FaceForge Core — Project Design Specification

```
Document Status: Design & Ideation
Scope: FaceForge Core
Last Updated: 2026-01-09
Version: v0.2.9
```

1. Source of Truth

This document is the authoritative design specification for **FaceForge Core**: what it must provide, what it intentionally avoids, and the interfaces that future work will build on.

FaceForge is designed to stay **stable and boring at the center**, while advanced functionality (recognition, training, integrations, rich visualizations, etc.) is delivered as optional plugins over the Core's public APIs and job system.

Note: This specification describes desired behavior and interfaces. Where implementation details are not specified (or could vary), those areas are marked as **TO-DO** while preserving constraints and intent.

2. Project Philosophy and Vision

2.1 What is FaceForge?

FaceForge is a **local-first, self-hosted Asset Management System (AMS)** specialized for **entities** (real and fictional, human and non-human) and **assets** related to those entities (facial maps, voice samples, attributes, relationships, etc.).

It is envisioned as a desktop-managed local services bundle:

- A lightweight **desktop shell** (user interface + desktop tray app + admin settings + application updates)
- A **local API server** (headless, API-first) with optional localnet access capabilities
- Managed local sidecars where appropriate (for example: a local S3-compatible object store)
- Optional out-of-process plugins for compute-heavy workloads and other integrations

External tools (for example, ComfyUI or StashApp) must be able to query FaceForge and retrieve assets through a stable API without depending on the UI.

A common target environment includes **very large local datasets**. FaceForge is expected to operate primarily on local storage while remaining compatible with remote object storage systems (like Amazon S3, Google Cloud Storage, MEGA S4, MinIO, etc.).

2.2 The "boring core" philosophy

The Core provides stable foundations:

- Database CRUD operations (create, read, update, and delete) for entity records and the metadata of their associated attachments

- Durable storage abstractions for assets (likely visual media, audio clips, Stable Diffusion LoRAs, etc.)
- A consistent jobs + logging model
- A plugin surface (API + worker/process + UI integration hooks)

The Core does **not** own compute-heavy intelligence such as face recognition, force-directed graph rendering, or LoRA training. Those are delivered as plugins. FaceForge Core is the asset storage and management **platform** those capabilities connect to.

2.3 Key Design Principles

1. **Keep the center stable** The Core stays small and dependable. New capabilities should plug in through stable interfaces rather than expanding Core scope.

2. **Transparent storage** Assets must not be hidden inside opaque container volumes. Users must be able to see and manage where their data lives (local disks, NAS, remote storage, etc.). The user can control the application's data storage broadly (running everything from a named folder path) or in a more modular way (e.g., separating database files from object storage).

3. **"Double-click" simplicity (cross-platform)** The system must be runnable by non-developer users via a single desktop launcher experience. **Container runtimes (Docker, Podman, etc.) are explicitly out of scope for end-user deployment.** All required components are managed directly by the desktop shell application.

4. **Integration-first**
   The Core is designed to serve downstream tools via a stable and well documented API. Documentation of the API is provided by OpenAPI/Swagger and kept in sync with the active implementation at all times.

3. What FaceForge Core Provides

The specific capabilities and features of FaceForge Core are categorized and defined as follows:

3.1 Central Metadata Engine

- Immutable **entity IDs** (SHA-256) assigned at creation time.
- Entity records with flexible fields, including:
  - standard descriptive information fields
  - arbitrary admin-defined custom fields
- A universal asset/attachment model that can represent **any file type** (images, audio, PDFs, archives, model artifacts, etc.).
- Durable representations of relationships between:
  - entities ↔ assets (attachments)
  - assets ↔ derived assets (e.g., thumbnail derived from an original image)
  - entities ↔ entities (relationships stored as metadata)

Note: The *visual relationship graph UI* is intentionally out of scope for the Core. The Core provides the data surface for relationship metadata; rendering and interaction will live in an official plugin (FFGraph).

3.2 Storage & Retrieval

- A storage abstraction that supports rule-based selection of storage backends per asset type/kind:
  - Local S3-compatible object storage (default: SeaweedFS)
  - Local filesystem-only provider (e.g., for small files)
  - Remote S3 endpoints (AWS S3, Wasabi, Backblaze, MEGA S4, etc.) configured by the user
- Stable, secure download URLs for assets (including large artifacts such as LoRAs).
- Streaming downloads with HTTP range support (resume-friendly).

