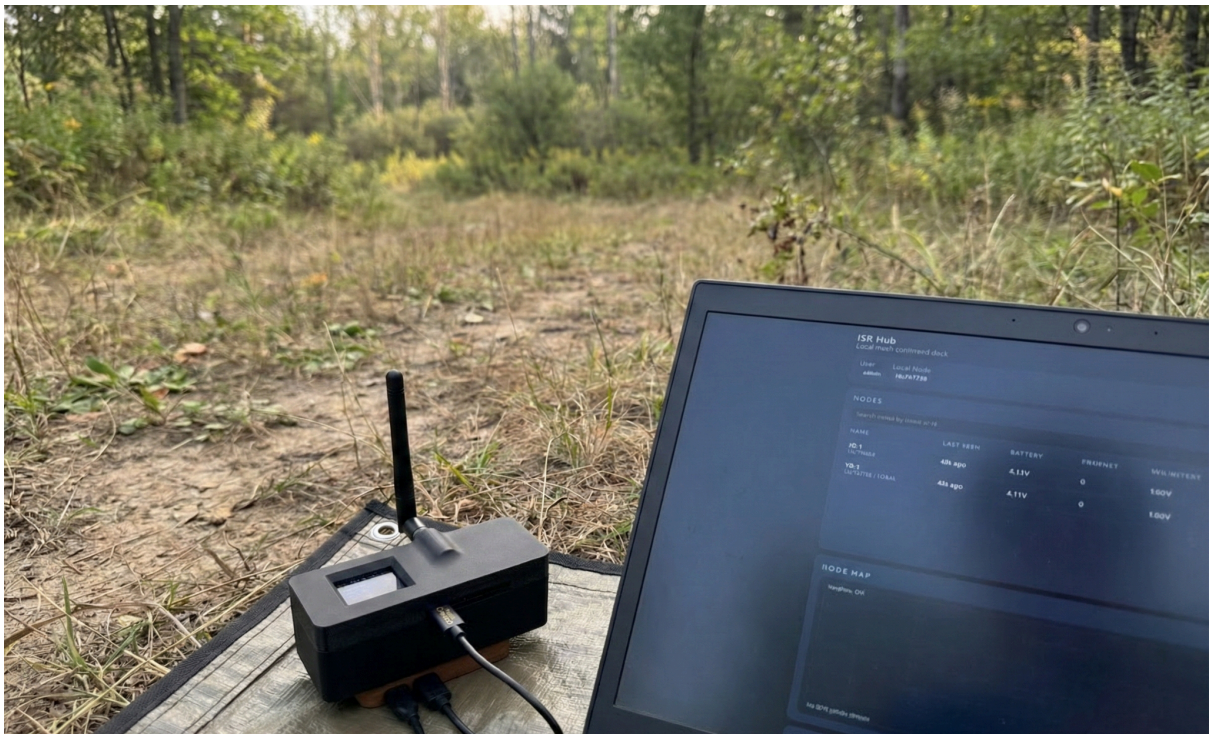
FieldMesh.

Rapid-Deploy Mesh Communications Hub.

Table of Contents

Introduction



The goal of this project was to build a small, self-contained mesh communications and sensing system that can operate without any external network infrastructure. The system is designed to demonstrate how low-power, long-range radios can be combined with local web technologies to collect, relay, and visualize data from distributed devices. A key requirement is that the system remains fully local and offline, while still being accessible from common user devices through a standard web browser.

System Overview and Field Validation

At its core, this system is a way for small, battery-powered devices to talk to each other over long distances without relying on cellular networks, Wi-Fi infrastructure, or the internet. Each device can send short messages and status updates, and the network automatically figures out how to relay those messages across multiple hops until they reach their destination. A central hub can then collect everything the network is doing and present it in a form that is easy for a person to understand.

In practical terms, this means devices can be placed far apart in the field and still communicate as long as there is a chain of nodes between them.

Messages do not need to travel directly from one device to another. Instead, intermediate nodes repeat traffic as needed, allowing the network to extend well beyond the range of a single radio link.

The system was tested using three nodes: two end nodes and one intermediate relay positioned between them. In this configuration, the relay node acted as a hop, forwarding traffic between the two endpoints. Under normal outdoor conditions with a modest elevation difference, the network achieved a maximum tested range of approximately 12 kilometers between the two endpoints. This distance was reached without specialized antennas or towers, using standard LoRa hardware and typical environmental conditions.

