

MQTT to Node-RED

Integration

Node-RED is a browser-based visual programming tool built on Node.js, ideal for wiring together MQTT data streams from Meshtastic with downstream services such as dashboards, databases, and messaging platforms. This page covers installing Node-RED, subscribing to Meshtastic MQTT topics, parsing protobuf and JSON payloads, building a simple monitoring dashboard, and forwarding mesh alerts to Telegram and Discord.

Installing Node-RED

On Raspberry Pi / Debian / Ubuntu

Use the official Node-RED install script for Debian/Ubuntu (it provisions a supported Node.js version). Review the script before running it, per nodered.org:

```
bash <(curl -sL https://github.com/node-red/linux-installers/releases/latest/download/update-nodejs-and-nodered-deb)

# Enable and start Node-RED as a service
sudo systemctl enable nodered.service
sudo systemctl start nodered.service

# Node-RED web UI is available at http://<IP>:1880
```

On x86 Linux via npm (Node.js v20+ already installed)

The bare npm method is only for environments where a supported Node.js (v20+) is already present. Install the build prerequisites first:

```
sudo apt install build-essential git curl
sudo npm install -g --unsafe-perm node-red
# Start manually:
node-red
# Or install as systemd service manually
```

Installing Required Palettes

Open the Node-RED web UI at `http://<IP>:1880`, then click the hamburger menu → *Manage Palette* → *Install*. Search for and install:

- `node-red-dashboard` - UI widgets (gauges, charts, text displays).
- `node-red-contrib-protobuf` - decode Meshtastic protobuf payloads.
- `node-red-contrib-telegrambot` - send messages to Telegram.

The Meshtastic project also publishes an official decode node, `@meshtastic/node-red-contrib-meshtastic` (linked from meshtastic.org), which you can install the same way.

Alternatively via CLI

```
cd ~/.node-red
npm install node-red-dashboard node-red-contrib-protobuf node-red-contrib-telegrambot
sudo systemctl restart nodered.service
```

Subscribing to Meshtastic MQTT Topics

Add an **mqtt in** node to your flow:

- **Server:** your broker address and port. (For the Meshtastic public broker `mqtt.meshtastic.org`, supply username `meshdev` / password `large4cats` - it is not anonymous.)
- **Topic:** `msh/#` to receive all Meshtastic traffic, or `msh/US/2/json/LongFast/#` for JSON output from the LongFast channel. JSON is only published when JSON mode is enabled on the gateway (`mqtt.json_enabled true`); note that `msh/#` also yields protobuf-encoded `/e/` topics that the json node cannot parse.

- **QoS:** 0 (at most once) is sufficient for mesh monitoring.
- **Output:** for protobuf `/e/` topics set Output to a *buffer*; for JSON topics a *parsed JSON object* / string is fine.

Parsing Protobuf Payloads

The protobuf format requires `.proto` schema files. The MQTT envelope type `ServiceEnvelope` is defined in `mqtt.proto`, so download that plus its imports (`mesh.proto`, `config.proto`, `portnums.proto`, etc.) into the same directory:

```
mkdir -p ~/.node-red/proto
cd ~/.node-red/proto
wget https://raw.githubusercontent.com/meshtastic/protobufs/master/meshtastic/mqtt.proto
wget https://raw.githubusercontent.com/meshtastic/protobufs/master/meshtastic/mesh.proto
wget https://raw.githubusercontent.com/meshtastic/protobufs/master/meshtastic/config.proto
wget https://raw.githubusercontent.com/meshtastic/protobufs/master/meshtastic/portnums.proto
wget https://raw.githubusercontent.com/meshtastic/protobufs/master/meshtastic/telemetry.proto
```

Flow wiring for protobuf decode. The encrypted `/e/` MQTT payload is **raw binary protobuf** (not base64), so feed the buffer straight into the decode node:

```
[mqtt in (Output: a buffer)] → [protobuf decode (ServiceEnvelope)] → [debug / further processing]
```

Set the **mqtt in** node's Output to a *buffer* and add a **function** node before the protobuf decode that points it at the right schema and type (do **not** base64-decode the payload first):

```
// The /e/ payload is already a raw binary protobuf buffer - pass it straight through.
msg.protofile = '/home/pi/.node-red/proto/mqtt.proto';
msg.protobufType = 'meshtastic.ServiceEnvelope';
return msg;
```

Base64 only applies to the *nested* `decoded.payload` bytes (for example when extracting `TEXT_MESSAGE_APP` text after the envelope is decoded) - not to the MQTT payload itself.

The **protobuf decode** node only deserializes the bytes into a JavaScript object representing the `ServiceEnvelope`. On encrypted channels the inner `packet.decoded` field is *absent* - the payload is still AES-encrypted with the channel PSK, which must be decrypted separately (or enable JSON output on the gateway, which decrypts before publishing).

“ **Note on channel decryption:** The official `@meshtastic/node-red-contrib-meshtastic` package provides a Meshtastic protobuf decode node. The `protobuf decode` node by itself only deserializes the envelope; for encrypted channels you must AES-decrypt with the channel PSK, or enable JSON output mode on the gateway (which the gateway decrypts before publishing) for full access to message text and telemetry.

