04 - RouteTrack Pi — GPS Data Logging Service

Date: December 24th, 2025
Category: Raspberry Pi / GPS / Logging / Linux Services

Backlink: RouteTrack Pi — gpsd Installation & GPS Validation

Project Goal

This page adds the next layer on top of the now-stable GPS subsystem:

- Continuously **collect TPV updates** from `gpsd` (localhost port **2947**)
- Write the data to a **local database** for later:
  - Route mapping (GeoJSON export)
  - Mileage calculation
  - Stop detection / time-on-site
  - Daily totals (hours, distance, etc.)

This is a **local-first** design so the system still works even when the truck is offline.

Prerequisites

This page assumes:

- `gpsd-standalone.service` is enabled and running
- `gpsd` is listening on **127.0.0.1:2947**
- `gpspipe -w -n 25` shows TPV + SKY messages and `mode: 3` once a fix is obtained

---

Data We Will Log (Minimum Viable Dataset)

From `TPV` messages, we will store:

- `time` (UTC timestamp from gpsd)
- `lat`, `lon`
- `alt` (optional)
- `speed`
- `track` (heading)
- `mode` (0/1/2/3 — we only trust **3** for real routes)
- `epx`, `epy`, `eps` (accuracy-ish fields if present)

This is enough to:

- Render a map route
- Compute distance
- Identify movement vs stops
- Generate daily summaries

Perfect — this is exactly the right moment to clean this up □□
Below is a **fully copy-pasteable replacement** for **both sections** you showed: **Folder Layout** and **Install Dependencies**, rewritten to match the **venv + dashboard** direction and your clean BookStack style.

You can drop this in **as-is** and delete what's there now.

Folder Layout

This project uses a structured directory layout under `/opt/routetrack` to keep scripts, data, logs, configuration, and the Python virtual environment clearly separated.

Create the directory structure:

```
sudo mkdir -p /opt/routetrack/{bin,data,logs,config}
sudo chown -R $USER:$USER /opt/routetrack
```

Directory Purpose

- **bin/**
  Application scripts and executables
  (e.g. GPS logger, web dashboard entrypoints)
- **data/**
  Persistent application data
  (SQLite database, exports)
- **logs/**
  Application logs
  (GPS logging, web service logs)
- **config/**
  Configuration files
  (database schema, environment variables)
- **venv/**
  Python virtual environment
  (created in a later step)

Reference Paths

These paths are used consistently throughout the RouteTrack project:

- **Logger script:**
  `/opt/routetrack/bin/routetrack-logger.py`
- **SQLite database:**
  `/opt/routetrack/data/routetrack.sqlite`
- **Application logs:**
  `/opt/routetrack/logs/routetrack.log`
- **Database schema / config files:**
  `/opt/routetrack/config/schema.sql`

Install Dependencies

RouteTrack runs on **Raspberry Pi OS Lite 64 bit** and uses a Python virtual environment to avoid modifying the system Python installation.

Update package lists:

```
sudo apt update
```

Install required system packages:

```
sudo apt install -y python3-venv python3-pip sqlite3
```

Package Purpose

- **python3-venv**
  Creates an isolated Python environment for RouteTrack
- **python3-pip**
  Installs Python packages inside the virtual environment
- **sqlite3**
  Lightweight local database for GPS data and summaries

Using a virtual environment avoids conflicts with the OS-managed Python environment and allows RouteTrack to safely install web and dashboard dependencies.

Nice — and you're right to call that out. Your BookStack section should explicitly include the **venv package install** step since that's where you're at now.

Here's a **drop-in BookStack section** you can paste **immediately after "Install Dependencies"** (or as the last part of it). It documents exactly what you did, cleanly.

Create the Python Virtual Environment (venv)

RouteTrack uses a dedicated Python virtual environment stored under `/opt/routetrack/venv`.
This avoids installing packages into the OS-managed Python environment and keeps the project portable and stable.

Create the virtual environment:

```
python3 -m venv /opt/routetrack/venv
```

Install Web Dashboard Dependencies (Flask + Gunicorn)

Upgrade `pip` inside the virtual environment:

```
/opt/routetrack/venv/bin/pip install --upgrade pip
```

Install the local dashboard requirements:

```
/opt/routetrack/venv/bin/pip install flask gunicorn
```

Verify Flask Works

Run a quick import test:

```
/opt/routetrack/venv/bin/python -c "import flask; print('Flask OK')"
```

Expected output:

```
zippyb@pi-gps:~ $ /opt/routetrack/venv/bin/python -c "import flask; print('Flask OK')"
Flask OK
```

Current Status

At this point:

- Folder layout exists under `/opt/routetrack`
- Python virtual environment is created at `/opt/routetrack/venv`
- Flask + Gunicorn are installed and verified successfully

The next phase will initialize the SQLite database schema and begin logging `gpsd` TPV points into the database for mapping and reporting.