> TO-DO: First-run storage UX (wizard choice vs defaults) and the initial set of supported remote endpoints.

3.3 Jobs & Logging

- Long-running operations are tracked as **jobs** in the database.
- Job state + append-only structured logs:
  - Viewable in the web UI
  - Retrievable via headless API requests
- Jobs can emit **progress updates** (percentage complete, current step, etc.).
- A consistent "job type" contract so plugins can contribute job types.

Design intent: Compute-heavy work happens out-of-process. The Core remains the job registry, API surface, and artifact store.

3.4 Plugin Foundations

Core is responsible for plugin discovery, registration, configuration, permissions, and versioning. Plugins must be able to:

- Add API routes (namespaced)
- Register job types (namespaced) and emit logs/progress updates
- Add new asset kinds / derived artifact types
- Add UI pages/panels and navigation entries (integrated into the Core UI shell)
- Subscribe to events (ingest completed, entity updated, asset created, etc.)

3.5 Web-Based UI for all Core Features

- The UI exposes *everything the Core can do* (no "backend-only" features are allowed).
- Plugin UIs integrate into the Core UI shell and navigation.

4. Planned Official Plugins

These named plugins are upcoming design targets that the Core must accommodate naturally. The capabilities of these various plugins give a glimpse of the Core's ultimate role and some potential intended use cases.

- **FFIdentify — face detection + identity matching**
  Detect faces, compute embeddings, match identities across media, and emit structured results back into Core.

- **FFGraph — relationship graphing**
  Force-directed graph visualization that turns Core relationship metadata into an interactive "entities

web."

- **FFLoRA — LoRA training**
  Produces per-entity LoRAs (SDXL, SD 1.5, SD 2.x). Outputs are stored as assets with clean provenance and derived-asset links.

- **FFSD — Stable Diffusion / ComfyUI integration**
  Custom nodes that can list, fetch, and resolve LoRAs and other artifacts directly from FaceForge with a stable API contract.

- **FFStash — StashApp integration**
  Bulk-friendly import and sync workflows (performers → entities, media scanning jobs, write-back of tags/notes where configured).

- **FFVoice — voice cloning**
  Manages voice samples as assets linked to entities and produces cloning artifacts per entity.

- **FFDialogue — conversational agent**
  Builds context-aware chat agents using entity metadata + attachments, with a dedicated UI surface.

- **FFMeta — media library enrichment**
  Scans libraries, extracts faces/audio, matches identities, and writes structured outputs (tags, notes, sidecar metadata files).

- **FFBackup — backup/export/restore**
  Scheduled backups of metadata + assets, selective exports (JSON + ZIP), and restore workflows.

5. Project Non-Goals & Constraints

- No paid SaaS should ever be required.
- Authentication secrets are never committed (appropriate `.gitignore` rules, environment variable support, etc.).
- Plugins must not silently exfiltrate data; permissions and explicit enablement are required.
- Avoid magical "it trains a LoRA." Training lives outside the Core and is explicit about tooling and invocation.
- Keep Core stable; breaking changes are rare and intentional.
- Avoid using any infrastructure that forces Electron-scale overhead without a hard requirement for it.

FaceForge **Core** does not directly:

- Perform face detection or recognition
- Generate embeddings
- Do identity matching
- Train or tune LoRAs
- Render a force-directed relationship graph
- Directly integrate with third-party apps (ComfyUI, StashApp, etc.)

The Core provides the platform surface those concerns plug into.

6. Core Architecture Baseline (Practical v1 Target)

This section describes a practical minimum baseline intended to be reliable and easy to run locally **without containerization**.

6.1 Runtime Layout (What Ships)

6.1.1 Desktop Shell Orchestrator

- **Tauri v2** desktop app (tray + updates + system popups)
- Responsible for:
  - Starting/stopping/monitoring managed components (Core API service, local object storage, plugin runners)
  - Autostart at login (optional)
  - Presenting status/logs and opening the web UI

6.1.2 Core Service (Python 3.11/3.12 + FastAPI + Uvicorn)

- Runs as a local HTTP API server:
  - `/v1/...` API
  - `/docs` and `/redoc` API playground
  - the web UI as static assets (one service, one port)

6.2 Database (SQLite)

- **SQLite** is the metadata database.
- The database lives under the user data directory (file-based, easy to relocate and back up).
- The Core uses a **relational spine** for durable relationships (entities/assets/jobs/relationships) plus **JSON fields** for flexible, admin-defined descriptors.

