirmation it already happened. Expect the device to be briefly unreachable afterward – give it several seconds before polling `GET /api/status` again.

3.6 POST /api/system/firmware_update_allowed

Auth required: any authenticated session.

Turns automatic OTA updates via hawkBit on or off for this device. Defaults to `true` on every device. When set to `false`, the device keeps polling hawkBit and reporting its status normally, but will not download or apply any deployment assigned to it until the flag is turned back on.

```
POST /api/system/firmware_update_allowed
Authorization: Bearer <token>
Content-Type: application/json

{"allowed": false}
```

Success (200): `{"status": "ok", "firmware_update_allowed": false}`. **Errors:** 400 (missing `allowed`), 500 (`"Failed to update setting"`).

The current value is also readable via the `firmware_update_allowed` field on `GET /api/status` and the `GET /api/config` degraded fallback response (Section 3 above), and reported to hawkBit itself as a `firmwareUpdateAllowed` target attribute – so a device with updates blocked stays visible with a pending-but-unapplied deployment in hawkBit/the Cloud portal rather than dropping off.

This is an all-or-nothing switch – turning it off blocks every future deployment indiscriminately, including a security-critical one. There is currently no way to allow only high-severity updates through.

Chapter 4

Mappings

A *mapping* is a single register-translation rule: read a value from a source device/register, optionally scale/offset it, and write it to a target device/register. This chapter covers cyclic and on-demand/on-event mappings (`/api/mappings`) – for the separate 1:1 passthrough system, see Chapter 5.

These endpoints always respond with HTTP 200, even on failure. Unlike the authentication endpoints in Chapter 2, a failed create/update/delete here does *not* produce a 4xx status – you must check the `status` field in the body on every call. A failure looks like `{"status": "error", "code": -1074}` (an ESP-IDF `esp_err_t` integer, not a human-readable message).

4.1 GET /api/mappings

Auth required: any authenticated session.

```
GET /api/mappings
Authorization: Bearer <token>
```

With no query parameters, returns a plain JSON array of every configured mapping. Each entry looks like:

```
{
  "source": {
    "slave_id": 1, "register": 100, "function": 3,
    "data_type": "u16", "byte_order": 0, "port": 0, "protocol": "rtu",
    "rtu": {"uart_num": 1, "baudrate": 9600, "parity": 0, "stop_bits":
       1}
  },
  "target": {
    "slave_id": 2, "register": 200, "function": 6,
    "data_type": "u16", "byte_order": 0, "port": 1, "protocol": "rtu",
    "rtu": {"uart_num": 2, "baudrate": 19200, "parity": 0, "stop_bits":
       1}
  },
  "source_config": {"rtu": {"uart_num": 1, "baudrate": 9600, "parity":
     0, "stop_bits": 1}},
  "target_config": {"rtu": {"uart_num": 2, "baudrate": 19200, "parity":
     0, "stop_bits": 1}},
  "_index": 0,
  "scale": 1.0,
  "offset": 0.0,
  "enabled": true,
  "mode": "cyclic",
  "source_slave_id": 1, "source_register": 100, "source_function": 3,
```

```
  "source_byte_order": 0, "source_data_type": 0, "source_port": 0, "
     source_protocol": 0,
  "target_slave_id": 2, "target_register": 200, "target_function": 6,
  "target_byte_order": 0, "target_data_type": 0, "target_port": 1, "
     target_protocol": 0
}
```

Every mapping is returned in **three overlapping representations at once**: the nested `source`/`target` objects, a redundant `source_config`/`target_config` pair (duplicates the `rtu`/`tcp` block already inside `source`/`target`), and a flat set of `source_*`/`target_*` fields. `_index` is this mapping's position – pass it back as `index` to the PUT and delete endpoints in this chapter.

The nested `source.data_type`/`target.data_type` string is unreliable. It only correctly distinguishes two values – `"float"` (32-bit IEEE 754) and `"u32"` – and silently falls back to `"u16"` for every other data type (`i16`, `i32`, `u64`, `i64`, `float64`, `bcd`). **Always read the numeric `source_data_type`/`target_data_type` flat fields instead** and decode them yourself using the table in Appendix 8 – those integer fields are correct for all 9 data types.

