HALEY STORAGE – FULL DETAILED SYSTEM EXPLANATION (PLAIN TEXT VERSION)

1. High-level purpose
   Haley Storage (DiskStorageService) is a complete document-storage subsystem designed for enterprise-grade on-premises storage. It manages both physical storage (Linux LVM recommended) and logical metadata through MariaDB. It supports structured storage hierarchies, versioning, streaming, folder management, and a hybrid model that can run either purely on filesystem or with database indexing.

---

2. Logical model
   The storage hierarchy is:
   Client → Module → Workspace → Directory → File → File Version.

Each level has:

- A logical identity (Name, DisplayName, GUID, CUID)
- A physical representation (folder path, except virtual workspaces)
- Optional indexing metadata stored in MariaDB

Client, Module, Workspace are “controlled” entities with system-managed IDs and folder layouts. Directory and File entries correspond to dynamic user-managed content inside a workspace.

---

3. Client
   A Client represents a top-level storage tenant (e.g., “bcde”).
   It stores:

- Name and DisplayName
- GUID and deterministic CUID
- Path suffix
- Encryption key, signing key, password hash
- A .client.dss.meta file inside the client folder containing serialized client info
  When registered, the service:
- Creates a physical base folder under BasePath
- Writes the meta file
- Registers client into dss_core DB (if indexer enabled)
- Auto-creates a default Module and default Workspace

---

4. Module
   Module sits under a Client, representing feature groups (e.g., “Files”).
   It stores:

- Its own GUID and CUID
- The module path suffix
- A .module.dss.meta file
  On registration:
- Physical folder created under Client folder
- Meta file written
- DB entry stored in dss_core
- A virtual default workspace automatically created

---

5. Workspace
   A Workspace is where actual files belong.
   It may be:
   (a) Physical workspace (actual folder)
   (b) Virtual workspace (no physical folder; everything must be DB-indexed)

Every workspace has:

- ContentControl Mode (None, Number, Guid)
- ContentParse Mode (Generate or Strict)
- Sharding rules (depth, split length, suffix for files)
- A .ws.dss.meta file
- Logical identity via GUID and CUID
  On registration:
- Physical folder created unless virtual
- DB entry inserted into dss_core
- Workspace configuration (control mode, parse mode) stored in DB

---

6. Directory
   A Directory represents a folder inside a workspace.
   Stored in the DB table “directory” inside the per-client DB.
   Fields include: workspace_id, parent_id, name, deleted bit.

Directory 0 or null parent represents root folder.
Directory paths are determined by combining workspace base path + directory chain.

---

7. File and file versions
   Files inside a workspace are represented by:

- document table → represents one file (like conceptual file)
- doc_version table → each upload is a version
- version_info table → physical file name, extension, saved path, size
- name_store table → stores base filename and extension
  This design allows multiple versions, metadata updates, and structured lookup.

---

8. Control modes
   Workspace determines how filenames are generated.
   None: user-controlled names, no system naming enforced.
   Number: system-generated numeric IDs, which are split into subfolders via sharding.
   Guid: system-generated GUID-based names, split by characters.

Examples:
Number mode: fileID 1458623 → shard: 01/45/86/23F
Guid mode: compact guid splitted 1 char per folder (depth 5 by default).

---

9. Sharding logic
   DiskStorageService uses split length and depth from DSSConfig:
   SplitLengthNumber, DepthNumber for numeric mode.
   SplitLengthHash, DepthHash for GUID mode.
   These determine the folder nesting, preventing filesystem overload (limit 10k–100k files per directory).
   Suffixes are appended (F for files, W for workspace, M for module, etc.)

---

10. Path generation
    ProcessAndBuildStoragePath() uses the following sequence:

- Fetch base path: BasePath + ClientPath + ModulePath + WorkspacePath
- If managed workspace:
  - If SaveAsName exists, extract shard path
  - Else if file CUID exists, retrieve version info from DB and use stored path
  - Else if name provided, look up via DB by name

    - Else if uploading: generate controlled ID (Numeric or GUID), compute shard path
- Combine: BasePath + controlled shard path + file name (with extension)
- Return the final absolute path and basePath for folder operations.

---

11. Case sensitivity
    Windows is naturally case insensitive; Linux is case sensitive.
    Haley supports per-client-module overrides loaded from Seed:OSSSource config.
    The service maintains a list _caseSensitivePairs and applies:

- Case-sensitive or case-insensitive matching in directory scanning
- Case-sensitive or insensitive path generation
- Distinct sharding and matching rules
    Used in: ProcessFileRoute, GenerateBasePath, Download (extension search), DeleteDirectory.

---

12. Upload flow (very detailed)
    Upload(IOSSWrite input):
13. Ensure write-mode enabled.
14. Generate CallID to track DB transaction grouping.
15. Build target path using ProcessAndBuildStoragePath:
    - Resolves workspace, directory, file name, controlled ID rules.
    - Queries DB for previous version path if existing.
16. Validate final target path.
17. If file exists:
    - Replace mode: overwrite or use TryReplaceFileAsync
    - Revise mode: copy current file to new version path, then replace
18. Write the stream with configurable buffer size.
19. After writing, if indexer enabled:
    - Call UpdateDocVersionInfo() to store file size, path, display name
    - FinalizeTransaction(CallID, commit=true) to commit all DB operations
      If any step fails, FinalizeTransaction(CallID, commit=false) rolls back DB changes.

Important: physical disk write and DB transaction are separate operations.
This can create FS/DB drift if write succeeds but DB fails, hence recommended two-phase commit improvement.

