# DODO Sensor Logger — Custom HTTPS Endpoint Guide

# DODO Sensor Logger — Custom HTTPS Endpoint Guide

**Firmware version:** 2.3.0 or later **Audience:** Customers who want to send sensor telemetry to their own server instead of the DODOTiK platform.

---

## Table of contents

1. What is Custom HTTPS mode?
2. Before you start
3. Step-by-step setup on the device
4. The Test Send button — verify before saving
5. Data your server will receive
6. HTTP request reference
7. Authentication
8. Server-side recipes
9. Troubleshooting
10. Operational notes
11. FAQ

---

## 1. What is Custom HTTPS mode?

By default, every DODO Sensor Logger publishes its readings to the DODOTiK platform over MQTT-TLS. **Custom HTTPS mode** is an alternative built into firmware v2.3.0+ that lets your device send the same readings to **your own HTTPS endpoint** instead — as a JSON POST.

You'd use this if:

- You already have a server, dashboard, or pipeline you want the device to feed (Node-RED, Home Assistant, AWS, Azure, GCP, your own SaaS, an SQL ingest API, etc.).
- You don't want to depend on the DODOTiK cloud for telemetry routing.
- You're integrating the device into a larger system that speaks REST.

**What stays the same when you switch to Custom HTTPS:**

- The device's local web interface (Cloud Settings, Sensor configuration, SD card download, OTA upload, RTC time).
- All Modbus and analog (0-10 V / 4-20 mA) sensor reading logic.
- The SD card CSV logging — every reading is written to `/sensors-YYYYMMDD.csv` regardless of telemetry mode.
- **Over-the-air firmware updates** still come from `platform.dodotik.com`. Only telemetry routing changes.

**What changes:**

- The device no longer connects to the DODOTiK MQTT broker (`mqtt.platform.dodotik.com:8883`).
- Every send interval (default 10 minutes), the device POSTs a single JSON document to your URL over HTTPS.
- You're responsible for receiving, validating, and storing that JSON on your side.

---

## 2. Before you start

You need:

| Requirement | Notes |
|---|---|
| **DODO Sensor Logger v2.3.0 or later** | Check the boot serial output or the firmware version shown in the device web UI footer. Older firmware doesn't have the Custom HTTPS option. Upload v2.3.0+ via the OTA page if needed. |
| **A reachable HTTPS endpoint** | Must start with `https://`. The device **refuses plain `http://` URLs** for security. The endpoint must have a valid public TLS certificate (Let's Encrypt is fine). Self-signed certificates work because the device uses `setInsecure()` for compatibility, but the URL must still be `https://`. |
| **Server-side code that responds with HTTP 2xx** | Any response in the 200-299 range is treated as success. Anything else counts as a failure (see Troubleshooting). |
| **WiFi credentials for the device's network** | The same network used for the existing DODOTiK setup is fine. |
| **(Optional) A Bearer token** | A string the device will send in the `Authorization` header on every POST. If you don't set one, no Authorization header is sent. |

---

## 3. Step-by-step setup on the device

### 3.1 Open the device's web interface

After the device is on your WiFi, browse to its IP (e.g. `http://192.168.1.45/`) or its mDNS name (`http://dodo-logger-XXXX.local/` where `XXXX` is the last 4 hex characters of the device MAC, shown on the boot serial line `[Boot] AP SSID: DODO-Logger-XXXX`).

## 3.2 Navigate to Cloud Settings

From the main panel tap **Cloud** (or **Cloud Settings** depending on the page).

## 3.3 Pick "HTTPS — Custom Endpoint"

The Protocol dropdown has three options:

| Option | Meaning |
|---|---|
| **MQTT — DODOTiK Platform (Recommended)** | The default. MQTT-TLS to the DODOTiK broker. Use this if you also want the DODOTiK dashboard. |
| **HTTPS — DODOTiK Platform (not available)** | Greyed out / disabled in v2.3.2+. Reserved for future use. |
| **HTTPS — Custom Endpoint (Your Server)** | **The option you want.** Selecting it hides the Device ID / Device Key fields and reveals three new fields. |