Creating the SQLite Database Schema

This project uses a **file-based SQL schema** to define the RouteTrack database structure. Storing the schema in a dedicated `.sql` file makes it easier to review, document, and extend later

as new features (stops, daily summaries, exports) are added.

Creating the Schema File

Create a schema file in the RouteTrack configuration directory:

```
sudo nano /opt/routetrack/config/schema.sql
```

Paste the following contents:

```
PRAGMA journal_mode=WAL;

CREATE TABLE IF NOT EXISTS gps_points (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  ts TEXT NOT NULL,
  lat REAL,
  lon REAL,
  speed REAL,
  track REAL,
  alt REAL,
  mode INTEGER,
  epx REAL,
  epy REAL,
  eps REAL
);

CREATE INDEX IF NOT EXISTS idx_gps_points_ts
  ON gps_points(ts);
```

Save and exit:

- `CTRL + O` → Enter or `CTRL + S`
- `CTRL + X`

Applying the Schema to the Database

Run the schema file once to initialize the SQLite database:

```
sqlite3 /opt/routetrack/data/routetrack.sqlite \
  < /opt/routetrack/config/schema.sql
```

If the database file does not already exist, SQLite will create it automatically.

Verify Database Creation

Confirm that the table was created successfully:

```
sqlite3 /opt/routetrack/data/routetrack.sqlite ".tables"
```

Expected output:

```
zippyb@pi-gps:~ $ sqlite3 /opt/routetrack/data/routetrack.sqlite ".tables"
gps_points
```

To inspect the table structure:

```
sqlite3 /opt/routetrack/data/routetrack.sqlite ".schema gps_points"
```

Why This Approach

SQLite was chosen because it is:

- **Lightweight** and ideal for Raspberry Pi hardware
- **Reliable** for long-running, vehicle-mounted deployments
- **Easy to export** later to GeoJSON or CSV
- Well-suited for **daily summaries** and route analysis

Using a standalone schema file keeps database changes explicit and versionable, which aligns with the long-term goal of expanding RouteTrack into a full route-tracking and reporting system

---

Create the RouteTrack Logger Script (SQLite + gpsd)

This logger is responsible for continuously collecting GPS position updates from `gpsd` (localhost port `2947`) and storing them in the local SQLite database.

Design Notes (Why this works well on a vehicle Pi)

- Connects to `gpsd` over TCP (`127.0.0.1:2947`) using newline-delimited JSON
- Stores only `TPV` class updates (time/position/speed/heading)
- Commits inserts in small batches to reduce SD card write amplification
- Runs under `systemd` so it starts on boot and self-heals if `gpsd` restarts

---

Create the Logger Script File

Create/edit the logger script:

```
sudo nano /opt/routetrack/bin/routetrack-logger.py
```

