Running the Engine as a Windows Service

MessageFoundry runs as a long-lived background service via NSSM (the "Non-Sucking Service Manager"). NSSM wraps the `messagefoundry serve` command: it starts the engine on boot, restarts it on crash, and captures its output to rotating log files. Stopping the service sends Ctrl+C so the engine drains its connections cleanly (the ASGI lifespan stops the engine in an orderly way).

This page covers the single-machine setup with the engine bound to loopback. If you expose a listener beyond `127.0.0.1`, front it with TLS and an authenticated interface — MessageFoundry supports mTLS, OAuth 2.0 client-credentials, and SMART-on-FHIR Backend Services for interface authentication. See the architecture overview for networked-deployment guidance.

Prerequisites

1. **A Python virtual environment with the package installed.** From the repo root: `powershell python -m venv .venv .venv\Scripts\python.exe -m pip install -e .` This puts `messagefoundry.exe` in `.venv\Scripts\` — the service points at it. For a **reproducible, pinned** deployment, install the locked, hash-verified dependency set first, then the package itself: `powershell .venv\Scripts\python.exe -m pip install --require-hashes -r requirements.lock .venv\Scripts\python.exe -m pip install -e . --no-deps` `requirements.lock` is the SHA-256-pinned dependency export, kept in sync and audited in CI.
2. **NSSM** — provisioned automatically. If `nssm.exe` isn't on `PATH` (or passed via `-NssmPath`), `install-service.ps1` downloads the pinned, SHA-256-verified release into `<DataDir>\bin\nssm.exe` and uses it. No manual install needed. (You can still pre-install it — `choco install nssm` or a download from https://nssm.cc — and it'll be used if found.)
3. **An elevated PowerShell** (Run as Administrator) — required to register a service. The admin console's Engine Status page can do this for you via **Install service…** (UAC prompt).

Install

```
# from repo root, elevated PowerShell
.\scripts\service\install-service.ps1 -Environment prod
```

`-Environment` is **required**: it selects which `environments/<name>.toml` value file the engine resolves and the instance's PHI posture. `serve` refuses to start without it (no silent default), so the install script refuses too — pass `dev`, `staging`, `prod`, or a custom name. The console's **Install service…** button prompts for it.

Defaults:

Setting	Default
Service name	`MessageFoundry`
Engine exe	`<repo>\.venv\Scripts\messagefoundry.exe`
Config dir	`<repo>\samples\config`
Active environment	*(required — `-Environment`)*
Data dir	`C:\ProgramData\MessageFoundry`
Message store	`<DataDir>\messagefoundry.db`
Logs	`<DataDir>\logs\service.out.log`, `service.err.log`
Bind	`127.0.0.1:8765`
Log level	`INFO`

Override any of them, e.g.:

```
.\scripts\service\install-service.ps1 -Environment prod -Port 9000 -LogLevel DEBUG `
    -Config D:\hl7\config -DataDir D:\MessageFoundry
```

The install script is idempotent — re-running it reconfigures the existing service.

Update to a new build (restart vs reinstall)

The service runs `<repo>\.venv\Scripts\messagefoundry.exe`. With the documented **editable** install (`pip install -e .`), that exe imports straight from the repo source — so a running service keeps the code it loaded **at process start**. To pick up new code (a pull, a branch switch, a merge), just **restart** it (elevated):

```
& C:\ProgramData\MessageFoundry\bin\nssm.exe restart MessageFoundry
curl http://127.0.0.1:8765/health
```

Because the install is editable, a restart runs **whatever branch is checked out** in the repo.

Reinstall instead when paths or flags change (port, config dir, data dir) or the service definition drifted — the install script is idempotent (it stops and reconfigures in place):

```
.\scripts\service\install-service.ps1 -Environment prod   # elevated; re-points the exe + AppParame
& C:\ProgramData\MessageFoundry\bin\nssm.exe start MessageFoundry
```

If the package was installed **non-editable** (a plain `pip install .`), the venv holds a snapshot of the old code — run `.venv\Scripts\python.exe -m pip install -e .` first, then restart.