4.1.1 Paging

For large mapping tables, pass any of `offset`, `limit`, or `compact` as query parameters and the response switches to a paged envelope:

```
GET /api/mappings?offset=0&limit=24&compact=1
```

```
{
  "items": [ ... mapping objects, as above ... ],
  "total": 40,
  "offset": 0,
  "limit": 24,
  "count": 24
}
```

`limit` is clamped to the range 1-64 (defaults to 24 if paging is triggered but `limit` isn't given). `compact=1` omits the nested `source`/`target`/`source_config`/`target_config` objects from each item, returning only the flat fields – useful for large tables where you only need the numerically-correct flat representation anyway (see the warning above).

4.2 POST /api/mappings – create a mapping

Auth required: any authenticated session.

```
POST /api/mappings
Authorization: Bearer <token>
Content-Type: application/json

{
  "source": {
    "slave_id": 1, "register": 100, "function": 3,
    "data_type": "float", "byte_order": "be_swap",
    "port": 0, "protocol": "rtu",
    "rtu": {"uart_num": 1, "baudrate": 9600, "parity": 0, "stop_bits":
       1}
  },
  "target": {
```

```
    "slave_id": 2, "register": 200, "function": 6,
    "data_type": "float", "byte_order": "be_swap",
    "port": 1, "protocol": "rtu",
    "rtu": {"uart_num": 2, "baudrate": 19200, "parity": 0, "stop_bits":
        1}
  },
  "scale": 1.0,
  "offset": 0.0
}
```

Success (200): `{"status": "ok"}`. **Failure (200, with an error body):** `{"status": "error", "code": <esp_err_t>}` – typically `ESP_ERR_NO_MEM` (mapping table full, max `MAX_MAPPINGS` = 512) or `ESP_ERR_INVALID_ARG` (the new mapping failed internal RTU-consistency validation – e.g. two mappings on the same RTU port with conflicting serial settings).

Sending nested `source`/`target` objects (as above) is the **preferred and fully-supported** format. Both `data_type` and `byte_order` accept *either* their string form (shown above) *or* the raw integer enum value from Appendix 8 – use whichever is more convenient for your integration.

Accepted `data_type` strings: `u16`, `u32`, `u64`, `i16`, `i32`, `i64`, `float` or `float32`, `float64`, `bcd`.

Accepted `byte_order` strings: `be`, `le`, `be_swap`, `le_swap`. Note these are *not* the same spelling as the short labels shown in the web UI (e.g. the UI's "BE-S" is `be_swap` here).

Legacy flat-field input is only partially supported. If you omit *either* `source` or `target` as a nested object, the device falls back to reading flat top-level fields (`source_slave_id`, `source_register`, `source_function`, `source_data_type`, `source_port`, and the `target_*` equivalents) – but this legacy path **cannot set `byte_order`, `protocol`, or any RTU/TCP port configuration** for either side (protocol is merely guessed from the port: Ethernet implies TCP, anything else implies RTU with 9600/8/N/1 defaults). Use the nested format for any mapping that needs a byte order other than the default, or non-default serial settings.

The register count used for a value is not something you can set, and creating a mapping with anything other than `u16` silently underestimates it. Internally, every mapping created through this endpoint is stored with a register count of exactly **1**, regardless of the `data_type` you send – even for 32-bit and 64-bit types that actually need 2 or 4 registers. There is no field to override this via `POST /api/mappings`. In practice this means multi-register data types (`u32`, `i32`, `u64`, `i64`, `float`, `float64`) created this way may not read/write all their underlying registers correctly. If you need a mapping for one of these wider types, build and verify it once via the Dashboard's mapping editor (which is not affected by this limitation) and treat the API primarily for `u16`-based mappings until this is addressed in firmware.

A newly-created mapping is always enabled (`enabled: true`) – this endpoint has no way to create a mapping in a disabled state. To disable one afterward, use `PUT /api/mappings` (below).