During testing, the nodes were able to reliably send and receive text messages across the mesh. Messages originating at one end node were successfully relayed through the intermediate node and received at the far endpoint, confirming correct multi-hop behavior. In addition to messaging, the system continuously logged GPS position data and telemetry from each node, including device status information. This data was collected by the hub and displayed in real time through the local web interface.

These tests demonstrate that the system functions as intended both as a communications network and as a distributed sensing platform. Nodes are able to operate independently in the field, relay traffic for one another, and report their state back to a central hub. The result is a portable, infrastructure-free mesh network that can provide situational awareness, messaging, and telemetry over distances well beyond what short-range wireless technologies typically allow.

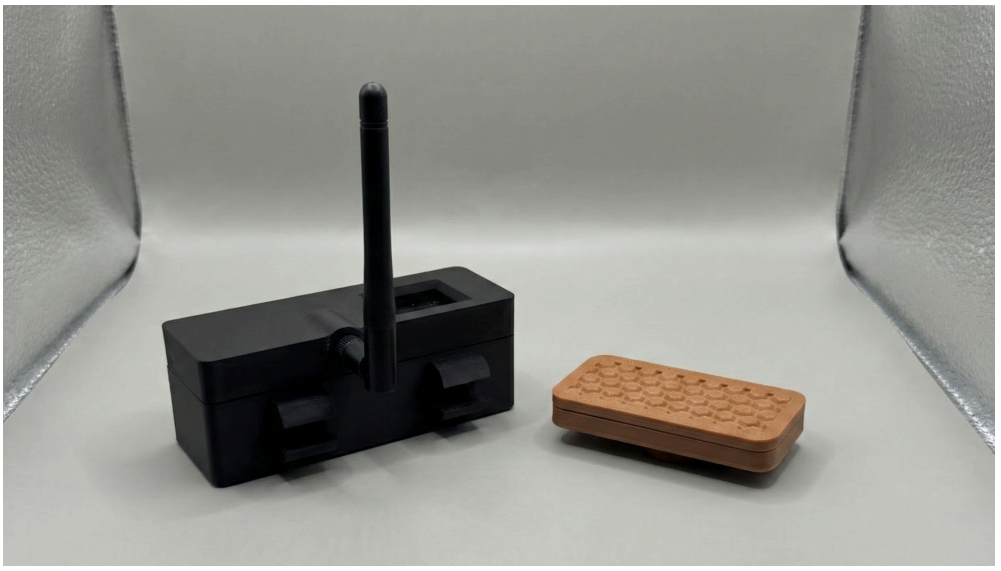
Hardware Overview

The mesh network itself is formed by LilyGO T-Beam devices running Meshtastic firmware. Each T-Beam integrates an ESP32 microcontroller, a LoRa radio, and GPS, allowing nodes to exchange messages and telemetry over a multi-hop LoRa mesh. These nodes act both as data sources and as relays, automatically forwarding traffic for other nodes in the network.

The hub is built around a Raspberry Pi Zero 2 W connected to a T-Beam node over USB. The Pi runs custom hub software that interfaces with the mesh radio and serves a local web dashboard. Any device with Wi-Fi and a web browser can connect to the hub to view node status, telemetry, and

messages, making the system easy to access without specialized client software or an internet connection.

Design Overview



The physical design of the system focuses on durability, modularity, and ease of use, while keeping the overall form factor compact and practical for portability and field use. Both enclosures were designed specifically around their roles in the system and around how they are used together.

T-Beam Enclosure

The T-Beam enclosure was designed as a snug, tool-free protective shell that clips together without screws or external hardware. The internal geometry closely follows the T-Beam PCB, ensuring the board is held firmly in place and protected from movement or impact. The enclosure is printed in ABS and avoids thin, unsupported features, which makes it mechanically strong and resistant to cracking. It has been drop tested and is robust enough to survive typical handling and accidental drops.

Functional access was a key requirement. The case includes cutouts for the USB port, physical buttons, and the onboard display, allowing full use of the device without removing it from the enclosure. Two small external ears are integrated into the design to hold the LoRa antenna securely when it is not actively deployed, reducing strain on the connector and making the unit easier to transport.