Start / stop / status

```
nssm start  MessageFoundry
nssm status MessageFoundry
nssm stop   MessageFoundry      # Ctrl+C -> graceful connection shutdown (up to 15s)
nssm restart MessageFoundry
```

If `nssm` isn't on `PATH`, it's the auto-downloaded copy at `<DataDir>\bin\nssm.exe` (e.g. `C:\ProgramData\MessageFoundry\bin\nssm.exe`). You can also use the built-in `sc.exe` / Services.msc once installed.

Security hardening (recommended)

Run as a least-privilege account

By default the service runs as **LocalSystem** — the most privileged local account. The engine needs only **read** on the config directory and **read/write** on the data directory, so LocalSystem grants far more than required and widens the blast radius of any compromise (for example, a config module is executed in-process — see *Lock down the config directory* below).

Install under a dedicated low-privilege account instead. A **virtual service account** needs no password and is the simplest option:

```
.\scripts\service\install-service.ps1 -Environment prod -ServiceAccount "NT SERVICE\MessageFoundry"
```

When `-ServiceAccount` is supplied the install script **auto-grants the account exactly what it needs**: read+execute on the config directory and read/write on the data directory (the manual `icacls` lines below are only needed if you point the engine at directories outside those). Omitting `-ServiceAccount` still works but prints a loud warning that the service is running as LocalSystem.

```
# only if config/data live outside the script-managed paths:
icacls "D:\hl7\config"                /grant "NT SERVICE\MessageFoundry:(OI)(CI)RX"
icacls "C:\ProgramData\MessageFoundry" /grant "NT SERVICE\MessageFoundry:(OI)(CI)M"
```

A domain **gMSA** or a dedicated local user works the same way (pass `-ServiceAccountPassword` for a password-based account — it's taken as a `SecureString`). The store file itself is further restricted to its owner at runtime; account choice governs who that owner is.

Protect the store encryption key at rest

PHI columns are AES-256-GCM-encrypted at rest when a key is configured (see the PHI handling guide). The key is a base64 32-byte secret. Two ways to supply it:

- **Environment (cross-platform default).** Set `MEFOR_STORE_ENCRYPTION_KEY` in the service's environment (`nssm set MessageFoundry AppEnvironmentExtra MEFOR_STORE_ENCRYPTION_KEY=...`). Simple, but the plaintext key sits in the service environment block, readable by any local administrator.
- **DPAPI-protected key file (Windows).** Keep the key in a file that Windows DPAPI binds to *this machine*, so a copied file is useless elsewhere and no plaintext key is in the environment:

`powershell # mint + protect a fresh key (machine scope, so the service account can read it at startup): messagefoundry protect-key --generate --out "C:\ProgramData\MessageFoundry\store.key.dpapi" # -> prints the base64 key ONCE to stderr; back it up offline (the file is machine-bound and # unrecoverable if the host is lost), then point the engine at it: toml [store] encryption_key_file = "C:/ProgramData/MessageFoundry/store.key.dpapi"`
Then **unset** `MEFOR_STORE_ENCRYPTION_KEY` (the env key takes precedence when both are set). The service account decrypts the file at startup; a missing, foreign, or unreadable file makes `serve` fail closed rather than store PHI unencrypted. `protect-key` already restricts the file to its owner; keep it under the data dir so the data-dir ACL covers it too. To rotate, `protect-key` a new key to the file and run `messagefoundry rotate-key` with the prior key in `MEFOR_STORE_ENCRYPTION_KEYS_RETIRED` (see the PHI handling guide).

> **External vault / managed identity.** DPAPI is the built-in on-box option. To source the key (or SQL/AD credentials) from an external secrets manager — Windows Credential Manager, HashiCorp Vault, Azure Key Vault via a **managed identity**, or an AD **gMSA** for SQL/LDAP — fetch the secret in your service-start wrapper and export it as the corresponding `MEFOR_*` variable, or place the DPAPI key file via your provisioning tool. The engine reads only the environment and `encryption_key_file`; it does not call a vault directly.