A per-mapping `mode` field exists, but only does something for one of its three values. You can send `"mode": "cyclic"`, `"on_demand"`, or `"on_event"` (or the equivalent integer, 0-2) alongside `source`/`target` in the body above; it's stored and returned by

`GET /api/mappings` like any other field. In practice, though: **Cyclic and On-Demand mappings are driven entirely by the global mode** (`POST /api/translator/mode`, Chapter 7) – the firmware's cyclic-polling and on-demand-read code paths only check the device-wide setting, never this per-mapping field, so setting `"mode": "on_demand"` on an individual mapping while the global mode is `Cyclic` has *no effect* on that mapping. **On-Event is the one value that does work per-mapping**: a mapping with `"mode": "on_event"` reacts to matching source data arriving via any other traffic on the bus (another mapping's read, transparent-mode passthrough, etc.) and pushes to its target immediately, *regardless of what the global mode is currently set to* – it's effectively an always-on side channel you opt individual mappings into, not a mode you switch the whole device into. If omitted, a mapping defaults to `cyclic`. Note also that On-Event mappings only support a **local** target (`target_slave_id: 0`) – writing to a remote slave from an On-Event mapping is accepted but silently does nothing.

4.3 PUT /api/mappings – edit a mapping

Auth required: any authenticated session.

```
PUT /api/mappings
Authorization: Bearer <token>
Content-Type: application/json

{
  "index": 0,
  "source": {"slave_id": 1, "register": 100, "data_type": 0, "
     byte_order": 0,
             "function": 3, "port": 0, "protocol": "rtu",
             "rtu": {"uart_num": 1, "baudrate": 9600, "parity": 0, "
                stop_bits": 1}},
  "target": {"slave_id": 2, "register": 200, "data_type": 0, "
     byte_order": 0,
             "function": 6, "port": 1, "protocol": "rtu",
             "rtu": {"uart_num": 2, "baudrate": 19200, "parity": 0, "
                stop_bits": 1}},
  "scale": 1.0,
  "offset": 0.0,
  "enabled": true
}
```

`index` (the `_index` value from `GET /api/mappings`) is required and identifies which mapping to replace.

Success (200): `{"status": "ok"}`. **Failure (200, with an error body):** `{"status": "error", "code": <esp_err_t>}` – `ESP_ERR_INVALID_ARG` (bad index, or the edit fails RTU-consistency validation) or `ESP_ERR_TIMEOUT` (couldn't acquire the internal config lock in time – retry).

Unlike `POST`, this endpoint's nested `source`/`target` objects only accept `data_type` and `byte_order` as *integers* – the convenience string forms (`"u16"`, `"be_swap"`, etc.) are **not** recognized here and are silently ignored, leaving the field at its zero value (`u16`/`big-endian`). Always send the numeric enum value (Appendix 8) when using `PUT`.

Unlike `POST`, this endpoint does read `enabled` from the request body, so it's the correct way to disable/re-enable an existing mapping without changing anything else – just resend its current values with `"enabled": false`.

The same legacy flat-field fallback described under `POST` above applies here too (triggered the same way, with the same limitations) if you omit `source`/`target` as nested objects.

4.4 Deleting mappings

There is no `DELETE /api/mappings` endpoint. Despite the name, deletion is a `POST` to a dedicated sub-path.

4.4.1 POST /api/mappings/delete

```
POST /api/mappings/delete
Authorization: Bearer <token>
Content-Type: application/json

{"index": 0}
```

Deletes the mapping at `index` and shifts every subsequent mapping's index down by one. **Success (200):** `{"status": "ok"}`. **Failure:** `{"status": "error", "code": <esp_err_t>}` (`ESP_ERR_INVALID_ARG` for an out-of-range index).

Because deleting shifts all later indices down, don't cache a list of `_index` values and delete them one by one in the order you fetched them – re-fetch `GET /api/mappings` between deletes, or delete from the highest index down to index 0 in a single pass.

4.4.2 POST /api/mappings/clear

```
POST /api/mappings/clear
Authorization: Bearer <token>
```

Deletes **every** mapping in one call. **Success (200):** `{"status": "ok", "message": "All mappings cleared"}`. There is no confirmation step – this is immediate and irreversible via the API (short of recreating the mappings or restoring from a saved `GET /api/config` snapshot).

Chapter 5

Transparent Mappings

Transparent mode is a separate, simpler passthrough system from the mappings in Chapter 4: each transparent mapping forwards an **entire Modbus request 1:1** between a source slave/port and a target slave/port, across the whole register range (0-65535), with **no scale, no offset, and no data-type interpretation**. Use it when you need to bridge a whole downstream device (e.g. expose an RTU slave over TCP, or vice versa) rather than translate specific registers.