## 3.4 Fill in the three Custom HTTPS fields

When you select "Custom Endpoint", the form changes to show:

| Field | What to enter | Example |
|---|---|---|
| **Device Name** (sent as `deviceId` in JSON) | A name your server will use to identify this physical device. Pre-filled with the device's MAC address by default (e.g. `909BA2080814`). You can rename to anything 1-32 characters long. | `Greenhouse-Tomatoes-01` |
| **Custom URL** | The full `https://` URL where the device should POST. Must start with `https://` — the device refuses plain `http://`. | `https://api.your-company.com/v1/sensor/ingest` |
| **Bearer Token (optional)** | If set, the device adds `Authorization: Bearer <your-token>` to every POST. Leave empty if your URL is itself a secret (signed query string, hard-to-guess path, etc.). | `9b1f8a2e4c6d7e9f1a3b5c7d` |

The other fields on the panel work like always:

- **Send Interval (minutes)** — How often the device POSTs real telemetry. Default 10. Min 1. Max 1440 (24 h). For testing, set to 1 so you see the next POST quickly; reset to 10+ for production to be kind to your server and the device's battery/data plan.

## 3.5 Verify with Test Send

Before saving, click the **Test Send** button (only visible in Custom HTTPS mode). The device immediately POSTs a small **fixed sample payload** to the URL you typed, using the bearer token (if any), and shows you the result inline:

- ✅ **OK (HTTP 200)** — Green check. Your endpoint accepted the test. Note: it'll show a body preview if your server returned anything in the response body.
- ❌ **FAIL: URL is required / URL must start with** `https://` — Fix the URL field.
- ❌ **FAIL: network error -1** — DNS lookup or TCP connection failed. Check the URL is reachable from the device's network.
- ❌ **FAIL: HTTP 401** — Your server rejected the bearer token. Update it.
- ❌ **FAIL: HTTP 403/404/etc** — Your server returned a non-2xx status. Check the server logs.

**Important:** Test Send uses the values **currently in the form**, not the saved values. So you can iterate — change URL → Test Send → change again → Test Send — without re-saving each time.

The test payload includes `"test": true` so you can filter it on your server (see §5).

### 3.6 Save

Once Test Send is green, click **Save All & Reboot** at the bottom of the panel. The device:

1. Writes the new settings to flash storage (NVS).
2. Reboots.
3. After reboot, starts POSTing real telemetry to your URL at the configured interval.

You should see the first real POST land within `Send Interval` minutes of the reboot.

---

## 4. The Test Send button — verify before saving

Test Send is the single most useful tool when you're integrating. It's a dry-run that:

- Reads the URL + bearer from the **form**, not NVS.
- Refuses anything that doesn't start with `https://`.
- Refuses if the device is currently in the middle of an OTA check or other TLS session (returns HTTP 409, retry in a few seconds).
- POSTs a **fixed sample payload** with `"test": true` so it works even on a brand-new device with no slots configured.
- Returns the customer's HTTP response code and first 200 bytes of response body inline.

Use it for:

- First-time URL configuration.
- Verifying a new bearer token.
- Confirming the device can reach your server after a firewall/DNS change.
- Smoke-testing your server-side handler before you point real production devices at it.

It does NOT:

- Write anything to NVS.
- Affect the running telemetry schedule.
- Use real sensor readings.

## 5. Data your server will receive

Every send interval, the device POSTs **one JSON object** to your URL containing all currently enabled slots and their latest readings.

### 5.1 Real-telemetry example

A device named `Greenhouse-Tomatoes-01` with three configured slots — one Modbus 7-in-1 soil sensor, one analog LM35 temperature, one analog water-level sensor — would POST something like this:

```
{
  "deviceId": "Greenhouse-Tomatoes-01",
  "firmware": "2.3.3",
  "ts": 1779023398,
  "ts_iso": "2026-05-17T13:09:58Z",
  "slots": [
    {
      "index": 1,
      "name": "RT-3000-WS-N01-PE",
      "fields": [
        {
          "key": "humidity",
          "label": "Humidity",
          "unit": "%RH",
          "value": 49.2
        },
        {
          "key": "temperature_c",
          "label": "Temperature",
          "unit": "°C",
          "value": 23.6
        }
      ]
    },
    {
      "index": 2,
      "name": "Temperature LM35",
      "fields": [
        {
          "key": "temperature",
          "label": "Temperature LM35",
          "unit": "°C",
          "value": 22.79
        }
      ]
    },
    {
      "index": 3,
      "name": "Water Level",
      "fields": [
        {
          "key": "water_level",
          "label": "Water Level",
          "unit": "cm",
          "value": 22.25
        }
      ]
    }
  ]
}
```