Lock down the config directory

`messagefoundry serve --config <dir>` and `POST /config/reload` **execute the Python** in the config directory in-process, with the service account's privileges. The directory is therefore a trust boundary: anyone who can write a `.py` file there can run code as the service.

- Restrict the config directory's ACL so only administrators / the service account can write it: `powershell icacls "D:\hl7\config" /inheritance:r /grant "Administrators:(OI)(CI)F" "NT SERVICE\MessageFoundry:(OI)(CI)R"`
- On POSIX hosts the loader additionally **refuses** to load from a group/world-writable directory or module file.
- `/config/reload` only loads from the startup `--config` directory and any directories listed in `[api].config_reload_roots` (see the configuration reference); an arbitrary path is rejected. Keep those roots admin-owned too.

Verify it's running

```
curl http://127.0.0.1:8765/health       # -> {"status":"ok", ...}
```

Send a test message and confirm it flows through:

```
.venv\Scripts\python.exe samples\send_mllp.py samples\messages\adt_a01.hl7
```

Then check the log:

```
Get-Content C:\ProgramData\MessageFoundry\logs\service.out.log -Tail 20 -Wait
```

You should see uvicorn's startup banner and `wiring started: N inbound, N outbound connection(s)`. On stop you should see `wiring stopped` and `engine stopping` — confirmation of a clean shutdown. (A live config swap logs `wiring reloaded: ...`.)

Logs

The engine logs to stdout/stderr with a stdlib `logging` setup (one timestamped UTC stream), with a CR/LF log-injection filter and a PHI-redaction chokepoint on the exception path (see the PHI handling guide). NSSM captures those streams to the files above and rotates them at ~10 MB. The engine also exposes operational telemetry on a `/metrics` endpoint for scraping into your own monitoring stack. As deployment guidance, **avoid raising the level to `DEBUG` in production**, since verbose output may include message content.

Restrict the log directory's ACL so the captured stdout/stderr (operational data, not message bodies) is readable only by administrators and the service account — NSSM's files would otherwise inherit a broadly-readable `ProgramData` ACL:

```
icacls "C:\ProgramData\MessageFoundry\logs" /inheritance:r `
  /grant "Administrators:(OI)(CI)F" "NT SERVICE\MessageFoundry:(OI)(CI)M"
```

Admin console (optional desktop shortcut)

This service is **headless**. Operators watch and run it from the **admin console** — a separate desktop app (not part of the service) that connects over the localhost API. Its Alerts and Dead-Letters views surface engine health and let operators triage failed messages. Give them a double-click icon instead of a command line:

```
pip install "messagefoundry[console]"            # into the engine venv
.\scripts\console\install-console-shortcut.ps1   # Desktop + Start-Menu icon (per-user; -AllUsers f
```

It launches the windowed `messagefoundry-console.exe`, connects to this service on `http://127.0.0.1:8765`, and prompts for sign-in. Local console accounts use built-in multi-factor authentication; enterprise users sign in through your identity provider via SSO federation, with MFA enforced there. See the install guide under "Launching the admin console".

Uninstall

```
.\scripts\service\uninstall-service.ps1
```

This stops and removes the service. The log files and message store under `DataDir` are left in place.

Troubleshooting

- **Service won't start / exits immediately.** Read `service.err.log`. The most common cause is a bad path baked into the service (relative paths resolve to the *system* directory for a service account); re-run the install script, which resolves all paths to absolute.
- **Port already in use (e.g. 2575).** The sample config's inbound connection binds MLLP port `2575`. If a stray `messagefoundry serve` (or a second copy of the service) is already running, the listener fails to bind. Make sure only one instance runs: `Get-Process messagefoundry,python | Format-Table Id,ProcessName,Path`.
- **`/health` doesn't respond.** Confirm the service is `SERVICE_RUNNING` (`nssm status MessageFoundry`) and that nothing else owns port `8765`.
- **Permissions on the data dir.** The service runs as `LocalSystem` by default, which can read/write `C:\ProgramData\MessageFoundry`. If you point `-DataDir` somewhere the service account can't write, startup fails — pick a writable location, or (recommended) install under a least-privilege `-ServiceAccount` and grant it read/write on the data dir (see *Security hardening* above).