A defining feature of the T-Beam enclosure is the integrated bayonet-style connector at the bottom. This provides a standardized mechanical interface for modular attachments. Depending on the use case, this connector can accept a simple blanking plate, a magnetic mount for attaching the device to metal surfaces, or a mating connector on another enclosure.

Raspberry Pi Zero 2 W Hub Enclosure

The Raspberry Pi Zero 2 W enclosure was designed with minimal size and hub-specific operation in mind. Openings are provided only for the two micro-USB ports: one for power input and one for the USB connection to the T-Beam acting as the radio bridge.

Like the T-Beam enclosure, the Pi case is made of two snap-fit halves that can be opened without tools to access the board when needed. The top of the enclosure features a male bayonet connector that mates directly with the connector on the T-Beam case. This allows the Pi hub and the T-Beam to physically lock together into a single unit when operating in hub mode, making it easier to handle, power, and access the local web interface during setup or use.

Together, the two enclosures form a modular physical system that mirrors the software architecture: independent components that can operate on their own, or securely combine into a compact, integrated hub when needed.

Software Architecture Overview

The software component of this project is a self-contained local hub service designed to run on a Raspberry Pi Zero 2W. Its role is to bridge a low-power LoRa mesh network to a secure, offline web dashboard that users can access over local Wi-Fi. The system is intentionally cloud-free and dependency-light, prioritizing reliability, clarity, and ease of deployment in constrained environments.

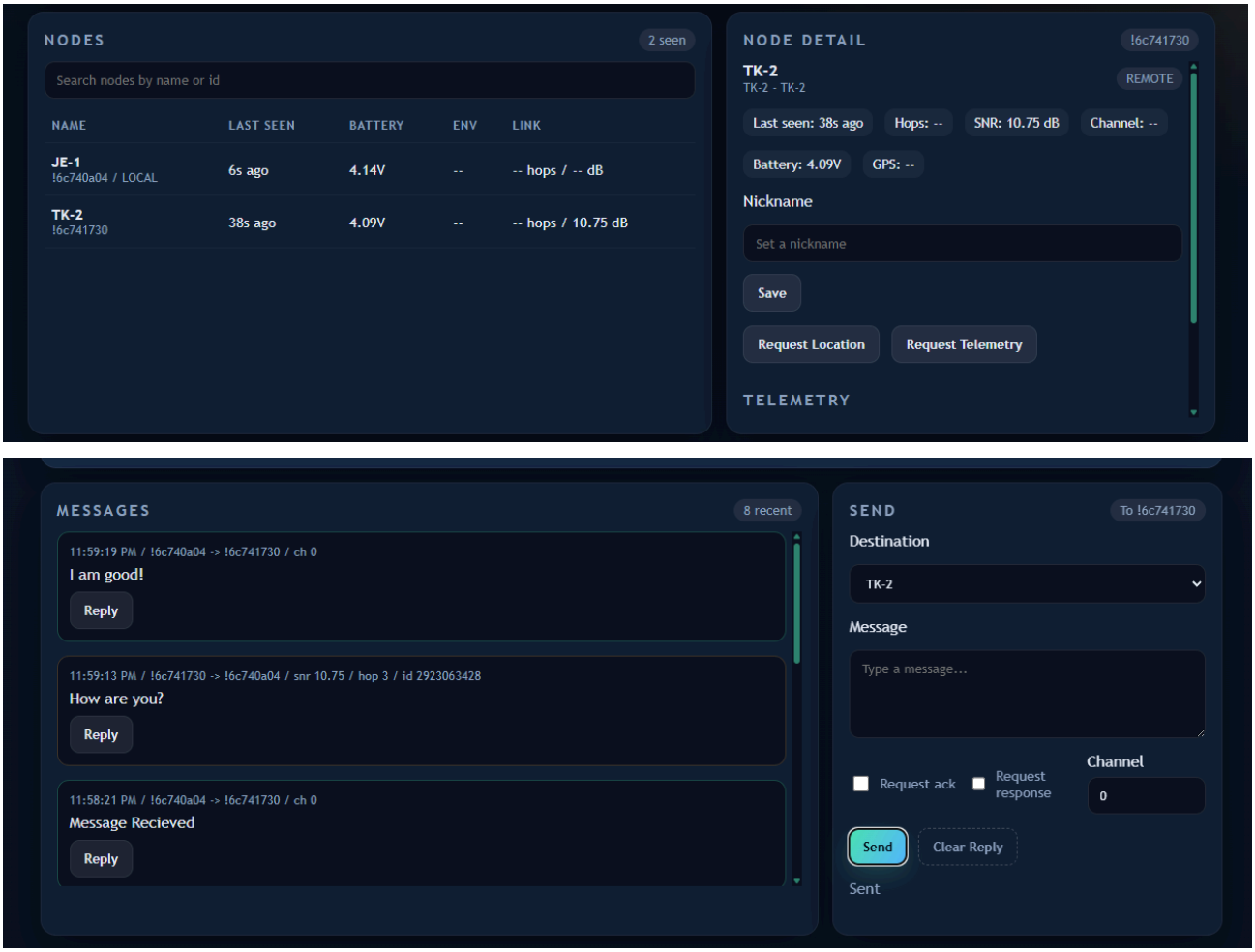
At a high level, the hub software performs four core functions. It interfaces with a Meshtastic radio over USB, maintains live and recent state for all mesh nodes, serves a browser-based dashboard over HTTP, and manages local authentication and session state. All of this logic lives in a single FastAPI application, making the system easy to work with, deploy, and extend.