---

13. Download and streaming logic
    Download(IOSSReadFile):

- Computes path; if file extension missing, searches directory for matching base name using configured case rules.
- Returns IOSSFileStreamResponse containing the FileStream.
  The controller layer handles:
- Range header parsing
- Content-Range, Accept-Ranges, and Content-Length headers
- Streaming the correct slice of the file
  ASP.NET Core's enableRangeProcessing=true can take over this entire responsibility safely.

---

14. Delete file
    Delete(IOSSReadFile):

- Validates write-mode
- Resolves file path
- If exists: TryDeleteFile
- Returns feedback message specifying success or precise error

---

15. Directory operations
    CreateDirectory(IOSSRead): automatically builds path and creates the directory if not exists.
    GetDirectoryInfo(IOSSRead): lists subfolders and files inside the directory.
    DeleteDirectory(IOSSRead, recursive): validates the target, ensures that path matches expected directory for safety, and deletes.

---

16. Registration system
    DiskStorageService can register everything from config or API:
    (a) RegisterClient:

- Creates Client folder
- Writes .client.dss.meta with password hash, signing key, encryption key

- Registers in dss_core
- Auto-registers default module → auto-registers default workspace
  (b) RegisterModule:
- Creates module folder
- Writes .module.dss.meta
- Registers module in dss_core
  (c) RegisterWorkSpace:
- Creates workspace folder unless virtual
- Writes .ws.dss.meta
- Registers workspace in dss_core
  (d) RegisterFromSource():
- Reads Seed:OSSSource entries from config
- Registers clients, modules, workspaces
- Builds case-sensitivity overrides

---

17. Indexing subsystem (MariaDBIndexing) – very detailed
    The indexer is responsible for all DB operations.

It manages:

- Client, module, workspace indexing in dss_core
- Document, directory, version, name_store tables in dss_client_x DB
- Path resolution during read/write
- Transaction coordination

Key features:
(a) Database bootstrap

- Ensures schema exists
- If not, loads and runs SQL files (dsscore.sql, dssclient.sql)
- Can replace schema names dynamically to avoid hardcoding

(b) Registration queries
RegisterClient()
RegisterModule()
RegisterWorkspace()
These perform upserts or updates and confirm existence via EXISTS queries.

(c) Directory/document creation helpers
RegisterDirectories() ensures directory tree in DB.
RegisterFileNameComponents() ensures extension + filename entries exist.
InsertVersion() creates document/version rows, returning ID and GUID.

(d) Version info updates
UpdateDocVersionInfo() stores file's final physical path, display name, size.
IMPORTANT: version_info SQL bug where size=VALUES(path) must be corrected.

(e) Document lookup
GetDocVersionInfo() supports:

- Lookup by doc-version CUID
- Lookup by doc-version ID
- Lookup by (workspace, directory, filename, extension)
  Useful for reading files where only logical name is provided.

(f) Transaction handler management
The service uses a transaction dictionary:
key = callID###dbName
value = (handler, createdTime)
This ensures that all DB actions for a single upload operation across multiple DBs are committed or rolled back together.
FinalizeTransaction(callId) loops over all keys starting with that call ID prefix.

This guarantees consistency inside DB but does not guarantee filesystem rollback.

(g) Caching
_cache: stores CUID → component info (client/module/workspace).
Used to speed up lookup and path generation.
_pathCache: stores CUID → constructed base path string.

---

18. Meta files
    Meta files store full serialized information in JSON form for:
    .client.dss.meta
    .module.dss.meta
    .ws.dss.meta
    They allow the system to rebuild component information even if the DB is not present. Also used to repopulate cache.

---

19. Security
    Base DiskStorageService only provides a stub AuthorizeClient which always approves.

Real authentication/authorization is expected to be handled at API layer (e.g., ADJL with JWT, API Key, viewer cookies, etc.).
The meta files also store sensitive keys (signing, encryption), so disk security is critical.

---

20. Streaming controller (VaultViewerControllerBase)
    Streams video, audio, images, and PDFs.
    Supports manual range parsing and streaming slices of files.
    Sets Accept-Ranges and Content-Range headers.
    For videos, seeking works perfectly because of full Range support.
    For PDFs, omit Content-Disposition:inline if you do not want opening in a new tab.
    Better practice: use File(stream, contentType, enableRangeProcessing:true) where possible.

---

21. Cross-platform considerations
    Haley Storage handles differences between Windows and Linux:

- Case sensitivity per client/module
- Path normalization
- Sharding rules compatible with both OS
  Linux recommended for better scalability and reliability.

---

22. Improvements recommended

- Implement two-phase upload (temp file, DB commit, atomic rename) to avoid FS/DB drift.
- Add transaction cleanup TTL to remove stale transaction handlers.
- Consolidate case-sensitivity logic via single helper.
- Validate Range header more robustly using long values.
- Move secrets (JWT, DB passwords) out of appsettings.json.
- Improve DeleteDirectory safety checks using normalized canonical paths.
- Correct SQL bug in version_info UPSERT.
- Add ETag support using document-version GUID.

---

23. Overall architecture flow
    Vue App → Razor API (ADJL) → DiskStorageService → MariaDBIndexing → MariaDB + Filesystem (Linux LVM)

Clients interact with Razor endpoints, which call DiskStorageService for:

- Upload (with versioning + index)
- Download (streaming)
- Directory management

The indexer ensures metadata is stored correctly, and filesystem stores the actual bytes.

---

24. Big-picture role
    Haley Storage is effectively an S3-like document-store subsystem designed for:

- CDE platforms
- On-prem enterprise file management
- Large-scale file storage with controlled hierarchy
- Secure metadata + versioning
- Streamable media + documents
- Integration with larger Haley ecosystem (WorkflowHub, LifecycleState, etc.)

It forms the foundation for your future Khopu Document Service and ISO 19650 compliant CDE.