Invarium HP Portal - Technical Manual

Invarium Financial Services — Hire Purchase Portal

Technical Manual

For staff-facing workflow documentation, see `USER_MANUAL.md` in this same folder. This manual is for whoever operates, maintains, or extends the system.

---

1. Architecture overview

```
Browser (HTTPS)
   |
   v
Caddy  (reverse proxy, auto Let's Encrypt TLS, security headers, on port 80/443)
   |
   v
waitress  (production WSGI server, 8 threads, bound to 127.0.0.1:8030 only)
   |
   v
Flask app  (routes/*.py blueprints, static/js/* served as plain files, no build step)
   |
   v
PostgreSQL  (primary datastore; SQLite is a dev/test-only fallback)
```

- **Frontend**: no framework. Server-rendered HTML templates + vanilla JS, loaded as ordered `<script>` tags (`static/js/00-core.js` through `99-enhance.js`). No bundler, no build step — edit and refresh.
- **Backend**: Python + Flask, organized as blueprints under `routes/`. Shared services (`core.py`, `security.py`, `db.py`, `storage.py`) have no circular imports by design.
- **PDF generation**: `reportlab` (contracts, offer letters, recovery notices, statements). **Excel**: `openpyxl`.
- **Process management**: both the app (`HPWizards` service) and Caddy run as always-on Windows Services via **NSSM** — see `PRODUCTION_DEPLOY.md` for the full setup checklist and `SERVICE_CONTROL.md` for day-to-day service operations.
- **Single instance, single node by design** — the app runs its own in-process background scheduler (§5). Running two instances would double-send every customer notification; never do this.

2. Data model — two storage layers

The app deliberately uses two different storage mechanisms:

2.1 Generic KV store (`kv` table)

Most configuration and semi-structured data lives as JSON blobs under string keys, read/written via `storage.py`'s `kv_get`/`kv_set`. Examples: `ifs:settings`, `ifs:company`, `ifs:quotes`, `ifs:apprlevels`, `ifs:catalogue`, `ifs:suppliers`. Some keys get "promoted" to dedicated Postgres tables for query performance (`storage._cfg_promoted`) — this is transparent to callers.

Three storage shapes recur throughout the code (see `storage.py`):

- `_KV_LISTPK` — a list of objects keyed by a field (e.g. `ifs:users` keyed by `u`). **Every write replaces the whole list** — this is why `security.merge_user_secrets()` exists (§3) to stop a partial client-side write from silently deleting users.
- `_KV_LISTORD` — an order-preserving list of scalars/objects, no dedup key.
- `_KV_SINGLE` — a single JSON blob (settings, company profile, etc.).

2.2 Relational tables (Postgres/SQLite, via `db.py`)

Transactional/fact data: `deals`, `receipts`, `outbox`, `audit`, `customers`, `schedule_lines`, `approvals`, `attachments`, `twofa`, `portal_accounts`, `portal_otp`, `penalty_journal`(`_lines`), and more. Schema lives in `db.py::init_db()`, which runs on **every app startup** and is fully idempotent — `CREATE TABLE IF NOT EXISTS` plus a series of `_safe_alter()` calls that add any column a newer code version needs, silently skipping ones that already exist. This means:

- A fresh install and an upgraded existing install converge to the same schema automatically.
- **A column that "should" exist but throws `UndefinedColumn` in production almost always means the running process hasn't been restarted since the code that added it was deployed** — restarting re-runs `init_db()` and self-heals it. (Real incident, 2026-07-30: the `audit` table's `level`/`category`/`extra` columns were missing in production for exactly this reason; fixed with a direct `ALTER TABLE` plus a restart.)
- `_safe_alter()` swallows *all* exceptions on each individual ALTER, including permission errors — so if the app's DB role doesn't **own** a table, a missing-column migration can fail completely silently. Ownership must match the connecting role (see §3.3).

3. Security model

3.1 Authentication

Staff sign in via `/api/login` (`routes/auth.py`) — username + password, verified server-side (`security.verify_legacy`), with optional TOTP (`twofa` table) for accounts that have it enabled. Sessions are Flask's signed-cookie sessions (`itsdangerous`), gated by:

- `[security] secret_key` in `config.ini` (or the `HP_SECRET_KEY` env var, which wins if both are set) — **without this set, a new random key is generated every process start, invisibly logging out every staff member on every restart/redeploy.** Always set this in production.
- `[security] cookie_secure = 1` — required once Caddy fronts the app with HTTPS; the browser won't send the cookie back over plain HTTP with this set, so don't flip it off except for local testing.
- `[security] enforce = 1` — the default-deny staff auth gate. Ships as `0`; must be flipped to `1` before go-live. Public customer endpoints (quote, apply, the customer portal) are exempt by design regardless of this setting.