Why SQLite (design reasoning):

- **Embeddable engine**: no separate database service to install/manage; lifecycle is bound to the Core process.
- **Local-first reliability**: single-file persistence with excellent durability; easy backups and restores.
- **Flexible fields without schema churn**: admin-defined fields live in JSON; common fields can be indexed via generated columns.
- **Operational simplicity**: fewer moving parts, fewer failure modes, and a smoother "double-click" experience.

6.3 S3-Compatible Object Storage (SeaweedFS)

- Default: **SeaweedFS** providing a **local, S3-compatible endpoint** for asset storage.
- SeaweedFS points at a user-selected directory inside the FaceForge data directory (or another user-chosen path).
- Optional: remote S3 endpoint configured by the user (AWS/Wasabi/Backblaze/etc.) in later phases.

Why SeaweedFS (design reasoning):

- **S3 compatibility for external tools** without bundling MinIO (and its licensing/packaging concerns).
- **User transparency** over where bucket data lives on disk.
- **Lightweight local services model** appropriate for a self-hosted desktop-managed bundle.

6.4 Network and Security Defaults

- Core binds to **127.0.0.1** by default (not LAN-accessible unless the user opts in).
- Sensitive endpoints require a **per-install token** (stored in the config in the user data directory).
- The system exposes **one localhost port** by default (API + UI served together).

6.5 Key Constraints

- Runs locally on one machine.
- Supports GPU-first plugin workflows, but the Core runs without a GPU.
- Windows-friendly setup is mandatory, while still keeping the architecture cross-platform.

7. Data model (Core-owned, plugin-friendly)

This section defines the *conceptual* model. The physical schema is implementation work, but the invariants must hold.

7.1 Entities

Entities are the primary identity objects in the system (humans, characters, creatures, etc.).

- `entity_id`: SHA-256 hex string (64 chars), immutable
- `display_name`: string
- `aliases`: string[]
- `fields`: JSON object for admin-defined fields (validated by field definitions)
- `primary_thumbnail_asset_id`: asset reference (optional)
- `tags`: string[]
- `created_at`, `updated_at`: timestamps
- `plugin_data`: JSON object keyed by plugin id (namespaced)

7.2 Assets (Attachments)

Universal attachment model (files, models, documents, etc.).

- `asset_id`: SHA-256 hex string (64 chars), immutable
- `owner`: `{ "type": "entity" | "system" | "plugin", "id": "..." }`
- `linked_entity_ids`: `entity_id[]` (many-to-many)
- `kind`: string (examples: `image.original`, `image.thumbnail`, `model.lora`, `doc.pdf`, `audio.clip`)
- `mime_type`, `size_bytes`
- `content_hash`: sha256 of file bytes (dedupe/integrity)
- `storage`: provider + bucket + key
- `derived_from_asset_id`: optional provenance link
- `extracted_metadata`: JSON (Core-generated at ingest time)
- `user_metadata`: JSON (optional)
- `plugin_data`: JSON (namespaced)

7.3 Descriptors and Field Definitions

Two complementary concepts:

- **Field definitions**: admin-declared keys that control validation and UX (type, options, regex, etc.).
- **Descriptors**: facts/attributes that may be time-bound or historical (addresses, phone numbers, notes, external IDs, etc.).

Design intent:

- The system must support *new* descriptor keys being defined at any time.
- Historic records are not required to have those keys.
- Historic records may be updated later, without migrations or schema changes.

7.4 Jobs, Events, Relationships

- **Jobs** track long-running operations and their logs.
- **Events** optionally provide an append-only integration surface.
- **Relationships** store metadata relationships (visualization belongs outside Core).

8. Plugin Model (Contract-Level)

8.1 Plugin Types

A plugin may provide any combination of:

- API plugins (routes)
- Worker/process plugins (job types and an out-of-process runner)
- UI plugins (pages, panels, nav items, settings forms)
- Asset processors (derive thumbnails, transcode, generate model artifacts)
- Integrations with third party applications (ComfyUI, StashApp, etc.)