## 5.2 Test Send example

When you click Test Send, the device POSTs this fixed sample (independent of real sensor configuration):

```
{
  "deviceId": "Greenhouse-Tomatoes-01",
  "firmware": "2.3.3",
  "ts": 1779023426,
  "ts_iso": "2026-05-17T13:10:26Z",
  "test": true,
  "slots": [
    {
      "index": 1,
      "name": "Test Slot",
      "fields": [
        {
          "key": "value",
          "label": "Test",
          "unit": "-",
          "value": 42.0
        }
      ]
    }
  ]
}
```


**The `"test": true` field is the easiest way to distinguish probes from real data on your server.** Add a one-line check:

```
if (body.test === true) {
  // It's a Test Send probe. Acknowledge but don't store.
  return res.status(200).json({ ok: true, message: "test acknowledged" });
}
```

## 5.3 Field reference

| Field | Type | Meaning |
|---|---|---|
| `deviceId` | string | The Device Name you typed in the web UI. Use it to look up which physical device sent this payload. |
| `firmware` | string | Firmware version on the device (e.g. `"2.3.3"`). Useful for detecting which devices need OTA updates. |
| `ts` | integer | UTC Unix epoch seconds at the moment of reading. |
| `ts_iso` | string | The same timestamp in ISO 8601 format, UTC (e.g. `"2026-05-17T13:09:58Z"`). Use whichever is easier on your side. |
| `test` | boolean | **Only present (and `true`) on Test Send probes.** Absent on real telemetry. |
| `slots` | array | One element per enabled slot, in slot-index order. Empty array if no slots are configured. |
| `slots[].index` | integer | Slot number on the device, 1-based (1, 2, 3, or 4). Stable for the life of the slot configuration. |
| `slots[].name` | string | The friendly name the operator typed for this slot in the device's Sensor Configuration page (e.g. `"Greenhouse Soil"`, `"Pump Pressure"`). |
| `slots[].fields` | array | One element per reading the slot produces. Modbus slots can have up to 8 fields per reading; analog slots always have exactly 1. |
| `slots[].fields[].key` | string | Machine-friendly identifier for the field (e.g. `"humidity"`, `"temperature_c"`). Stable across firmware updates. Use this as a database column name. |
| `slots[].fields[].label` | string | Human-friendly label for dashboards (e.g. `"Humidity"`, `"Temperature"`). May change if the operator edits the sensor profile. |
| `slots[].fields[].unit` | string | Engineering unit (e.g. `"%RH"`, `"°C"`, `"cm"`, `"%"`, `"mA"`, `"V"`, `"-"` for unitless). |
| `slots[].fields[].value` | number or `null` | The reading. `null` means the sensor read failed on this cycle — handle as missing data, don't store as zero. |

# 6. HTTP request reference

## 6.1 Method, URL, body

```
POST <your-custom-url>
Content-Type: application/json
X-Device-ID: <your-device-name>
X-Firmware-Version: 2.3.3
Authorization: Bearer <your-token>   (only if you set one)

<JSON body from §5>
```

## 6.2 Headers

| Header | Always present? | Value |
|---|---|---|
| `Content-Type` | Yes | `application/json` |
| `X-Device-ID` | Yes | Same as `deviceId` in the JSON body. Convenience for routing/sharding before parsing the body. |
| `X-Firmware-Version` | Yes | Same as `firmware` in the JSON body. Useful for fleet-wide upgrade tracking. |
| `Authorization` | Only if Bearer Token is non-empty | `Bearer <your-token>` |
| `User-Agent` | Yes | Set by the ESP32 HTTP library (typically `ESP32HTTPClient` or similar — don't rely on the exact string). |
| `Content-Length` | Yes | Set automatically. |

## 6.3 Timing

| Setting | Value |
|---|---|
| Connect timeout | 5 seconds |
| Read timeout | 15 seconds |
| Body size (typical) | 400-700 bytes for 2-4 active slots |
| Body size (maximum) | ~2 KB for 4 slots × 8 fields each, all populated |
| Concurrency | At most 1 in-flight request per device |

## 6.4 What constitutes success

The device treats **HTTP status codes 200-299** as success. Body content is ignored — you don't need to return any specific JSON shape, just status code 200 with an empty body is fine.

