Project FileVault – Week 1

Milestone 1: Build the FileVault CLI

📘 Objective

Welcome to Week 1 of the FileVault project!

In this milestone, you will build a **local command-line interface (CLI)** for interacting with a simple file manager system. This CLI will allow users to upload, list, retrieve, and delete files using command-style inputs, just like `git`, `curl`, or `docker`.

You will create a simple **Command Pattern–based CLI**, with a clean modular structure and routing. You'll also set up a **basic data storage layer** and **start separating core business logic into a service layer.**

> ⚠️ **This milestone is purely CLI-based.**

Concepts to Learn & Practice

- The **Command Pattern** for CLI design
- Basic file storage and I/O operations
- Separation of concerns: CLI layer vs. business logic
- Storing file metadata
- Error handling and basic validation
- Project structure and modularization

Required Commands

You must implement the following commands in your CLI tool:

1. `upload <filepath>`

Uploads a file to the system and saves its metadata.

- Stores a copy in a local folder (e.g., `uploads/`)
- Stores metadata (filename, upload time, size, etc.)
- Generates a unique `file_id` (can be a UUID or slug)

Example:

```
$ vault upload ./notes.txt
File uploaded successfully! ID: 8f7a1c21
```

2. `list`

Lists all uploaded files with basic metadata.

Example:

```
$ vault list
ID         | Name        | Size     | Uploaded At
-----------|-------------|----------|---------------------
8f7a1c21   | notes.txt   | 2.4 KB   | 2025-06-24 22:40:01
a9d3cfe1   | photo.jpg   | 1.2 MB   | 2025-06-25 09:13:45
```

3. `read <file_id>`

Displays metadata for a specific file.

Example:

```
$ vault read 8f7a1c21
Filename: notes.txt
Size: 2.4 KB
Path: ./uploads/notes.txt
Uploaded at: 2025-06-24 22:40:01
```

4. `delete <file_id>`

Deletes a file from the system (both metadata and actual file).

Example:

```
$ vault delete a9d3cfe1
 File deleted successfully!
```

Project Requirements

- The CLI tool should be executable from the terminal, e.g., `vault upload ...`
- Use the **Command Pattern** to route commands.
- Store files in a local folder (e.g., `./uploads/`)
- Store file metadata persistently (in a JSON file)
- Must use a separate **service layer** (`FileService`) to handle core logic
- Log or display user-friendly error messages (e.g., "File not found")
- Ensure basic input validation (e.g., check if file exists before upload)

Suggested Folder Structure

```
vault/
├── cli/
│   ├── command_router.py
│   ├── commands/
│   │   ├── upload.py
│   │   ├── list.py
│   │   ├── read.py
│   │   └── delete.py
├── services/
│   └── file_service.py
├── storage/
│   ├── metadata.json
│   └── uploads/
├── utils/
│   └── helpers.py
├── main.py
└── README.md
```

Architecture Diagram

- Project Architecture
- CLI Pattern Architecture

Bonus

- Add file type validation (e.g., reject `.exe` files)
- Add human-readable file size formatting (e.g., 1048576 → 1 MB)
- Use UUIDv4 or a custom slug generator for `file_id`

📦 Deliverables

- `main.py` (or equivalent) that serves as the entr point and can run from the terminal

- `README.md` explaining:
  - How to run the CLI
  - Supported commands
  - How files and metadata are stored
- A `storage/` folder with uploaded files and metadata
- Clear separation of CLI layer and service logic

✅ Milestone Completion Checklist

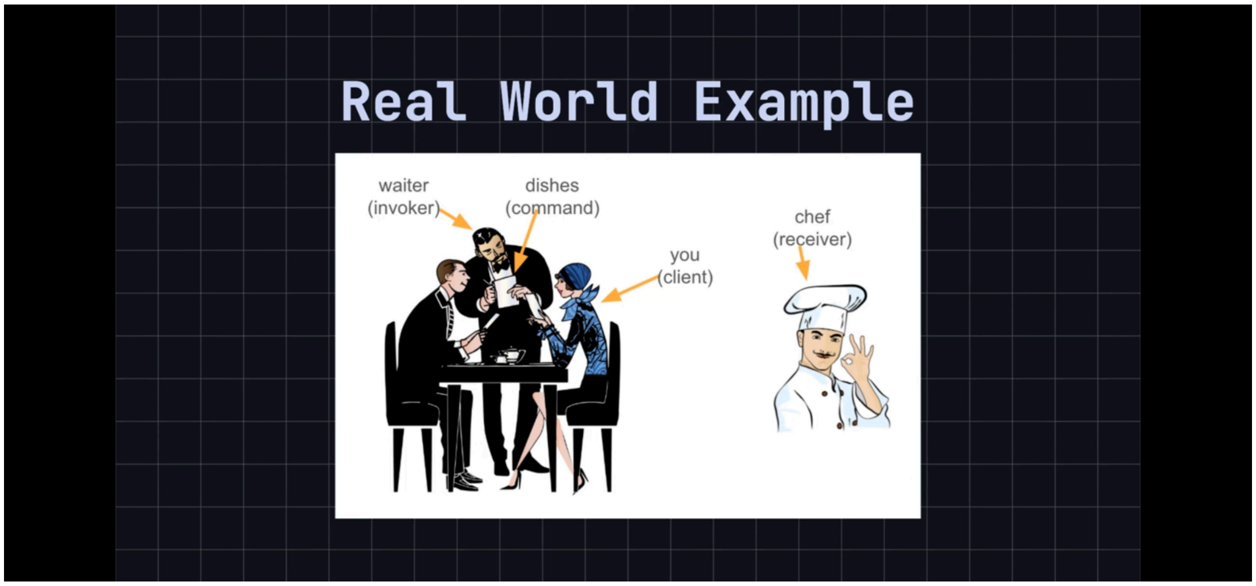
Before moving to the next milestone, ensure the following:

- [ ] You can upload a file via CLI
- [ ] You can list all uploaded files
- [ ] You can fetch file metadata
- [ ] You can delete a file
- [ ] All commands run with clear, validated input
- [ ] You've used the command pattern to separate commands
- [ ] All file logic are in a `FileService` class/module

📚 Resources

- [Command Pattern – Refactoring Guru]()

- UUID – Python docs
- Pathlib for file operations
- Building a CLI Tool In Golang
- File Handling - Rust
- Creating a CLI tool in Nodejs


Real World Example