

Deploying Spiti — for your engineering team

*This is the technical deployment guide, written for engineers who run servers. It provisions the **actual delivered stack**. If you have never opened a terminal, you want **RUN.md** instead — a first-timer's click-through on a managed host. This document and **RUN.md** ship together; they describe two ways to run the same software, not two different products.*

0. What you are standing up

One long-running Node process (`node server.js`) listening on `PORT` (default 3000), fronted by a reverse proxy that terminates TLS. State is a single SQLite file (`data/spiti.db` , WAL mode) on the server's own disk. No external database service is required to run. Auth is passwordless magic-link; email is `dev` mode (links shown on screen) until pointed at a real provider. The flight feed is `simulated` and labeled as such in-app until a provider key is set.

The delivered stack, precisely:

Layer	What shipped
Runtime	Node.js ≥ 18 (ESM, <code>"type": "module"</code>)
Web	Express 4, server-rendered HTML
Data	<code>better-sqlite3</code> on a local file (WAL)
Auth	Magic-link (passwordless); a demo role-picker layered on top
Flight feed	Adapter interface; <code>simulated</code> default, FlightAware/Cirium via config
Dependencies	<code>express</code> , <code>better-sqlite3</code> , <code>cookie-parser</code> , <code>nanoid</code> (see <code>package.json</code>)

One honesty note. What was **built and delivered** is the Node/Express/SQLite single-server stack above. This guide provisions what exists. §9 documents the Postgres migration when you outgrow the single box — the target scale route for concurrent write volume and multiple app instances. A deployment guide describes the software you received, not the software someone might have written greenfield.

1. VPS baseline

- **OS:** any current LTS Linux (Ubuntu 22.04/24.04 LTS is the reference).
- **Sizing (illustrative):** 2 vCPU / 4 GB RAM / 40 GB SSD comfortably runs one instance with the SQLite file and headroom. The load driver is concurrent ops users and flight-poll jobs, not raw page traffic. Scale the box vertically first (more RAM/CPU) before reaching for \$9.
- **Build toolchain:** `better-sqlite3` is a native addon. Provision `build-essential` and `python3` (`apt-get install -y build-essential python3`) so `npm ci` can compile it, or rely on its prebuilt binary for common Node/glibc combos (the build tools are the safe fallback).
- **Node:** 18 LTS or newer (`package.json` engines: `>=18.0.0`). Pin the major version (nodesource or nvm) so a host upgrade doesn't move it under you.

2. Install & first boot

```
# as a non-root deploy user, in the app directory
npm ci                # exact deps from package-lock
cp .env.example .env  # then edit .env – see §5. NEVER commit .env (it's gitignored)
node server.js        # first boot auto-migrates the schema and seeds the demo if
empty
```

First-boot behavior (`server.js`): it runs `migrate()` to create/upgrade the schema, and if there are zero events it seeds the LV Cruise 2027 demo so the app is never empty. `npm run reset` wipes and reseeds from scratch.

For a **clean production instance with no demo data**: boot once, confirm, then `DELETE` the seeded rows through the admin surface or a SQL session, and load your real events; or add a `migrate-only` entry point (the team's call — today's documented behavior is first-boot seed).

Schema migrations are additive and idempotent. `migrate()` runs on every boot: it creates any missing tables, adds any missing columns (guarded, a no-op if present), and performs one guarded table rebuild to relax a single legacy `NOT NULL` (see below). Re-running it never destroys data and never rewrites an existing row. Upgrading a running instance is: deploy the new code, restart — the migration runs on boot.

***Note on the one table rebuild.** The `messages.event_guest_id` column was shipped `NOT NULL` in the first release; a later feature (participant message lanes) needs it nullable for general event threads. SQLite cannot drop a `NOT NULL` with `ALTER`, so `migrate()` performs the standard SQLite "create new / copy / drop / rename" once, copying every existing row verbatim and changing only the column's nullability. It is guarded (runs only if*

the column is still `NOT NULL`) and idempotent. Take a backup before the first upgrade boot (§8) as a matter of habit; the rebuild is transactional and loses no rows, but a backup before any migration is good discipline.