Parsing JSON Payloads (Recommended for Simplicity)

If JSON output is enabled on your gateway node (`mqtt.json_enabled true`, ESP32 only - nRF52 gateways do not emit JSON), subscribe to the JSON topic and use a **json** node to parse directly - no protobuf library needed:

```
[mqtt in (topic: msh/US/2/json/LongFast/#)] → [json] → [switch] → [handlers]
```

A decoded JSON message is a flat envelope and looks like:

```
{
  "id": 3456789012,
  "channel": 0,
  "from": 2887456789,
  "to": 4294967295,
  "sender": "!abcd1234",
  "type": "text",
  "payload": {
    "text": "Hello from City A!"
  },
  "timestamp": 1714953600
}
```

Note: the JSON envelope does **not** include RF metadata - there is no top-level `rssi`, `snr`, or `hops_away` field. Those values are not available via gateway JSON output; if you need them, obtain them from a local serial/BLE API or from the protobuf `MeshPacket` (`rx_rssi`/`rx_snr`) instead.

Building a Simple Dashboard

With `node-red-dashboard` installed, add a UI group for "Mesh Monitor":

1. Add a **ui_text** node: label "Last Message", wired from the JSON parse output using a function to extract `msg.payload.payload.text`.
2. Add a **ui_chart** node: "Message Rate / min", use an **rbe** (report-by-exception) node plus a counter function to track message frequency.
3. Add a **ui_table** node: "Recent Messages" - build a rolling array of the last 20 messages using a context store.

RF signal gauges (RSSI/SNR) cannot be driven from the JSON MQTT feed because those fields are not present in the JSON envelope. If you want an RSSI display, source the value from a non-MQTT-JSON path (local serial/BLE API, or the protobuf `MeshPacket.rx_rssi` on the `/e/` decode path).

Access the dashboard at `http://<IP>:1880/ui`.

Forwarding Alerts to Telegram

Pre-requisite: create a Telegram Bot via @BotFather and note the bot token and your chat ID.

Wire a flow like this:

```
[mqtt in] → [json] → [switch: msg.payload.type == "text"] → [function: format alert] → [telegram sender]
```

Function node code to format the Telegram message:

```
const p = msg.payload;
if (!p.payload || !p.payload.text) return null; // skip non-text packets

msg.payload = {
  chatId: "YOUR_CHAT_ID_HERE",
  type: "message",
  content: `📶 Mesh Message
` +
  `From: ${p.sender}
` +
  `Text: ${p.payload.text}`
};
return msg;
```

Configure the **Telegram Sender** node with your bot token. (RSSI/SNR/hops are intentionally omitted - they are not present in the JSON envelope.)

Forwarding Alerts to Discord

Use a **http request** node pointed at your Discord webhook URL:

```
[mqtt in] → [json] → [switch: msg.payload.type == "text"] → [function: build webhook body] → [http request]
```

Function node code:

```
const p = msg.payload;
if (!p.payload || !p.payload.text) return null;

msg.method = "POST";
msg.url = "https://discord.com/api/webhooks/YOUR_WEBHOOK_ID/YOUR_WEBHOOK_TOKEN";
msg.headers = { "Content-Type": "application/json" };
msg.payload = JSON.stringify({
  username: "MeshAmerica Bot",
  content: `**Mesh Message** from \`${p.sender}\`
  +
  > ${p.payload.text}`
});
return msg;
```

Example: Full Flow JSON Export

Below is a minimal Node-RED flow (importable via *Import* → *Clipboard*) that subscribes to the JSON MQTT topic, parses messages, displays them in a dashboard, and forwards them to a Discord webhook:

```
[
  {
    "id": "mqtt_in_1",
    "type": "mqtt in",
    "name": "Meshtastic JSON",
    "topic": "msh/US/2/json/LongFast/#",
```

```

"qos": "0",
"broker": "local_broker",
"wires": [["json_parse_1"]]
},
{
"id": "json_parse_1",
"type": "json",
"name": "Parse JSON",
"wires": [["switch_1", "dashboard_table_1"]]
},
{
"id": "switch_1",
"type": "switch",
"name": "Text only",
"property": "payload.type",
"rules": [{"t": "eq", "v": "text"}],
"wires": [["format_discord_1"]]
},
{
"id": "format_discord_1",
"type": "function",
"name": "Format Discord",
"func": "const p = msg.payload;
if (!p.payload||!p.payload.text) return null;
msg.method='POST';
msg.url='https://discord.com/api/webhooks/XXX/YYY';
msg.headers={'Content-Type':'application/json'};
msg.payload=JSON.stringify({username:'MeshBot',content:`**${p.sender}**: ${p.payload.text}`});
return msg;",
"wires": [["http_discord_1"]]
},
{
"id": "http_discord_1",
"type": "http request",
"name": "Discord Webhook",
"method": "POST",
"wires": [[]]
}
]

```

Replace the Discord webhook URL placeholder with your real webhook before importing.

Revision #3

Created 2026-05-03 05:36:32 UTC by Mesh America Admin

Updated 2026-06-10 05:30:19 UTC by Mesh America Admin