Backend Service

The backend is a Python FastAPI application implemented in a single entry point, main.py. When started, it binds to 0.0.0.0 on port 8000 and serves both a REST API and a static frontend. The choice of FastAPI provides an asynchronous runtime, structured request handling, and clear separation between internal state and exposed endpoints, while remaining lightweight enough for the low processing power of a Pi Zero 2 W.

Configuration is loaded at startup from config.json, with environment variables allowed to override sensitive values such as the password hash, session secret, serial port, and telemetry port number. This allows the same codebase to be used in development, testing, and deployment without modification.

Meshtastic Integration



The hub connects to a Meshtastic-compatible radio over USB using the meshtastic.serial_interface.SerialInterface. Once connected, it subscribes to incoming packets from the mesh network. These include standard Meshtastic text messages as well as custom telemetry packets sent on a configured application port.

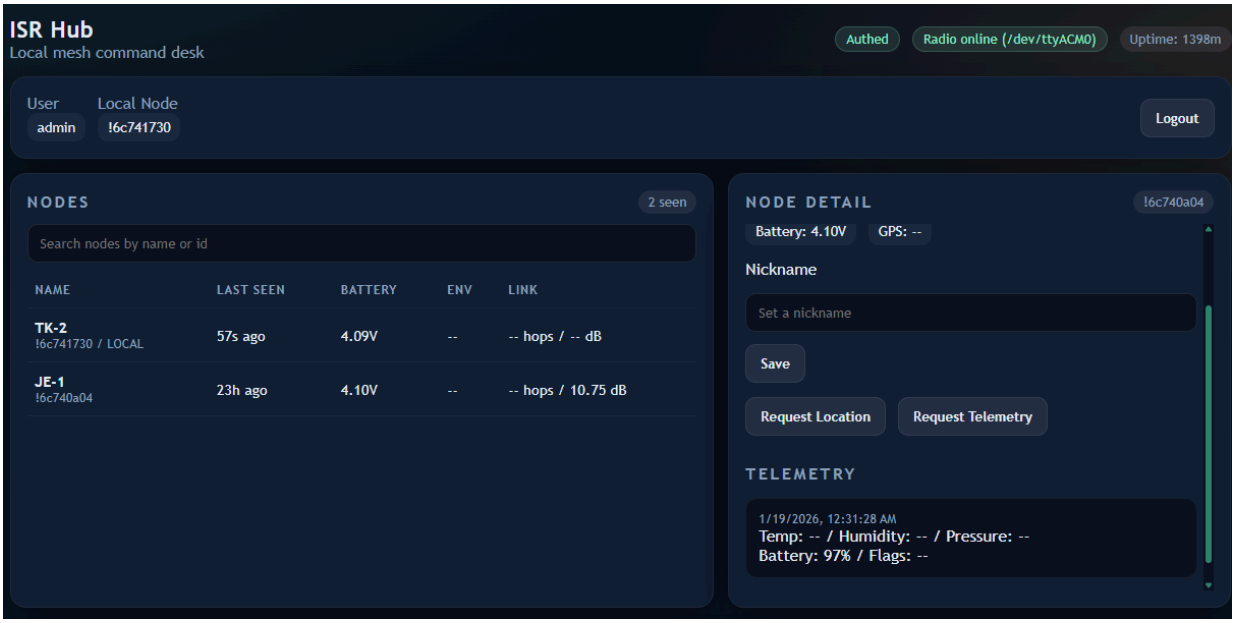
Incoming packets are normalized immediately on receipt. Binary and protobuf-backed Meshtastic objects are converted into JSON-safe Python structures so they can be safely stored, streamed to clients, and serialized by FastAPI without runtime errors. This normalization layer is a key resilience feature, as it prevents API failures caused by unexpected packet contents or schema changes.