3. Process management

The process must survive crashes, reboots, and deploys. Pick one:

systemd (recommended – zero extra dependencies)

`/etc/systemd/system/spiti.service`:

```
[Unit]
Description=Spiti by OC Travel – Event Concierge Portal
After=network.target

[Service]
Type=simple
User=spiti
WorkingDirectory=/opt/spiti
EnvironmentFile=/opt/spiti/.env
ExecStart=/usr/bin/node server.js
Restart=always
RestartSec=3
# Harden a little:
NoNewPrivileges=true
ProtectSystem=full
ProtectHome=true

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

```
sudo systemctl daemon-reload
sudo systemctl enable --now spiti
sudo systemctl status spiti
journalctl -u spiti -f          # logs
```

pm2 (if the team already runs it)

```
pm2 start server.js --name spiti
pm2 save
pm2 startup # boot persistence
```

One instance per SQLite file. `better-sqlite3` is a single-writer embedded database. Do **not** run multiple pm2 cluster instances against the same file. One process per DB file. When you need more than one app instance, that is the signal to move to Postgres (§9).

4. Domain, TLS, reverse proxy

Bind the Node app to `127.0.0.1:3000` (via `PORT`) so it is not directly reachable; put a proxy in front to terminate TLS.

Caddy (simplest – automatic Let's Encrypt)

`/etc/caddy/Caddyfile`:

```
spiti.octravel.com {
  reverse_proxy 127.0.0.1:3000
}
```

Caddy provisions and renews TLS automatically. `sudo systemctl reload caddy`.

nginx (if standardized on it)

```
server {
  listen 443 ssl;
  server_name spiti.octravel.com;

  # ssl_certificate ... (via certbot / Let's Encrypt)

  location / {
    proxy_pass http://127.0.0.1:3000;
    proxy_set_header Host $host;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header X-Forwarded-Proto $scheme;
  }
}
```

MADE BY KALDUNE

Use certbot for the certificate and auto-renew.

- **DNS:** an A/AAAA record for the hostname → the VPS IP.
- **APP_URL:** set it in `.env` to the exact public `https://` origin, no trailing slash. Magic-link sign-in URLs are built from it; a wrong value points links at localhost.
- **Health check:** the app serves `GET /healthz → 200 "ok"`. Point the proxy / monitor / load-balancer probe there.

5. Secrets & environment (`.env`, never in source)

Every setting is read from the environment. `.env.example` documents them all; `.gitignore` excludes the real `.env`. **A real secret is never committed; every key is rotated at handover so nothing the studio ever saw stays live.**

Var	Purpose	Production note
<code>PORT</code>	Listen port	Bind behind the proxy; default 3000
<code>SESSION_SECRET</code>	Signs login sessions	Generate fresh: <code>openssl rand -hex 32</code> . Rotating it logs everyone out (expected). See the encryption note below.
<code>APP_URL</code>	Public origin for magic-link URLs	Must equal the real <code>https://</code> domain, no trailing slash
<code>EMAIL_PROVIDER</code>	<code>dev</code> <code>postmark</code> <code>sendgrid</code>	<code>dev</code> shows links on screen; set a real provider for production sign-in (§6)
<code>EMAIL_API_KEY</code> / <code>EMAIL_FROM</code>	Transactional email creds + sender	Sender on an OC-Travel-owned domain with SPF/DKIM/DMARC
<code>FLIGHT_PROVIDER</code>	<code>simulated</code> <code>flightaware</code> <code>cirium</code>	<code>simulated</code> (labeled in-app) until a real key is set (§7)
<code>FLIGHT_API_KEY</code>	FlightAware AeroAPI / Cirium key	Adapter reads it; no code change
<code>SMS_PROVIDER</code> / <code>TWILIO_*</code>	Optional SMS/WhatsApp notifications	Optional

Var	Purpose	Production note
<code>DATABASE_FILE</code>	SQLite file path	Default <code>data/spiti.db</code> ; see §8 for placement, §9 for Postgres

`chmod 600 .env`, owned by the deploy user. If the host offers a secrets manager, inject these as environment variables at process start (systemd `EnvironmentFile` or the manager's agent) rather than a file on disk.

Encryption-at-rest note — read before rotating `SESSION_SECRET`. Passport / PNR / DOB are encrypted at rest inside the DB (`encField` / `decField` in `src/db.js`, AES-256-GCM). In this release the at-rest key **derives from** `SESSION_SECRET`. Consequence: rotating `SESSION_SECRET` makes existing encrypted fields unreadable. For production, move to a dedicated `DATA_ENCRYPTION_KEY` / KMS **before** rotating that secret independently — the call sites do not change, only where the key comes from. Flag this to whoever owns secret rotation so a well-meaning rotation does not strand encrypted PII.

6. Email / SMTP (magic-link + notifications)

Production sign-in needs real email; `dev` mode only prints the link on screen.

1. Choose a transactional provider (**Postmark** or **SendGrid** are the documented adapters). Create an account; verify a **sending domain OC Travel owns**; set SPF, DKIM, and DMARC on that domain's DNS. Auth email must originate from a domain the client controls.
2. Set `EMAIL_PROVIDER=postmark` (or `sendgrid`), `EMAIL_API_KEY=<key>`, `EMAIL_FROM=concierge@<owned-domain>`. Restart. No code change.
3. Verify: trigger a magic link to a real inbox; confirm delivery and that the link's origin equals `APP_URL`.

7. Flight-data integration (labeled where simulated)

- **Default:** `FLIGHT_PROVIDER=simulated` — a realistic in-process simulation drives the delay-propagation feature, **labeled "simulated" in the UI** so no one mistakes it for live data. The app runs and demonstrates the signature touch with no external account.
- **Going live:** provision a **FlightAware AeroAPI** key (fastest to start) or a **Cirium Flight Status API** key (deeper international coverage). Set `FLIGHT_PROVIDER=flightaware` (or

(`circium`) + `FLIGHT_API_KEY=<key>`, restart. The provider sits behind an adapter interface (`src/flight-sim.js` is the seam), so the swap is config-only.

- **Fallback (built-in, keep it):** a concierge's manual override writes `actual_arrival` with `source: manual` and propagates identically; if the API is down or a flight isn't found, the app degrades to `scheduled_arrival` plus an "awaiting live status" flag on the ops exceptions panel. No pickup time silently goes wrong.
- **Polling** runs in-process in this release (a timer inside the Node process). At the scale where you split the poller to its own worker, see §9.

8. Backups & restore drill (the SQLite file IS the database)

All state lives in one file (`DATABASE_FILE`, default `data/spiti.db`, WAL mode, so `-wal` / `-shm` sidecars exist at runtime). Back it up as a file, correctly.

- **Consistent snapshot** (do NOT just `cp` a live WAL database):