> The device-wide `transparent_mode_enabled` switch (visible in `GET /api/config`/`GET /api/status`) and `translator_enabled` are mutually exclusive at the system level – enabling transparent mode disables the regular mapping translator. The endpoints in this chapter manage the list of transparent mappings themselves, not that top-level switch.

> Unlike Chapter 4, these endpoints use **flat fields only** – there is no nested `source`/`target` object format here, and no `data_type`/`byte_order` fields at all (transparent mode doesn't interpret the data it forwards).

5.1 GET /api/transparent_mappings

Auth required: any authenticated session.

```
GET /api/transparent_mappings
Authorization: Bearer <token>
```

```
[
  {
    "enabled": true,
    "source_slave_id": 1,
    "source_function": 3,
    "source_port": 0,
    "source_protocol": "rtu",
    "source_uart_num": 1,
    "source_baudrate": 9600,
    "source_parity": 0,
    "source_stop_bits": 1,
    "target_slave_id": 1,
    "target_function": 3,
    "target_port": 2,
    "target_protocol": "tcp",
    "target_ip_addr": "192.168.1.50",
    "target_tcp_port": 502
  }
]
```

The `source_*`/`target_*` RTU fields (`*_uart_num`, `*_baudrate`, `*_parity`, `*_stop_bits`) are only present when that side's protocol is `"rtu"`; the TCP fields (`*_ip_addr`, `*_tcp_port`) are only present when it's `"tcp"`. There is no `_index` field in each item – an item's **position in the array** (0-based) is its index, used by the PUT/DELETE endpoints below.

Supports the same `offset`/`limit` paging query parameters as `GET /api/mappings` (Section 4), but **not** `compact` (there's nothing to compact – there's only one representation here).

5.2 POST /api/transparent_mappings – create

Auth required: any authenticated session.

```
POST /api/transparent_mappings
Authorization: Bearer <token>
Content-Type: application/json

{
  "enabled": true,
  "source_slave_id": 1, "source_function": 3, "source_port": 0,
  "source_protocol": "rtu",
  "source_uart_num": 1, "source_baudrate": 9600, "source_parity": 0, "
     source_stop_bits": 1,
  "target_slave_id": 1, "target_function": 3, "target_port": 2,
  "target_protocol": "tcp",
  "target_ip_addr": "192.168.1.50", "target_tcp_port": 502
}
```

`enabled` defaults to `true` if omitted. `source_protocol`/`target_protocol` are strings (`"rtu"` or `"tcp"`) – unlike Chapter 4, there is no integer form accepted here. Supply the matching RTU or TCP fields for each side based on its protocol.

Success (200): `{"status": "ok", "message": "Transparent mapping added"}`.

Errors: `400 Bad Request` (`"Source and target ports must be different"` – `source_port` and `target_port` cannot be the same physical port), `500 Internal Server Error` (`"Failed to add transparent mapping"` – typically the transparent-mapping table is full, max `MAX_TRANSPARENT_MAPPINGS` = 256).

> Unlike the regular mappings endpoints in Chapter 4, this endpoint returns a genuine `4xx`/`5xx` HTTP status on failure rather than always answering 200 – checking the HTTP status code alone is sufficient here.

5.3 PUT /api/transparent_mappings – edit

Auth required: any authenticated session.

```
PUT /api/transparent_mappings
Authorization: Bearer <token>
Content-Type: application/json

{
  "index": 0,
  "enabled": true,
  "source_slave_id": 1, "source_function": 3, "source_port": 0,
  "source_protocol": "rtu",
  "source_uart_num": 1, "source_baudrate": 19200, "source_parity": 0, "
     source_stop_bits": 1,
  "target_slave_id": 1, "target_function": 3, "target_port": 2,
  "target_protocol": "tcp",
```

```
  "target_ip_addr": "192.168.1.50", "target_tcp_port": 502
}
```

`index` is the item's position in the `GET` array (required). This call replaces the entire entry – omitted fields fall back to their zero value, not their previous value, so send the complete object back (as returned by `GET`, with your changes applied), not just the fields you're changing.