Node State Model

The backend maintains an in-memory model representing the current view of the mesh. Each node entry tracks identifiers, optional nicknames, last-seen timestamps, recent telemetry samples, position data, device metrics such as battery level, and basic link statistics when available. Telemetry history and message logs are implemented as rolling buffers whose size is configurable, ensuring predictable memory usage.

This in-memory model is treated as the authoritative live state. All API responses and live updates are derived from it, allowing the frontend to remain simple and stateless.

Persistence Layer



Persistence is optional and enabled via configuration. When active, the backend uses a local SQLite database named hub.db with WAL mode enabled. Telemetry records and message history are written to disk in a lightweight schema designed for append-heavy workloads.

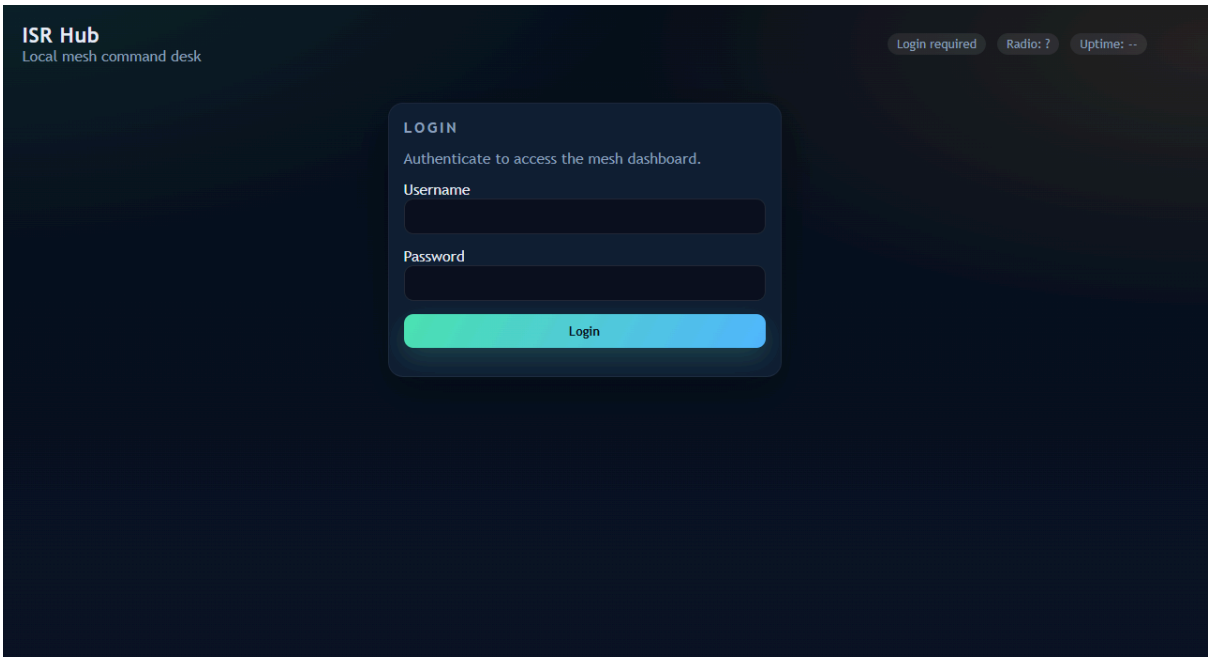
On startup, the application rehydrates its in-memory state from SQLite, allowing the dashboard to show recent history even after a reboot or power interruption. The persistence layer is deliberately minimal and does not attempt to function as a long-term analytics store, aligning with the project's focus on operational awareness rather than data warehousing.

Live Updates

To provide real-time updates without the overhead of WebSockets, the system uses Server-Sent Events. Clients connect to /api/stream and receive a continuous event stream that includes telemetry updates, node presence changes, new messages, and hub status changes. This approach works well on low-power hardware and integrates cleanly with standard browser APIs.