**Anything else** (3xx redirects, 4xx client errors, 5xx server errors, network errors, TLS handshake failures) counts as a failure. The device logs the response code to its serial output but does **not** retry the same payload — the next scheduled cycle sends fresh data. After 5 consecutive failures, the device disconnects + reconnects WiFi and resets the failure counter.

---

## 7. Authentication

You have three ways to keep your endpoint from being abused, in order of strength:

### 7.1 Bearer token (recommended)

Set a long random string in the Bearer Token field. The device sends `Authorization: Bearer <that-string>` on every POST. Your server checks the header.

**Generate a token** (32 random bytes hex-encoded):

```
# Linux/Mac
openssl rand -hex 32

# Windows PowerShell
[Convert]::ToHexString((1..32 | %{ Get-Random -Max 256 }))
```

Pick something unguessable. Don't reuse the same token across many customers — give each customer or device fleet its own.

### 7.2 Hard-to-guess URL path

Generate a random path like `/api/v1/ingest/8f9a3c2d1b4e6f7a9c0e1d2f3a4b5c6d` and treat the URL itself as a secret. Cheap and simple but the URL is visible in your TLS-terminating proxy's access logs.

### 7.3 Both

Combine 7.1 and 7.2 for defense in depth. This is what we recommend for production.

### 7.4 What the firmware does NOT do

- The firmware does **not** generate or verify HMAC signatures on the payload.
- The firmware does **not** support mutual TLS (mTLS) — only standard TLS where you present a server certificate.
- The firmware does **not** rotate tokens. If you want to rotate, push a new token via the device's web UI (or update via OTA-friendly configuration storage in a future firmware version).

### 7.5 TLS posture

The device uses ESP32's `WiFiClientSecure` in `setInsecure()` **mode**, meaning it negotiates TLS encryption but does not validate the server's certificate chain. This is the same posture used for the device-to-DODOTiK MQTT connection. It's a deliberate trade-off:

- ✅ Works with self-signed certs, Let's Encrypt, expired-but-still-trusted certs.
- ✅ No need to bundle CA roots in firmware (saves ~50 KB flash).

- ❌ Does not protect against attackers who can perform on-path TLS interception with their own cert.

For most field-deployed IoT use cases — devices on customer LANs talking to known endpoints over WAN — this is appropriate. Pair with a Bearer token so spoofing the endpoint alone isn't enough to send fake data.

---

## 8. Server-side recipes

Pick whichever stack matches your existing infrastructure. All examples assume:

- Your endpoint is `https://api.example.com/v1/sensor`
- You configured the bearer token as `YOUR_SECRET`
- You want to ignore Test Send probes (filter on `body.test === true`)

### 8.1 Quick mock — listen with `nc` + a TLS-terminating proxy

Easiest way to see real device traffic in a few seconds. Use a free webhook receiver like https://webhook.site:

1. Open https://webhook.site in your browser.
2. Copy the unique URL shown at the top (`https://webhook.site/<uuid>`).
3. Paste that URL into the device's Custom URL field, click Test Send.
4. Watch the request show up on webhook.site with all headers and body.

This is the **fastest way to integration-test the device** without writing any server code.

## 8.2 Express (Node.js)

```
const express = require('express');
const app = express();

app.use(express.json({ limit: '32kb' }));

app.post('/v1/sensor', (req, res) => {
  // Auth
  if (req.headers.authorization !== 'Bearer YOUR_SECRET') {
    return res.status(401).end();
  }

  const body = req.body;

  // Filter out Test Send probes
  if (body.test === true) {
    console.log(`Test probe from ${body.deviceId}`);
    return res.status(200).json({ ok: true, test: true });
  }

  // Real telemetry — handle each slot/field
  console.log(`${body.ts_iso} ${body.deviceId} (fw ${body.firmware}):`);
  for (const slot of body.slots) {
    for (const f of slot.fields) {
      const v = f.value === null ? 'NULL' : f.value;
      console.log(`  slot ${slot.index} ${slot.name} / ${f.label}: ${v} ${f.unit}`);

      // Example: write to your database here
      // await db.insertReading({
      //   deviceId: body.deviceId,
      //   ts: body.ts,
      //   slotIndex: slot.index,
      //   slotName: slot.name,
      //   fieldKey: f.key,
      //   value: f.value,
      //   unit: f.unit,
      // });
    }
  }

  res.status(200).end();
});

app.listen(3000);
```