Success (200): `{"status": "ok", "message": "Transparent mapping updated"}`. **Errors:** `400 Bad Request` (missing/invalid `index`, or `source_port` equals `target_port`), `500 Internal Server Error` (`"Failed to update transparent mapping"` – bad index, or the mapping table lock couldn't be acquired).

5.4 DELETE /api/transparent_mappings

Auth required: any authenticated session.

```
DELETE /api/transparent_mappings
Authorization: Bearer <token>
Content-Type: application/json

{"index": 0}
```

Success (200): `{"status": "ok"}`. **Failure:** `{"status": "error"}`, still with HTTP 200 – check the body's `status` field for this one call, since (unlike `POST`/`PUT` above) it doesn't distinguish failures with a 4xx/5xx status code.

This is the one transparent-mapping endpoint that genuinely uses the HTTP `DELETE` method (contrast with Chapter 4, where deleting a regular mapping is a `POST` to `/api/mappings/delete`). Deleting also shifts every later index down by one, same caveat as Section 4 applies: re-fetch between deletes, or delete highest-index-first.

Chapter 6

Manual Read

This endpoint performs a single, one-off Modbus read directly from the device out to a slave – useful for commissioning, wiring verification, or ad-hoc diagnostics, without needing a configured mapping.

Calling this endpoint changes the device's running configuration as a side effect. If either `translator_enabled` or `transparent_mode_enabled` is currently on, the very first call to this endpoint **turns them off** (so the manual read can use the master port exclusively) – these are the same flags visible in `GET /api/config`/`GET /api/status` (Chapter 3). They are *not* automatically turned back on afterward. If your integration relies on the translator or transparent mode running continuously, do not call this endpoint against a production device without re-enabling the appropriate mode afterward (via `POST /api/config`).

6.1 GET /api/manual_read

Auth required: any authenticated session.

```
GET /api/manual_read?port=0&slave_id=1&start_addr=0&count=4&function=3&
    data_type=2&byte_order=0
Authorization: Bearer <token>
```

Parameter	Default	Meaning
port	0	0 = RS485-1, 1 = RS485-2, 2 = Ethernet (TCP).
slave_id	1	Modbus slave/unit ID.
start_addr	0	First register address.
count	10	Number of *raw registers* to read (not values) – capped at 100.
function	3	Modbus function code: 1 (read coils), 2 (read discrete inputs), 3 (read holding registers), 4 (read input registers). Any other value silently falls back to 3.
data_type	0	Integer `data_type_t` value (Appendix 8) used only to group the raw registers into interpreted values (see below) – it does not affect what's actually read on the wire.
byte_order	0	Integer `byte_order_t` value (Appendix 8), applied to each group of registers before interpretation.
baud	9600	RTU only: baud rate for this one-off read.
parity	0	RTU only: 0 = none, 1 = even, 2 = odd.
stop_bits	1	RTU only.
ip	(none)	TCP only (`port=2`): target IP address – **required** when `port=2`.
eth_port	502	TCP only: target Modbus/TCP port.

Success (200):


```
{
  "quality_reason": "OK",
  "quality_detail": "ESP_OK",
  "registers": [
    {
      "address": 0,
      "raw_values": [17224, 16960],
      "transformed_values": [17224, 16960],
      "interpreted": "123.45",
      "quality": true
    },
    {
      "address": 2,
      "raw_values": [0, 0],
      "transformed_values": [0, 0],
      "interpreted": "N/A",
      "quality": false,
      "quality_reason": "TIMEOUT",
      "quality_detail": "ESP_ERR_TIMEOUT"
    }
  ]
}
```


Each entry in `registers` is one *interpreted value*, not one raw register – `count` raw registers are grouped into `count / regs_per_value` entries, where `regs_per_value` is derived from `data_type` (1 register for `u16`/`i16`, 2 for `u32`/`i32`/`float`, 4 for `u64`/`i64`/`float64`). `raw_values` are the untouched register words as read from the wire; `transformed_values` are the same words

after `byte_order` has been applied; `interpreted` is the human-readable decoded value as that `data_type` (or `"N/A"` if the read failed).