```
sqlite3 data/spiti.db ".backup '/backups/spiti-$(date +%F-%H%M).db'"
```

This produces one consistent file while the app runs.

- **Schedule:** a cron job on the cadence the client needs (illustrative: hourly during an active event window, daily otherwise). Ship snapshots **off-box** (object storage / another host), encrypted — the file contains guest PII (passport/PNR/DOB are encrypted inside it, but treat the whole file as sensitive).
- **Permissions:** `data/` dir `700`, the DB file `600`, owned by the deploy user. It lives outside `public/`, so it is never web-served.
- **Restore drill (rehearse it):** stop the process; replace `data/spiti.db` with a snapshot; remove stale `-wal` / `-shm`; start the process; sign in; confirm data. Run this drill once before go-live. A backup you have never restored is a hope, not a backup.

```
sudo systemctl stop spiti
cp /backups/spiti-<stamp>.db data/spiti.db
rm -f data/spiti.db-wal data/spiti.db-shm
sudo systemctl start spiti
# then sign in and confirm the data is present
```

8.5 Migrating your data from the old platform (curated catalog vs. live guests)

There are **two kinds** of data moving off the old Spiti platform, and they travel by **two different paths on purpose** — because one is curated inventory the studio can seed for you, and the other is sensitive guest PII that must never leave your control.

What the studio already seeded for you (the curated world)

The delivered app ships with your **real curated inventory already in it**: the venues, restaurants, and hotels from your old-platform export, mapped into the Property catalog, plus your real brand and zone reference data (Louis Vuitton and the zones). This was extracted from your old-platform database dump by `src/real-inventory.js` (a strict, auditable table allowlist — see below) and loaded by `src/seed-real-inventory.js` at seed time. A fresh stand-up of the app comes up with the catalog populated; the property filters (category / country / city / neighborhood) work on real data from the first boot.

***The hard privacy line — what did NOT come across.** The extractor reads ONLY four tables from your old dump: `experience` (→ Property catalog), `company` (→ Brand), `zone` (→ Zone), and `country`. It never reads `guest`, `customer`, `person`, `booking`, or `employee`. **No guest record, no HNW client name, no passport, no PII, and no booking tied to a real person is in the delivered package.** Any curated row's link back to a guest or a booking is dropped — the property is seeded, never the person. A second scan (`scanForPII`) runs over the curated rows before insert and drops anything PII-shaped. This split is deliberate and client-confirmed: the studio seeds the curated world; **you** migrate the live guests, on your own server, by the path below.*

To re-run the curated seed against a fresh copy of the dump on your own box: point `SPITI_INVENTORY_DUMP=/path/to/dump.sql` at your copy of the old-platform database export you provided, then run `npm run reset`. If the dump is absent (a stripped production tree), the app still stands up — it simply comes up on the built-in demo catalog with no error.

What YOU migrate — the live guest records (the sensitive path, your hands only)

Guest records never travel through the studio. You import them yourself, on your own running server, through the app's **scoped Excel/CSV import** — the same feature your store managers and OC admins use day to day. This keeps every passport, DOB, and booking under your control end to end.

The steps, exactly:

1. **Sign in on your own server** as OC Admin (full-event import) or as a Store Manager (that store's guests only). The import is RBAC-scoped: a store manager's rows land in **their store only**, and a row that names another store is **rejected**, never reassigned.
2. From the guest list, open **Import from a spreadsheet** (`/import/guests`).
3. **Prepare your rows** as CSV, one guest per line, columns in this order: `display_name`, `vip_tier` (1–3), `justification`. (Paste directly from Excel — each spreadsheet row is one line.) A larger nomination template also accepts `legal_name`, `store`, `purchase_summary`, `languages`, `email`, `phone` per the data model; the three-column form is the quickest start.
4. Press **Validate & import**. Every row is validated **before any write** — required fields present, VIP tier in range. Invalid rows come back as a numbered error report; valid rows commit in one transaction. Nothing invalid is guessed.
5. Confirm the committed count and review any rejected rows. The import writes one `import_jobs` row (the retained receipt) and an `audit_entries` row (who imported what, when, and into which scope).