8.2 Plugin Manifest (Concept)

Each plugin ships with a manifest (example: `plugin.json`) containing:

- `id`, `name`, `version`
- `core_compat`: semver range
- `capabilities`: `["api", "runner", "ui", ...]`
- `config_schema`: JSON Schema (for UI-rendered settings)
- `permissions`: declared needs (read entities, write assets, create jobs, filesystem scopes, network, GPU access, etc.)
- `routes_prefix`: e.g. `/v1/plugins/<id>`
- `job_types`: list (namespaced)
- `entrypoints`:
  - `runner`: how to start the plugin process
  - `ui`: route(s) exposed in the Core UI shell

8.3 Execution Model (Out-of-Process First)

Phase 1 (MVP):

- plugins installed as folders inside a local install dir
- Core loads manifests and exposes plugin metadata via API

- Desktop shell starts/stops plugin runner processes as needed
- Plugins communicate with Core via **HTTP**:
  - create jobs
  - stream logs/progress
  - upload artifacts (assets) and link them to entities
- UI renders navigation based on manifest + enabled status

Phase 2 (later):

- install from URL/local `.whl`/`.zip`
- per-plugin virtualenvs (and potentially per-plugin Python versions)
- signed bundles + marketplace

> TO-DO: Bundle signing, distribution, sandboxing strategy, and permission enforcement depth need a separate design doc when in scope.

9. API Contract (Core + Plugin Friendly)

9.1 Versioning

- Core API version prefix: `/v1/...`
- Avoid breaking changes. Prefer adding fields rather than removing.

9.2 Core Endpoints (Minimum)

9.2.1 Entities

- `GET /v1/entities`
- `POST /v1/entities`
- `GET /v1/entities/{entity_id}`
- `PATCH /v1/entities/{entity_id}`
- `DELETE /v1/entities/{entity_id}` (soft delete recommended)

9.2.2 Descriptors

- `GET /v1/entities/{entity_id}/descriptors`
- `POST /v1/entities/{entity_id}/descriptors`
- `PATCH /v1/descriptors/{descriptor_id}`
- `DELETE /v1/descriptors/{descriptor_id}`

9.2.3 Field Definitions (Admin)

- `GET /v1/admin/field-defs`
- `POST /v1/admin/field-defs`
- `PATCH /v1/admin/field-defs/{field_key}`
- `DELETE /v1/admin/field-defs/{field_key}`

9.2.4 Assets / Attachments

- `POST /v1/assets/upload` (multipart; supports companion `_meta.json`)
- `POST /v1/assets/bulk-import` (directory import job; reads `*_meta.json` sidecars)

- `GET /v1/assets/{asset_id}`
- `GET /v1/assets/{asset_id}/download` (stream; must support HTTP range)
- `POST /v1/entities/{entity_id}/assets/{asset_id}/link`
- `POST /v1/entities/{entity_id}/assets/{asset_id}/unlink`

9.2.5 Relationships

- `GET /v1/relationships?entity_id=...`
- `POST /v1/relationships`
- `DELETE /v1/relationships/{relationship_id}`
- `GET /v1/relation-types?query=sp`

9.2.6 Jobs

- `POST /v1/jobs`
- `GET /v1/jobs/{job_id}`
- `GET /v1/jobs/{job_id}/log`
- `POST /v1/jobs/{job_id}/cancel`

9.2.7 Plugins

- `GET /v1/plugins`
- `POST /v1/plugins/{id}/enable`
- `POST /v1/plugins/{id}/disable`
- `GET /v1/plugins/{id}/config`
- `PUT /v1/plugins/{id}/config`

9.3 Integration-Driven Core Requirements

Stable asset listing & download

- `GET /v1/entities/{entity_id}/assets?kind=model.lora`
- `GET /v1/assets/{asset_id}/download` supports large files and resumable downloads (HTTP range).

Bulk operations

- `POST /v1/entities/bulk-upsert` (idempotent; external IDs supported)
- `POST /v1/jobs` must support large inputs via stored "job input assets" (avoid giant JSON payloads)

> TO-DO: The exact shape of external IDs and bulk-upsert conflict resolution rules require specification.

10. Storage & Path Configuration

All persistent data is stored in one or more user-controlled, user-defined directories.

A system-wide environment variable controls where the application lives. The primary location where the application binary (FaceForge.exe) and required static assets reside is designated by `FACEFORGE_HOME`:

- `$env:FACEFORGE_HOME = 'C:\FaceForge'` (Windows example)
- `FACEFORGE_HOME=/media/faceforge` (Linux example)

Inside that directory, FaceForge maintains several subdirectories.