## 8.3 FastAPI (Python)

```python
from fastapi import FastAPI, Header, HTTPException
from pydantic import BaseModel
from typing import List, Optional

app = FastAPI()

class Field(BaseModel):
    key: str
    label: str
    unit: str
    value: Optional[float] = None

class Slot(BaseModel):
    index: int
    name: str
    fields: List[Field]

class Payload(BaseModel):
    deviceId: str
    firmware: str
    ts: int
    ts_iso: str
    slots: List[Slot]
    test: Optional[bool] = False

@app.post("/v1/sensor")
def receive(p: Payload, authorization: str = Header(None)):
    if authorization != "Bearer YOUR_SECRET":
        raise HTTPException(401)

    if p.test:
        print(f"Test probe from {p.deviceId}")
        return {"ok": True, "test": True}

    print(f"{p.ts_iso} {p.deviceId} (fw {p.firmware}):")
    for slot in p.slots:
        for f in slot.fields:
            v = "NULL" if f.value is None else f.value
            print(f"  slot {slot.index} {slot.name} / {f.label}: {v} {f.unit}")
            # db.insert(...)

    return {"ok": True}
```

## 8.4 Flask (Python)

```python
from flask import Flask, request, jsonify

app = Flask(__name__)

@app.post('/v1/sensor')
def receive():
    if request.headers.get('Authorization') != 'Bearer YOUR_SECRET':
        return '', 401

    p = request.get_json(force=True)

    if p.get('test'):
        return jsonify(ok=True, test=True)

    for slot in p['slots']:
        for f in slot['fields']:
            v = f['value']
            print(f"slot {slot['index']} {slot['name']} / {f['label']}: {v} {f['unit']}")

    return jsonify(ok=True)
```

## 8.5 Node-RED

1. Drag an **HTTP In** node, set Method to `POST`, URL to `/v1/sensor`.
2. Connect it to a **Function** node:

```javascript
const auth = msg.req.headers.authorization;
if (auth !== 'Bearer YOUR_SECRET') {
  msg.statusCode = 401;
  msg.payload = '';
  return [null, msg];
}
if (msg.payload.test === true) {
  msg.statusCode = 200;
  msg.payload = { ok: true, test: true };
  return [null, msg];
}
// Pass real telemetry downstream
return [msg, null];
```

3. Add an **HTTP Response** node returning the second output (for auth/test acks).
4. Wire the first output to your storage/processing flow (InfluxDB, MQTT republish, etc.).

## 8.6 PHP

```php
<?php
$auth = $_SERVER['HTTP_AUTHORIZATION'] ?? '';
if ($auth !== 'Bearer YOUR_SECRET') {
    http_response_code(401);
    exit;
}

$body = json_decode(file_get_contents('php://input'), true);

if ($body['test'] ?? false) {
    echo json_encode(['ok' => true, 'test' => true]);
    exit;
}

foreach ($body['slots'] as $slot) {
    foreach ($slot['fields'] as $f) {
        $v = $f['value'] ?? 'NULL';
        error_log("slot {$slot['index']} {$slot['name']} / {$f['label']}: $v {$f['unit']}");
        // INSERT INTO readings ...
    }
}

http_response_code(200);
```

## 8.7 cURL (manual probe / debugging)

To send a payload manually that mimics the device:

```bash
curl -X POST https://api.example.com/v1/sensor \
  -H "Content-Type: application/json" \
  -H "X-Device-ID: TEST-DEV" \
  -H "X-Firmware-Version: 2.3.3" \
  -H "Authorization: Bearer YOUR_SECRET" \
  -d '{
    "deviceId":"TEST-DEV",
    "firmware":"2.3.3",
    "ts":1779023398,
    "ts_iso":"2026-05-17T13:09:58Z",
    "slots":[{"index":1,"name":"Test","fields":[{"key":"v","label":"V","unit":"-","value":1.0}]}]
  }'
```