The top-level `quality_reason`/`quality_detail` describe the read as a whole; each entry additionally carries its own `quality` boolean (and, only when `false`, its own `quality_reason`/`quality_detail`) – in practice these always match the top-level ones today, since the entire read either succeeds or fails as one unit, but treat the per-entry fields as authoritative if that ever changes.

Failure (bad request, wrong parameters) is still HTTP 200, with `quality_reason: "ERROR"` (or `"TIMEOUT"`) and empty/zeroed values – this endpoint does not use HTTP error status codes for read failures, only for things like a malformed query string.

`GET /api/registers` is a legacy alias for this exact same endpoint – both paths are handled identically. Prefer `/api/manual_read` in new integrations.

If the configured RTU Slave (Modbus TCP/RTU slave-side server) happens to be running on the same UART you're targeting with `port=0`/`port=1`, the device **temporarily stops it** for the duration of this one read (and restarts it afterward, roughly 200 ms later) to avoid a bus conflict. Expect a brief gap in slave-side availability on that port when using manual read against a live system.

Chapter 7

Network & Translator Control

7.1 GET /api/network

Auth required: any authenticated session for the response in general; `wifi_password` specifically is only populated for the `superadmin` account.

```
GET /api/network
Authorization: Bearer <token>
```


```
{
  "wifi_mode": 2,
  "wifi_ssid": "Factory-Floor-2G",
  "wifi_password": "",
  "wifi_enabled": true,
  "wifi_dhcp_enabled": true,
  "wifi_static_ip": "",
  "wifi_netmask": "",
  "wifi_gateway": "",
  "wifi_dns1": "",
  "wifi_dns2": "",
  "eth_dhcp_enabled": false,
  "eth_static_ip": "192.168.10.111",
  "eth_netmask": "255.255.255.0",
  "eth_gateway": "192.168.10.1",
  "eth_dns1": "8.8.8.8",
  "eth_dns2": ""
}
```


`wifi_mode` is an integer: 0 = AP only, 1 = STA only, 2 = AP+STA. Ethernet has no on/off flag – it's always active; only its IP configuration is here.

> For customer accounts `wifi_password` is always empty (`""`) in this response: the configured Wi-Fi password is write-only through the API. To change it, send the new value with `POST /api/network`.

7.2 POST /api/network

Auth required: any authenticated session.

```
POST /api/network
Authorization: Bearer <token>
Content-Type: application/json

{
  "wifi_mode": 2,
```

```
  "wifi_enabled": true,
  "wifi_ssid": "Factory-Floor-2G",
  "wifi_password": "new-password",
  "wifi_dhcp_enabled": true,
  "eth_dhcp_enabled": false,
  "eth_static_ip": "192.168.10.111",
  "eth_netmask": "255.255.255.0",
  "eth_gateway": "192.168.10.1",
  "eth_dns1": "8.8.8.8"
}
```

Success (200): `{"status": "ok", "message": "Network config saved"}`. **Errors:** 500 (`"Failed to save network config"`).

This call **loads the existing network configuration first** and only overwrites the fields you include – so a partial body (e.g. just `{"eth_static_ip": "..."}`) is safe and won't blank out your Wi-Fi settings. The `saved_networks` list (Section 7.4) is always preserved regardless of what you send here. Two fields default to `true` whenever they're entirely absent from the body, even on an update: `wifi_dhcp_enabled` and `eth_dhcp_enabled`.

Changing network settings (especially `wifi_mode`, static IP settings, or disabling DHCP with an incorrect address) can make the device unreachable at its previous address. Confirm you have another way to reach the device (serial console, or the AP-mode fallback network) before applying a network change you're not certain about, particularly over a remote/tunneled connection.

7.3 GET /api/wifi/scan

Auth required: any authenticated session.

Triggers a live Wi-Fi scan (blocking – this call takes a few seconds) and returns nearby access points.

```
GET /api/wifi/scan
Authorization: Bearer <token>
```

```
[
  {"ssid": "Factory-Floor-2G", "rssi": -52, "auth": 3, "channel": 6},
  {"ssid": "Guest-WiFi", "rssi": -71, "auth": 0, "channel": 11}
]
```

`rssi` is signal strength in dBm (higher/less negative is stronger). `auth` is the raw ESP-IDF `wifi_auth_mode_t` value (0 = open, 3 = WPA2-PSK, etc.) – this reference doesn't reproduce the full ESP-IDF enum; treat `0` as open and any non-zero value as secured if you don't need the exact scheme. Up to 20 access points are returned.

