Operational Rollback Guide

This service now supports an automatic SQLite fallback. Use this guide when primary Postgres connectivity fails or a deployment must be rolled back quickly.

Scenario 1: Primary DB Outage

1. Hit `/api/v1/healthz`.
2. If `ok=true` and `fallback=true`, service already operates on SQLite. Traffic continues (BUT data divergence risk—writes are not replicated back). Decide if read-only mode is acceptable.
3. Restore Postgres, then restart the application process to attempt reconnect (current code does not live-switch back). Confirm `fallback=false` afterward.

Scenario 2: Bad Migration (Future Alembic)

1. Stop deploy.
2. Restart previous container/image pointing at same Postgres.
3. If schema corrupted, set maintenance page, provision fresh DB from last backup, update `DATABASE_URL`.
4. Validate with `/api/v1/healthz`.

Scenario 3: Local Dev Data Reset

1. Stop server.
2. Remove `dev.db` and `test.db`.
3. Restart server; seed will re-create baseline chart of accounts.

Scenario 4: Forced Fallback Test

1. Export an invalid `DATABASE_URL`:

```
export DATABASE_URL=postgresql+asyncpg://bad:bad@localhost:5432/none
```

2. Start server; verify `/api/v1/healthz` returns `fallback=true`.

Data Divergence Notice

Operating in fallback (SQLite) while expecting Postgres means data written is isolated. Do NOT keep long-running fallback sessions in production; treat as emergency continuity only.

Recovery Checklist

- [ ] Postgres reachable
- [ ] `/api/v1/healthz` shows `fallback=false`
- [ ] Core endpoints functional (run `scripts/smoke_tests.sh`)

- [ ] Recent critical writes re-applied if fallback window occurred
- [ ] `/readyz` returns `status=ready` and `seed=true`
- [ ] `/metrics` latency and error counts stable after recovery
- [ ] No sustained 429 responses (rate limit) in logs
- [ ] Request IDs trace continuity across error logs

Future Improvements

- Automatic promotion back to primary without restart.
- Write-ahead dual logging for temporary fallback windows.
- Metrics export for fallback entry/exit events.
- Persistent metrics backend & dashboard (e.g., Prometheus + Grafana).
- Distributed rate limiting (Redis-based) instead of in-memory.
- Structured security event logging (auth failures, 429 bursts).

Observability & Rate Limiting Notes

- `/metrics` provides in-memory counters: `requests_total`, `requests_errors`, `avg_latency_ms` (resets on process restart).
- Rate limiting middleware returns `429` with JSON `{detail, retry_after_seconds}`; currently per-process and IP keyed.
- Use `X-Request-ID` from responses to correlate log events during incident analysis.

Readiness vs Liveness

- `/api/v1/healthz` (liveness) only reports DB connectivity/fallback.
- `/readyz` (readiness) ensures base seed (chart of accounts) exists plus DB ping; fail readiness to pull instance out of rotation.