***Passport, DOB, PNR, and security flags are NOT import columns** for any brand-side or vendor caller. A spreadsheet round-trip **cannot** carry them — those are OC-only fields, entered and encrypted at rest inside the app (§5, §11), never moved through a CSV. This is by design: the sensitive fields are exactly the ones that must not sit in an emailed spreadsheet, so the import surface does not accept them.*

The split in one line: the studio seeds the curated catalog (no PII); **you** import the live guests on your own server (PII stays yours, validated and audited).

9. Scaling for multiple clients

The recipient runs events for several maisons; multi-tenancy is already in the data model — every row is under a `brand_id`, and the RBAC layer scopes every query. Options, in order of reach:

- **One instance, many brands (default, recommended for now).** The app is already multi-tenant: one process, one SQLite file, all brands inside it, walled by `brand_id` + the RBAC layer. Simplest and correct until load or data residency forces more. Scale the box vertically first.
- **Instance-per-client isolation** (if a maison contractually requires its data on its own box / region): one Spiti process + one SQLite file per client, each behind its own subdomain, each its

own `.env`. Clean blast-radius and residency story, more boxes to run. The codebase does not change; you deploy it N times.

- **The Postgres migration (the documented scale route).** SQLite's single-writer model is the ceiling. When you need concurrent write volume, multiple app instances behind a load balancer, or a managed-HA database, migrate the data layer to **PostgreSQL**. `src/db.js` is the single seam; `DATABASE_FILE` becomes a `DATABASE_URL`. Postgres unlocks: multiple stateless app instances, Row-Level Security as a second tenancy wall beneath the app query layer, managed backups/PITR, and read replicas. Move the flight-poller to its own worker at the same time so app instances stay stateless.
- **Sizing as clients grow (illustrative):** each active event adds ops users and flight-poll jobs, not heavy traffic. Watch RAM and the poller's API-call budget. A single 4–8 GB box carries several concurrent events; the trigger to move to Postgres is write concurrency and the need for >1 app instance, not page volume.

10. Logging & monitoring

- **App logs:** the process logs to stdout/stderr (a calm error handler — never a raw stack to the client). Capture via the systemd journal or pm2 logs; ship to the client's aggregator if they run one. Structured request logging (morgan/pino) is a noted enhancement, not shipped in this release.
- **Uptime/health:** monitor `GET /healthz` externally; alert on non-200 or process-down.
- **Error tracking:** wiring Sentry (or the client's equivalent) into the Express error handler is a small recommended addition — a hook point, not a dependency.
- **Audit vs. logs — do not conflate.** The `audit_entries` table is the immutable business audit trail (who saw/changed what), retained per compliance window and read in-app by OC Admin. Server logs are operational. Back up the DB (which holds the audit trail) per §8; keep operational logs per the client's ops policy.

11. Security & compliance posture at deploy

- TLS everywhere (§4); the Node app bound to localhost behind the proxy.
- `SESSION_SECRET` strong and rotated at handover; all provider keys rotated at handover (§5). Mind the encryption-at-rest note before rotating `SESSION_SECRET` independently.

- Passport/PNR/DOB encrypted at rest inside the DB. The DB file is PII; treat backups accordingly (§8).
- GDPR/CCPA: consent captured on the guest profile, right-to-erasure implemented, data residency addressable via instance-per-region (§9) or region-pinned Postgres.
- Keep the OS and Node patched; enable unattended security upgrades on the VPS.

12. Verifying the deploy

```
curl -fsS https://spiti.octravel.com/healthz # → ok
```

Then open the site, use the role-picker to sign in as each role, and confirm the signature: sign in as the Concierge, open a guest, Flights tab, "Delay 90 min," and check the Transport / Hotel / itinerary times re-flowed. Run the adversarial RBAC suite against a scratch DB to confirm the permission walls hold:

```
npm run test:rbac
```

It seeds a throwaway DB (`data/test-rbac.db`) and asserts every wall — including the Chapter Two walls (Client Director reaches zero guest PII, Zone Manager can't cross zones or reach logistics, the restaurant portal is type-keyed and window-gated, the message lanes hold, the guest wall is absolute). A green run is the definition of "the need-to-know model works."