### 8.8 AWS Lambda (Function URL or API Gateway)

```
exports.handler = async (event) => {
  const auth = event.headers.authorization || event.headers.Authorization;
  if (auth !== 'Bearer YOUR_SECRET') {
    return { statusCode: 401, body: '' };
  }

  const body = JSON.parse(event.body);

  if (body.test === true) {
    return { statusCode: 200, body: JSON.stringify({ ok: true, test: true }) };
  }

  // Write to DynamoDB / Timestream / S3 / etc.
  // ...

  return { statusCode: 200, body: '' };
};
```

---

## 9. Troubleshooting

### Test Send shows `✗ FAIL: TypeError: Failed to fetch`

The device is busy with another TLS session (OTA check, ongoing telemetry POST, or MQTT handshake) at the moment you clicked. Wait 5-10 seconds and click again. If it persists, reboot the device.

### Test Send shows `✗ FAIL: URL must start with https://`

You typed an `http://` URL. The device refuses plain HTTP for security. Get a valid TLS certificate (Let's Encrypt is free and easy via Certbot, or use a managed TLS proxy like Cloudflare).

### Test Send shows `✗ FAIL: network error -1`

The device's TLS handshake to your server failed. Possible causes:

- DNS lookup failed (typo in hostname).
- The server is offline or unreachable from the device's network.
- The server's TLS handshake is malformed (wrong cert, expired cert in a way that even `setInsecure()` rejects, unsupported cipher).
- A firewall is blocking outbound HTTPS from the device's IP.

Check from another machine on the same network: `curl -v https://api.example.com/v1/sensor`. If that fails too, the problem is your server, not the device.

### Test Send shows `✗ FAIL: HTTP 401` / `403` / `405`

Your server is reachable but rejected the request. Check:

- The bearer token in the device matches the one your server expects (whitespace, capitalization).

- Your server's route accepts `POST` (some frameworks default to `GET` only).
- Your server's auth middleware reads the header named `Authorization` (case-sensitive in some frameworks).

### Test Send works but no scheduled telemetry arrives

Check the device's serial output for `[CUSTOM]` lines. Expect to see one per Send Interval:

```
[CUSTOM] POST len=532 heap=148452 url=https://api.example.com/v1/sensor
[CUSTOM] OK (200)
```

If you see `[CUSTOM] Skipped: NTP time not valid` — the device's clock isn't synced. Make sure it can reach `pool.ntp.org` (port 123 UDP).

If you see `[CUSTOM] No URL configured` — the URL didn't save. Try the Save flow again.

### Sensor readings show `"value": null`

The sensor itself failed to respond on that cycle. Common causes for Modbus: wrong baud rate or slave ID, broken RS-485 wiring, sensor power dropped out. For analog: ADS1015 chip not detected (check I2C), sensor not connected to AN0/AN2 input.

`null` is correct and intentional — the device explicitly does not invent fake zero readings to fill gaps.

### After 5 failures, WiFi resets

This is by design. If your server is unreachable for 5 consecutive cycles, the device assumes the WiFi side might be the problem and reconnects to clear any stuck state. Telemetry resumes on the next cycle after reconnect.

### Where to see all the device's serial logs

Connect the device's USB port to a computer running a serial terminal at **115200 baud**. The device logs every step of every POST, plus boot info, sensor reads, MQTT activity, OTA checks, etc.

---

## 10. Operational notes

### 10.1 OTA updates still come from DODOTiK

Switching to Custom HTTPS does NOT disable over-the-air firmware updates. Every 24 hours the device checks `https://platform.dodotik.com/api/v1/firmware/latest` for a new firmware version. If you're switching to Custom HTTPS to fully isolate your devices from DODOTiK, note that they will still phone home for OTA. The OTA request includes only:

- The device's `deviceId` (the Device Name you configured).
- The current firmware version string.

No sensor data, no customer URL, no bearer token. If you need 100% no-contact, you'd need to fork the firmware to disable OTA — contact us if that matters.