Paste the following script:

```
#!/usr/bin/env python3
"""
RouteTrack GPS Logger
---------------------

Purpose:
  - Connect to gpsd (localhost:2947)
  - Subscribe to JSON streaming (WATCH)
  - Extract TPV messages (Time/Position/Velocity)
  - Insert points into SQLite (gps_points table)
  - Commit periodically for SD-card friendly writes
  - Print logs to stdout so systemd journald captures them

Key assumptions:
  - gpsd is already running as gpsd-standalone.service and listening on port 2947
  - SQLite DB exists at /opt/routetrack/data/routetrack.sqlite
  - Table gps_points exists (created by schema.sql)
"""
```

```
import json
import socket
import sqlite3
import time
from datetime import datetime, timezone


# gpsd host/port (your standalone service binds gpsd to localhost:2947)
GPSD_HOST = "127.0.0.1"
GPSD_PORT = 2947

# SQLite database path created earlier
DB_PATH = "/opt/routetrack/data/routetrack.sqlite"

# Commit every N points:
# - Reduces disk writes vs committing each insert
# - Helps SD card longevity in vehicle deployments
COMMIT_EVERY = 10


def utc_now() -> str:
    """Return a UTC timestamp string for logging."""
    return datetime.now(timezone.utc).isoformat()


def db_connect() -> sqlite3.Connection:
    """
    Open SQLite connection and apply performance/safety pragmas.

    - WAL (Write-Ahead Logging) mode is already enabled via schema.sql,
      but repeating it here is harmless.
    - synchronous=NORMAL is a common setting for WAL mode:
      better performance with good durability.
    """
    conn = sqlite3.connect(DB_PATH, timeout=30)
    conn.execute("PRAGMA journal_mode=WAL;")
    conn.execute("PRAGMA synchronous=NORMAL;")
    return conn
```

```
def insert_point(cur: sqlite3.Cursor, tpv: dict) -> None:
    """
    Insert a TPV message into gps_points.

    We use tpv.get(...) so missing keys do not crash the logger.
    This keeps the service robust when gpsd emits partial data during fix acquisition.
    """
    cur.execute(
        """
        INSERT INTO gps_points (ts, lat, lon, speed, track, alt, mode, epx, epy, eps)
        VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
        """,
        (
            tpv.get("time"),   # gpsd's UTC timestamp (string)
            tpv.get("lat"),    # latitude (float)
            tpv.get("lon"),    # longitude (float)
            tpv.get("speed"),  # speed (typically meters/second)
            tpv.get("track"),  # heading/course (degrees)
            tpv.get("alt"),    # altitude (meters)
            tpv.get("mode"),   # 0/1/2/3 (3 = best fix quality)
            tpv.get("epx"),    # estimated longitude error (meters)
            tpv.get("epy"),    # estimated latitude error (meters)
            tpv.get("eps"),    # estimated speed error
        ),
    )



def main() -> None:
    """
    Main loop:
      - Connect to SQLite
      - Forever:
          - Connect to gpsd
          - Send WATCH request to enable JSON streaming
          - Read gpsd lines (newline-delimited JSON)
          - Store TPV messages to SQLite
      - On disconnect/errors:
          - commit anything pending
          - sleep briefly
          - reconnect
```

```
"""
print(f"{utc_now()} RouteTrack logger starting...", flush=True)

# Create DB connection and cursor once.
# SQLite is local, fast, and lightweight for Pi deployments.
conn = db_connect()
cur = conn.cursor()

# Count uncommitted inserts so we can batch commits.
pending = 0

while True:
    try:
        # Establish TCP connection to gpsd service.
        with socket.create_connection((GPSD_HOST, GPSD_PORT), timeout=10) as s:

            # Enable JSON output streaming from gpsd.
            # gpsd will emit multiple classes (TPV, SKY, etc.); we filter for TPV.
            s.sendall(b'?WATCH={"enable":true,"json":true}\n')

            # gpsd responses arrive in chunks; accumulate until newline.
            buf = b""

            while True:
                chunk = s.recv(4096)
                if not chunk:
                    # Socket closed; force reconnect
                    raise RuntimeError("gpsd socket closed")

                buf += chunk

                # Process all complete lines currently buffered.
                while b"\n" in buf:
                    line, buf = buf.split(b"\n", 1)

                    if not line.strip():
                        continue

                    # Convert bytes -> string -> JSON dict
                    try:
```

```
                msg = json.loads(line.decode("utf-8", errors="replace"))
            except json.JSONDecodeError:
                # Skip malformed lines without crashing
                continue

            # Only store TPV messages (position/time/speed)
            if msg.get("class") != "TPV":
                continue

            # Skip TPV messages without time.
            # This can occur before a real fix is established.
            if "time" not in msg:
                continue

            # Insert into SQLite
            insert_point(cur, msg)
            pending += 1

            # Commit every N points to reduce write load
            if pending >= COMMIT_EVERY:
                conn.commit()
                pending = 0

    except Exception as e:
        # If gpsd restarts, USB hiccups, or anything breaks, we reconnect.
        # Commit any pending inserts first (best effort).
        try:
            conn.commit()
        except Exception:
            pass

        print(f"{utc_now()} ERROR: {e} (reconnecting in 3s)", flush=True)
        time.sleep(3)


if __name__ == "__main__":
    main()
```

Make the script executable:

```
sudo chmod +x /opt/routetrack/bin/routetrack-logger.py
```

Create the systemd Service (RouteTrack Logger)

This service ensures the logger starts at boot, stays running, and is tied to the GPS subsystem.