7.4 Saved Wi-Fi Networks

The device can remember up to **5 Wi-Fi networks** with a priority order, independent of the single active `wifi_ssid`/`wifi_password` pair in `GET /api/network`.

7.4.1 GET /api/wifi/networks

```
GET /api/wifi/networks
Authorization: Bearer <token>
```

```
[
  {"ssid": "Factory-Floor-2G", "priority": 0},
  {"ssid": "Backup-Router", "priority": 1}
]
```

Unlike `GET /api/network`, saved networks' passwords are **not** included in this response – only `ssid` and `priority` (0 = highest priority). An item's position in the array is its index, used by the delete call below.

7.4.2 POST /api/wifi/networks – add

```
POST /api/wifi/networks
Authorization: Bearer <token>
Content-Type: application/json

{"ssid": "Backup-Router", "password": "backup-password", "priority": 1}
```

`password` defaults to an empty string (open network) if omitted; `priority` defaults to `0`.

Success (200): `{"status": "ok"}`. **Errors:** 400 (missing `ssid`), 500 with `{"status": "error", "error": "Maximum networks reached"}` once you already have 5 saved, or `{"status": "error", "error": "Failed to add network"}` on any other failure.

Note the error key here is `error`, not `message` – inconsistent with most other endpoints in this document (see Chapter 1's general envelope note). Check for either key defensively if you centralize error handling across endpoints.

7.4.3 DELETE /api/wifi/networks/{index}

```
DELETE /api/wifi/networks/1
Authorization: Bearer <token>
```

The index is part of the URL path, not the body. **Success (200):** `{"status": "ok"}`. **Errors:** 500 (`"Failed to remove network"` – out-of-range index).

7.5 Translator Controls

These four endpoints operate on the global mapping translator as a whole (see `translator_enabled`/`global_mapping_mode` in Chapter 3), rather than on individual mappings.

7.5.1 POST /api/translator/enable

```
POST /api/translator/enable
Authorization: Bearer <token>
Content-Type: application/json

{"enabled": true}
```

Turns the whole mapping translator on or off. **Success (200):** `{"status": "ok", "translator_enabled": true}`. **Errors:** 400 (missing `enabled`), 500 (`"Failed to update translator state"`).

7.5.2 POST /api/translator/reload

```
POST /api/translator/reload
Authorization: Bearer <token>
```

Reloads the translator's in-memory mapping table from the saved configuration without a full device restart – use this after making mapping changes via a route that doesn't already trigger a reload on its own, or if mappings seem stale. **Success (200):** `{"status": "ok", "message": "Configuration reloaded"}`.

7.5.3 POST /api/translator/mode

```
POST /api/translator/mode
Authorization: Bearer <token>
Content-Type: application/json

{"mode": 0}
```

Sets the **global** mapping mode, which is what actually governs Cyclic and On-Demand behavior for every mapping at once – see Section 4.2 for the one exception (per-mapping `on_event`). `mode` is an integer: 0 = Cyclic, 1 = On-Demand, 2 = On-Event (see Appendix 8). **Success (200):** `{"status": "ok", "global_mapping_mode": 0}`. **Errors:** 400 (missing `mode`, or a value outside 0-2).

7.5.4 GET /api/translator/stats

```
GET /api/translator/stats
Authorization: Bearer <token>
```

Per-mapping live counters, indexed to match `GET /api/mappings`' `_index`:

```
{
  "translator_enabled": true,
  "stats": [
    {
      "index": 0,
      "source_value": "123.45",
      "target_value": "123.45",
      "source_raw": "0x4123",
      "target_raw": "0x4123",
      "success_count": 4821,
      "error_count": 3,
      "last_update_ms": 8453210
    }
  ]
}
```

If the translator is currently disabled, `stats` is an empty array. This response is internally cached for **1 second** – polling faster than that returns the same cached snapshot rather than fresher data.