If a client cannot maintain an SSE connection, the frontend falls back to periodic polling of REST endpoints, ensuring basic functionality even in degraded conditions.

Authentication and Sessions



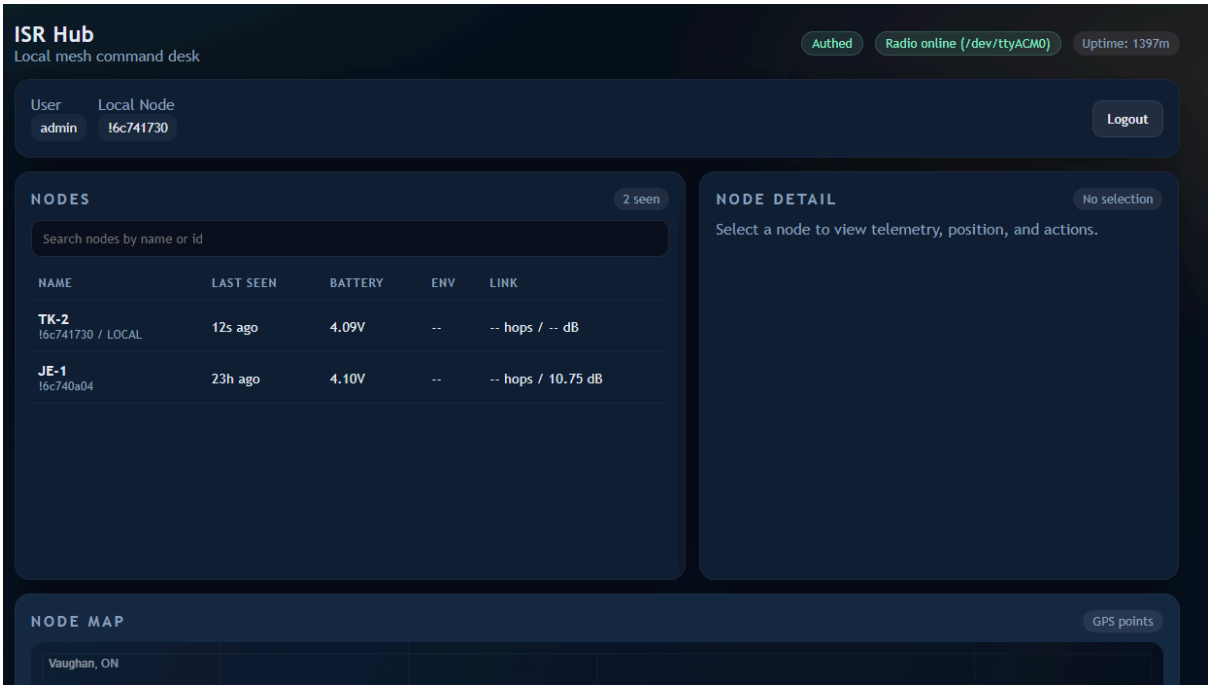
The hub implements a simple single-user authentication model suitable for a local, offline system. Login is handled via /auth/login, which verifies credentials against a password hash generated using Python's standard library scrypt implementation. This avoids the need for heavy native dependencies while still providing a memory-hard key derivation function.

Authenticated sessions are tracked using a signed cookie created with itsdangerous. The session secret and password hash are stored in configuration or environment variables and never hard-coded. There is no account creation flow, role system, or external identity provider, by design.

API Surface

The REST API exposes a small, focused set of endpoints that mirror the hub's internal responsibilities. Clients can query live node summaries and per-node details, retrieve recent message history, and check hub health and radio connectivity. The API also supports sending text messages into the mesh, requesting telemetry or position updates from specific nodes, and assigning local nicknames that are stored only on the hub.

Frontend Dashboard



The frontend is a single-page HTML and JavaScript application served directly by the backend. It has no external CDN dependencies and is fully self-contained, allowing the hub to operate without internet access. The UI uses standard browser APIs, fetch for REST calls, and SSE for live updates.

The dashboard presents a node table with live status indicators, a detail panel for inspecting telemetry and metadata, and a message view that supports sending and replying to mesh messages. A lightweight canvas-based map plots GPS positions using a fixed bounding box centered on local regions in Ontario. This avoids the complexity and resource cost of full mapping libraries while still providing spatial awareness.