Create the unit file:

```
sudo nano /etc/systemd/system/routetrack-logger.service
```

Paste:

```
[Unit]
Description=RouteTrack GPS Logger
# Start after gpsd is online and networking is available
After=gpsd-standalone.service network.target
# Pull gpsd up if needed, and fail if gpsd is missing
Wants=gpsd-standalone.service
Requires=gpsd-standalone.service

[Service]
Type=simple

# Run from /opt/routetrack so relative paths (if added later) behave predictably
WorkingDirectory=/opt/routetrack

# IMPORTANT:
# Use the virtual environment Python so packages (Flask, etc.) remain isolated
ExecStart=/opt/routetrack/venv/bin/python /opt/routetrack/bin/routetrack-logger.py

# Always restart if the logger exits (gpsd restarts, USB dropouts, etc.)
Restart=always
RestartSec=3

# Send script output to journald
```

```
StandardOutput=journal
StandardError=journal

[Install]
WantedBy=multi-user.target
```

Reload systemd + enable and start the service:

```
sudo systemctl daemon-reload
sudo systemctl enable --now routetrack-logger.service
```

Check service status:

```
sudo systemctl status routetrack-logger.service --no-pager -l
```

View logs live:

```
sudo journalctl -u routetrack-logger -f
```

Verify GPS Data is Being Written to SQLite

Confirm row count is increasing:

```
sqlite3 /opt/routetrack/data/routetrack.sqlite "SELECT COUNT(*) FROM gps_points;"
```

View the latest 10 points:

```
sqlite3 /opt/routetrack/data/routetrack.sqlite \
"SELECT ts, lat, lon, speed, mode FROM gps_points ORDER BY id DESC LIMIT 10;"
```

Optional: verify you are receiving `mode: 3` fixes consistently:

```
sqlite3 /opt/routetrack/data/routetrack.sqlite \
"SELECT mode, COUNT(*) FROM gps_points GROUP BY mode ORDER BY mode;"
```

Notes for Later Phases (Mileage + Stops)

- gpsd `speed` values are typically **meters/second**
  - mph conversion: `mph = mps * 2.23694`
- Some “movement” may appear when parked due to GPS drift.
  - Mileage calculations should apply filtering later:
    - count movement only when `mode = 3`
    - ignore points under a speed threshold (example: `>= 0.5 m/s`)

Log Rotation (Prevent SD Card Bloat)

RouteTrack services write their runtime output to **systemd journald** (viewable via `journalctl`). journald handles rotation automatically and is capped by the retention settings configured in `/etc/systemd/journald.conf`.

In addition, a `logrotate` policy is created for any **file-based logs** that may be added later under `/opt/routetrack/logs/` (for example: helper scripts, exporters, or future components that write `.log` files).

Create the logrotate Policy (File Logs)

Create a logrotate config for RouteTrack:

```
sudo nano /etc/logrotate.d/routetrack
```

Paste:

```
/opt/routetrack/logs/*.log {
  daily
  rotate 14
  compress
  delaycompress
```

```
    missingok
    notifempty
    copytruncate
}
```

Fix: logrotate “Insecure Permissions” Error

logrotate may refuse to rotate logs if the parent directory is writable by non-root users (reported as “insecure permissions”). To resolve this securely, the RouteTrack logs directory is locked down to root ownership:

```
sudo chown root:root /opt/routetrack/logs
sudo chmod 755 /opt/routetrack/logs
```

Verify permissions:

```
ls -ld /opt/routetrack /opt/routetrack/logs
```

Test logrotate

Force a rotation to validate the config:

```
sudo logrotate -f /etc/logrotate.d/routetrack
```

RouteTrack Logging Note (Primary Logging)

The RouteTrack logger service outputs to **journald**, so you can view logs with:

```
sudo journalctl -u routetrack-logger -f
```

This is the primary logging method. File-based log rotation is included for future scripts or components that write to `/opt/routetrack/logs/*.log`.

Current Status

- File-based RouteTrack logs are rotated daily
- Old logs are compressed and retained safely
- Disk usage remains controlled for long-running deployments

Next Steps

Next phase will build the actual “route intelligence”:

- **Mileage calculation**
  - Haversine distance between points
  - Only count `mode: 3` fixes
  - Ignore drift when speed is near zero
- **Stop detection**
  - Define stop events (speed threshold + dwell time)
  - Write stop events to a second table
- **Daily summaries**
  - Total mileage
  - Total moving time
  - Total stopped time (time-on-site)
  - Start time / end time per shift

Revision #4
Created 25 December 2025 01:18:04 by Nate Nash
Updated 25 December 2025 18:15:28 by Nate Nash