Chapter 8

Enum Reference

This appendix collects every enumerated field used across this document, with its integer value, and (where the API accepts one) its string form. When a field is documented elsewhere as accepting only an integer, use the integer column; where a string form is listed, both are interchangeable unless the relevant chapter says otherwise.

8.1 data_type

Int	String	Meaning
0	`u16`	16-bit unsigned integer (1 register).
1	`u32`	32-bit unsigned integer (2 registers).
2	`float` / `float32`	32-bit IEEE 754 float (2 registers). Also accepted as legacy alias `DATA_TYPE_FLOAT`.
3	`bcd`	Binary Coded Decimal (2 registers).
4	`i16`	16-bit signed integer (1 register).
5	`i32`	32-bit signed integer (2 registers).
6	`u64`	64-bit unsigned integer (4 registers).
7	`i64`	64-bit signed integer (4 registers).
8	`float64`	64-bit IEEE 754 double (4 registers). Also accepted as legacy alias `DATA_TYPE_DOUBLE`.

Recall Section 4.1: the nested `source.data_type`/`target.data_type` *string* field returned by `GET /api/mappings` can only render `"float"` or `"u32"` correctly – everything else in this table comes back mislabeled as `"u16"` in that specific nested field. Always decode the numeric `source_data_type`/`target_data_type` flat fields using this table instead.

8.2 byte_order

Int	String	Meaning
0	`be`	Big-endian (most significant register first, natural word order).
1	`le`	Little-endian (least significant register first).
2	`be_swap`	Big-endian with byte-swap within each register. Shown as "BE-S" in the web UI.
3	`le_swap`	Little-endian with byte-swap within each register. Shown as "LE-S" in the web UI.

8.3 modbus_function

Int	Meaning
1	Read Coils.
2	Read Discrete Inputs.
3	Read Holding Registers (the default used when an invalid/omitted function code is supplied).
4	Read Input Registers.
5	Write Single Coil.
6	Write Single Register.
16	Write Multiple Registers.

There is no string form for this field anywhere in the API – always send the integer function code.

8.4 port (port_type_t)

Int	Meaning
0	RS485-1 (first serial port).
1	RS485-2 (second serial port).
2	Ethernet (TCP).

8.5 protocol (modbus_protocol_t)

Int	String	Meaning
0	rtu	Modbus RTU (serial).
1	tcp	Modbus TCP.

Only POST/PUT /api/mappings's nested source/target objects and the transparent-mapping endpoints (Chapter 5) accept the string form; anywhere else in the API that carries a protocol value, send the integer.

8.6 mapping_mode (mapping_mode_t)

Int	String	Meaning
0	cyclic	Poll continuously at the configured interval (the default).
1	on_demand	Trigger a read and forward it immediately on request, rather than on a timer.
2	on_event	Forward as soon as source data is received, rather than polling.

This is a **global** setting (POST /api/translator/mode, Chapter 7) applied to the whole translator – there is no per-mapping mode field exposed through POST/PUT /api/mappings, despite mapping_mode_t being defined per-mapping internally in firmware. The string form ("on_demand"/"on_event") is only accepted by the full-configuration import path (POST /api/config, Chapter 3); POST /api/translator/mode itself only accepts the integer.

8.7 wifi_mode

Int	Meaning
0	AP only (device hosts its own Wi-Fi network, no client connection).
1	STA only (device joins an existing Wi-Fi network as a client).
2	AP+STA (both simultaneously – the default).

8.8 HTTP status vs. body status – quick reference

Because response conventions are not fully uniform across this API (see the individual chapters for each endpoint's specifics), here's a quick summary of which pattern each family of endpoints follows:

Endpoint family	Failure convention
Authentication (Ch. 2)	Real 4xx/5xx status codes.
Configuration & Status (Ch. 3)	Real 4xx/5xx status codes.
Mappings (Ch. 4)	**Always HTTP 200** – check `status` in the body.
Transparent Mappings – POST/PUT (Ch. 5)	Real 4xx/5xx status codes.
Transparent Mappings – DELETE (Ch. 5)	**Always HTTP 200** – check `status` in the body.
Manual Read – read failures (Ch. 6)	**Always HTTP 200** – check `quality_reason` in the body.
Network & Translator Control (Ch. 7)	Real 4xx/5xx status codes.