### 10.2 SD card CSV is kept regardless of mode

The device writes every reading to `/sensors-YYYYMMDD.csv` on its SD card whether you're on MQTT, HTTPS, or Custom HTTPS. So a failed cloud post is recoverable — visit the device's web UI → SD Card panel → Download to retrieve the day's CSV.

### 10.3 Send intervals and battery / data plan

For most use cases 10-15 minutes is a good cadence. If the device is on battery + cellular modem, consider 30-60 minutes. The minimum allowed is 1 minute (mostly for debugging — sustained 1-minute cadence will heat the radio and drain the modem data plan fast).

### 10.4 What happens during the brief TLS-handshake window

The ESP32 web server, MQTT client, OTA checker, and Custom HTTPS poster all share one TLS session at a time, serialized via an internal flag. So you can't have a customer hit the web UI while a Custom HTTPS POST is in flight — the web request will queue or briefly return 409. The whole thing is over in ~200 ms typically.

### 10.5 Adding / removing slots while in Custom HTTPS mode

Configure your sensor slots the same way you would on the DODOTiK platform: in the device's Sensor Configuration page. Saved slot changes take effect on the next scheduled send. Your server will see the new slot's `name`, `index`, and `fields[]` in the very next POST — no separate "schema" call is needed because every payload is self-describing.

### 10.6 Multiple devices with the same custom URL

Perfectly fine. Use `deviceId` (or the `X-Device-ID` header) to tell them apart. The recommended pattern is to give each customer their own Bearer token (one token = one tenant), and within a tenant, use the unique `deviceId` to identify individual devices.

---

## 11. FAQ

**Q: Can the device push to two endpoints at once (DODOTiK MQTT and my server)?** A: No. The device picks exactly one telemetry path per Send Interval. If you need both, replicate from one to the other on your side (e.g., your server forwards to MQTT, or the DODOTiK platform proxies to your URL).

**Q: Can I change the JSON shape?** A: Not without modifying the firmware. The shape in §5 is what the device produces. We chose it to be self-describing so a customer can render readings without a separate schema lookup.

**Q: Why are field labels and units in every payload? Isn't that wasteful?** A: Each payload is about 500-700 bytes. Repeating labels adds maybe 150 bytes per send (×6/hr × 24 h × 365 d ≈ 8 MB/year/device) — trivial. The benefit is your server can decode any device's payload without first fetching a schema; new sensor types just work; relabeling a slot in the device UI propagates without redeploying server code.

**Q: Can I receive over WebSocket instead of HTTP POST?** A: Not in the current firmware. Consider an HTTPS-to-WebSocket bridge on your side.

**Q: Can the device authenticate the server (mutual TLS / pin a certificate)?** A: Not in the current firmware. The device uses one-way TLS with `setInsecure()` (no cert validation). Bearer tokens + hard-to-guess URLs are the recommended way to authenticate.

**Q: What happens if my server is down for hours?** A: The device keeps trying every Send Interval. After 5 consecutive failures it reconnects WiFi and retries. Readings posted during the outage are lost from the cloud's perspective but are preserved on the SD card.

**Q: Can the device send historical readings on recovery?** A: Not automatically over Custom HTTPS. The SD card holds the historical CSV. If you need replay-on-recovery, write a small process on your side that polls `/files` and `/download?file=...` on the device's local IP after an outage.

**Q: I have a custom firmware variant. Will it work?** A: The Custom HTTPS path was added in the `sensor` variant (DODO Sensor Logger). If you're running a different DODO variant (e.g. DODO Command Pro / DODO Logger), check that variant's release notes — the same code is being ported but on its own schedule.

**Q: How do I migrate back to DODOTiK MQTT?** A: Open the web UI → Cloud Settings → switch Protocol to `MQTT — DODOTiK Platform`. Your `Device ID` and `Device Key` need to be set (they were hidden in Custom mode but kept in storage if you were previously on MQTT). Save & Reboot. Your custom URL and bearer are preserved for future use.

---

## Need help?

For firmware bugs or feature requests: open a GitHub issue at https://github.com/dodotik123/DODO_Logger For commercial integrations / fleet onboarding: contact info@dodotik.com

---

*This document covers Custom HTTPS as introduced in firmware v2.3.0 and refined through v2.3.3. Some details may change in future firmware releases — check the README's "Releases" section for version notes.*