3.2 Authorization

Role-based, checked client-side via `can(perm)` (`static/js/20-users.js`) against a `PERM` map (role → allowed permission strings), with per-user overrides (`USER.rights`) settable in Settings → Users & Roles. The `admin` username always passes every check. Permission strings in use: `set`, `usr`, `rcp`, `rep`, `appr`, `rec`, `lic`, `insedit`, `vpo`, `invpost`, `bo` — see `USER_MANUAL.md` §1 for what each gates. **This is a client-side convenience layer, not the security boundary** — sensitive routes also check role/session server-side (`security.py`, `routes/admin.py`).

3.3 Database role

Production should connect as a dedicated least-privilege role (`hpuser`), not `postgres` — see `scripts/setup_postgres_hpuser.sql` for the provisioning script (creates the role, transfers ownership of every table/sequence/view/function, revokes `PUBLIC` access, and grants only the DML the app actually performs). After running it, update `config.ini`'s `[database] user`/`password` to match and restart the service. If the app still connects as `postgres` (a superuser), everything keeps working regardless of table ownership — Postgres superusers bypass ownership/permission checks entirely — but you lose the least-privilege benefit this script exists for.

3.4 Customer portal (separate auth system)

The customer-facing portal (`/portal`, `routes/portal.py`) is **not** protected by staff login at all — it's a fully separate OTP-based flow:

• First-time enrollment: customer submits their National ID only; if a matching customer record exists, an OTP is sent to the **contact already on file** (never to anything the user types), which is the actual security control.

• Returning login: same OTP-to-known-contact flow against the resulting `portal_accounts` record.

• OTP codes expire after 5 minutes, allow 3 verify attempts, and are capped at 3 resends per rolling hour (`portal_security.py`). All portal endpoints are separately rate-limited by IP+identifier.

• The portal is read-only for the customer: deal status, statement, and document downloads for their own deal(s) only — no apply/edit capability.

4. Logging

Three structured JSON logging categories (`applog.py`), each writing daily-named files under `logs/`:

Category	File	What it captures
`application`	`application_YYYY-MM-DD.jsonl`	Internal errors/warnings/business events. Every unhandled exception on any `/api/*` route is logged here automatically (`app.py`'s global `@app.errorhandler(Exception)`), with a full traceback — check this first for any "server error" a user reports.
`access`	`access_YYYY-MM-DD.jsonl`	One line per HTTP request: client IP, method, path, status, timing, user-agent (`access_log.py`).
`security`	`security_YYYY-MM-DD.jsonl`	Mirrors every `core.audit()` call (logins, role changes, privilege escalations). The SQL `audit` table remains the durable source of truth for this category; the JSON file is a convenient secondary copy for grepping/future SIEM shipping.

• **Levels**: standard ERROR/WARN/INFO/DEBUG/TRACE, each a *threshold* per category (not independent toggles) — set to WARN and you get WARN+ERROR; set to TRACE and you get everything. TRACE is a custom level (5, below stdlib DEBUG=10) registered in `applog.py`.

• **Configurable live, no restart**: Settings → System → Logging has one dropdown per category; changes apply immediately via `apply_settings_levels()`, called on save.

• **Async by design**: each category's logger feeds a `QueueHandler` → background `QueueListener` thread, so a slow/full disk can never block a request thread.

• **Files never overwrite or cap** — unlike the older `ifs:loginlog` KV blob (still used by the legacy Activity Log PDF export in Back Office/Users), which keeps only the most recent 100 entries and silently drops anything older. The JSON logs have no such limit; prune them manually if disk space ever becomes a concern.

• **Gotcha**: keep any `.ps1`/log-related script **plain ASCII** — Windows PowerShell 5.1 can misread a non-ASCII character (e.g. an em-dash) in a script saved without a UTF-8 BOM, corrupting a string literal and breaking the parser with a confusing "string is missing the terminator" error. Real incident, 2026-07-30 (`scripts/run_backup.ps1`).