- `${FACEFORGE_HOME}/db`: Metadata database (configurable)
  - The SQLite database file that lives here is named: `faceforge.db`
- `${FACEFORGE_HOME}/s3`: S3 Object Storage (configurable)
  - SeaweedFS volume data lives here
  - This can be bypassed by configuring a remote S3 endpoint in the application config
- `${FACEFORGE_HOME}/logs`: System Logs (configurable)
- `${FACEFORGE_HOME}/backups`: Backups (configurable)
- `${FACEFORGE_HOME}/run`: Application runtime dependencies (NOT configurable)
  - Temporary files, PID files, port files
  - Python venv (if applicable)
  - Additional required binaries, etc.
- `${FACEFORGE_HOME}/config`: Core configuration files (NOT configurable)
  - Core config files (YAML/JSON)
- `${FACEFORGE_HOME}/plugins`: Plugins (configurable)

The items marked "configurable" can be relocated by the user via the application settings UI or by editing the configuration files directly while the application is not running.

> TO-DO: Define the config file's internal structure, format (JSON suggested), and name.

11. Frontend Requirements (Core UI + plugin UI)

11.1 Core UI Coverage

The Core UI covers all Core features:

- manage entities and descriptor fields
- upload/link/download attachments
- browse/search/filter entities and assets
- view job status and logs
- manage plugins (enable/disable/config)

11.2 UI Delivery Model (Aligned with Desktop + Local Services)

- The Core service serves the web UI as static assets.
- Users can open it in a browser at `http://127.0.0.1:<port>` *or* view it inside the desktop shell window.

Minimum UX expectations:

- **Entities** page with list + gallery view
- **Entity detail** page with tabs:
  - Overview
  - Descriptors
  - Attachments
  - Relationships (metadata)
  - Plugin panels (rendered per plugin)

> TO-DO: SPA vs server-rendered UI is an implementation choice. The hard requirement is that the UI is served by the Core service and does not require NodeJS at runtime.

12. Packaging and "Double-Click to Run"

This section defines the desired end-user experience for running FaceForge Core locally and the specific design decisions intended to reduce friction in that process as much as possible.

12.1 Network Ports

- Core exposes two localhost ports which are configurable via the application config files:
  - Core port (API + UI) (default port: 43210)
  - SeaweedFS S3-compatible port (default port: 43211)
- Launcher supports auto-picking a free port and writing the chosen port to: `${FACEFORGE_HOME}/runtime/ports.json`

12.2 Desktop Launcher Experience (Cross-Platform)

A desktop launcher is a first-class deliverable.

Release UX requirements:

- ship a **FaceForge Desktop** app (Tauri) that behaves like a real app:
  - first-run wizard for choosing `FACEFORGE_HOME` and Core port
  - starts/monitors required components (Core API service, SeaweedFS services, plugin runners)
  - exposes only the UI/API port
  - minimizes to system tray instead of exiting when the window is closed
  - tray menu: **Open UI**, **Status**, **Logs**, **Stop**, **Restart**, **Exit**
  - on exit: prompt whether to stop services or leave them running

User-facing simplification goals:

- one installer per OS (or portable zip where appropriate)
- one desktop launcher app
- one user-chosen data directory
- no loose scripts

Documentation is written for non-engineers first and includes screenshots of first-run and tray controls.

12.3 Runtime Constraints

- NodeJS is not required at runtime.
  Node may be used at build time to produce frontend assets bundled into the Core service.
- The desktop shell must be lightweight (reserve CPU/RAM/VRAM for the likely AI workloads managed by FaceForge plugins).

12.4 Packaging Notes (Implementation Direction)

- Tauri packages the desktop app per-OS.
- Core Python service ships as a sidecar:
  - either a frozen binary (PyInstaller/Nuitka), or
  - an embedded Python runtime + venv

- SeaweedFS binaries live alongside the app (or are user-supplied) and are launched with explicit paths/config.

> TO-DO: Update strategy for the Core service, SeaweedFS, and plugin bundles.

13. Relevant Links (Informational References)

- FastAPI: https://fastapi.tiangolo.com/
- SeaweedFS: https://seaweedfs.com/
- SQLite: https://www.sqlite.org/
- StashApp: https://stashapp.cc/
- Tauri: https://tauri.app/
- Uvicorn: https://www.uvicorn.org/