5. Scheduled/background work

A single background thread (`start_dunning_scheduler()`, started once at app boot in `app.py`) ticks every 60 seconds and does two independent things:

1. **Daily dunning cycle** — fires once per calendar day at the configured time (Settings, default 08:00): credit allocation → tracking-subscription invoices → renewal reminders → renewal invoices → the reminders engine → end-of-day → penalty accrual on overdue accounts → vehicle-service follow-up → recovery phase-date

stamping. Each step is independently wrapped so one failure doesn't block the rest.

2. **Outbox auto-flush** — independent of the daily cycle; if enabled (Settings → Notifications) and the configured interval has elapsed, retries any `QUEUED` email/WhatsApp messages. This is deliberately separate from the manual "Send queued" button's behavior: the automatic timer never revives a message that's permanently `FAILED` after 3 attempts, while a manual click does — this split was added specifically to stop a genuinely broken SMTP config from retrying forever (real incident: a missing "From" address).

Both operate on their own DB connections outside Flask's request/response cycle, so the thread also handles its own Postgres connection-pool cleanup each tick.

6. Backups & restore

Nightly Postgres backup via a Windows Scheduled Task calling `scripts/run_backup.ps1`, which wraps `scripts/backup_postgres.py`:

- Output: `C:\HP_Backups\hpsystem_YYYYMMDD_HHMMSS.dump` (timestamped, never overwritten), pruned after 14 days (`--keep-days`).
- Logs: `C:\HP_Backups\BackupLogs\backup_YYYY-MM-DD.log`, one file per day, with start/finish timestamps and exit code.
- Task Scheduler's own run history is **disabled by default** on Windows — enable via Action → "Enable All Tasks History" if you want it visible there too; it's not required for the backup itself to work (check `Last Run Result` in the task's properties either way).

Restore (test into a throwaway database first, never straight into production without a drill):

```
pg_restore -h localhost -U hpuser -d hpsystem --clean --if-exists --no-owner "C:\HP_Backups\hpsystem_&lt;times
```

`--clean --if-exists` drops existing objects before recreating them (needed if the target isn't empty); `--no-owner` avoids ownership-mismatch errors if the connecting role differs from the dump's original owner.

7. Deployment / redeploy

Full clean-room setup checklist: `PRODUCTION_DEPLOY.md`. Don't duplicate it here — key points worth restating:

- `venv\`, `tools\`, `logs\`, and the `Caddyfile` are **preserved** across a redeploy; only the app source files change.
- Restart the `HPWizards` service after any redeploy — `init_db()`'s migrations only run at startup (§2.2).
- If the server ever reboots (not just an app redeploy), there's a known race where the app can start before Postgres has finished its own startup, causing a transient crash (NSSM auto-restarts and it self-heals within seconds, but for a clean fix, set an explicit Windows service dependency: `nssm set HPWizards DependOnService <postgres-service-name>`).

8. Safe operational scripts

Every destructive script under `scripts/` defaults to a **dry run** — it only prints what it would do. Real execution requires both `--yes` and `--confirm CLEAR-TRANSACTIONS` (or the script's equivalent phrase). None of these are wired into any route or scheduled job — they are deliberately terminal-only, run-by-hand tools:

- `clear_transactions_keep_users.py` / `clear_transactions_keep_admins.py` — wipe deals/receipts/audit/etc. for a clean production launch, keeping either every user or admins-only.
- `backup_postgres.py` — see §6.
- `setup_postgres_hpuser.sql` — see §3.3.
- `seed_suzuki_catalogue.py` / `seed_preowned_catalogue.py` — populate the shared vehicle catalogue.

There is deliberately **no live-reachable "wipe everything" endpoint** in the running app — one existed (`/api/reset`) and was removed entirely after a safety audit; anything destructive now only exists as one of the

scripts above.

9. Known operational gotchas (real incidents worth remembering)

- **Blocking outbound calls in the request path can starve waitress's thread pool.** Already fixed once (a missing SMTP timeout) — any new outbound integration must set an explicit timeout and, ideally, run off the request thread.
- `_safe_alter()`**'s silent-failure design is intentional but has a blind spot**: it can't distinguish "column already exists" from "permission denied" — both are swallowed identically. If a migration seems to not be taking effect, check table ownership (§3.3) before assuming the migration code itself is wrong.
- **PowerShell scripts must stay plain ASCII** (§4).
- `schtasks /TR` **does not go through a shell** — redirection operators like `>>`/`2>&1` in a raw `/TR` string are passed as literal argv to the target program, not interpreted. Wrap in `cmd.exe /c "..."` or (cleaner) put the logic in a wrapper script and call that.