Audion Hub Manager — why it is built this way

Русский · About · User guide · How it is built

Contents

- Decision
- Why apart
- What follows
- Decision
- The only source of truth
- A real write needs an explicit intent
- A filter cannot silently become a full copy
- The project's word stands over the profile
- Writing goes in phases, deleting comes last
- Three ways of comparing
- Differences are not success
- Small things that once cost dearly
- Checks that must stay green
- Decision
- What it means in practice
- An exception that is not one
- Decision
- Why
- What follows
- Decision
- Why
- Decision

- Why

The decisions that shaped the program. None was made for beauty; each was made because the other way was worse.

---

1. Three layers instead of one big folder

Decision

A project lives in three layers that may share a root but stay apart in meaning.

```
SOURCE      the whole live project
GIT COPY    a filtered technical copy with Git
DOCS        a neutral folder of notes and descriptions
```

Why apart

The SOURCE is complete. With its builds, caches, logs and the settings of the machine — or it cannot be worked on. Git history only gets in the way of such a folder: it fills up with what is rebuilt every day.

The GIT COPY is technical. It holds what the profile lets through: source code, documentation, settings without secrets, launchers. It is the layer for comparing, committing, the server and reusing code.

DOCS is neutral. The layer does not scan thousands of source files. It holds Markdown, pointers, overviews, notes. It is synced by anything — an editor, a file manager, Obsidian, LogSeq, a cloud or nothing. The GIT COPY is kept through Git and is not needed on a phone.

What follows

The DOCS layer is not checked on its own: it is a reading layer, and the canonical technical copy is in the GIT COPY already. The check runs only between the SOURCE and the GIT COPY.

---

2. A mirror that cannot spoil the source

Decision

The rules below were gathered from real breakages, and there is no need to repeat them.

The only source of truth

The SOURCE is the only source of truth. The GIT COPY is derived: the mirror may delete and rebuild its files, but it **never** changes the SOURCE.

The SOURCE may be an ordinary Git working tree — the mirror does not care, it reads only what the profile allows. The GIT COPY is a Git tree too, and its `.git/**` is protected from the walk, deletion and upkeep.

A real write needs an explicit intent

A profile declares the preview the default, and the command line may not ignore it:

```
hub_cli.cmd mirror apply PROJECT             a preview, if the profile says so
hub_cli.cmd mirror apply PROJECT --apply     a real write
hub_cli.cmd mirror apply PROJECT --dry-run   a forced preview
```

In the window it is the tick "dry run" at APPLY.

A filter cannot silently become a full copy

A profile must declare that filters are required and how many masks at least. With no masks the plan fails instead of building a full duplicate of the project. Only a separate flag of a profile allows a full copy, and no profile the program ships sets it.

The project's word stands over the profile

A profile is common to every project; `.gitignore` is the project's own word on what not to keep. What it hides the mirror does not copy and `check-paths` does not offer to a commit. Otherwise keys in `config\keys\` or a build tree in `build-ui\` would lie in the GIT COPY as a spare copy on disk, even if they never reached the history. The profile drops none of its own bans: a project without a `.gitignore` is guarded by it alone.

The profile's guesses about build folders give way to the same word. `app\` of Next.js holds pages, `build\` of Electron icons, `bin\` scripts, `vendor\` of Go committed dependencies. Where the project has a `.gitignore`, it says what a build is; the guesses work only without one. The bans of safety — secrets, binaries, heavy files — hold always.

Writing goes in phases, deleting comes last

```
1  create the allowed folders
2  copy and update files
3  set the modification times
4  if there were no errors and no differences – delete what is extra in the GIT COPY
5  remove folders left empty
6  markers of empty folders – only where a profile asks
```

Deleting does not begin until copying has succeeded. When any earlier phase fails, deleting is skipped and the report says so plainly. That is exactly the case where the mirror would lose files that exist only in it; no setting of a profile turns this off.

Three ways of comparing

Way	How it compares	When
quick	size and time	a rough check
safe	size and time, sums only when the size is the same and the time differs	everyday work
strict	sums for every file of the same size, even with the same time	checkpoints

The safe way is fast but on purpose does not catch "same size, same time, other content". Checkpoints need the strict way — and the profiles shipped keep it.

Sums always carry their algorithm: `blake3:…` or `sha256:…`. SHA-256 is a fallback, not an equal.

Differences are not success

Any real apply with errors or differences returns a non-zero exit code.

Small things that once cost dearly

Report names with microseconds — otherwise fast automation overwrites its own reports.

The area of the mirror is declared. "Filtered" means the GIT COPY is brought to the SOURCE only by the chosen masks; files outside the area are left alone by the plan.

A plan is checked again before writing. A stale or edited plan could have deleted under a path overlapping the SOURCE — the rules and the ban on overlapping are checked anew.

Checks that must stay green

```
test_profile_requires_include_masks
test_strict_mode_detects_same_size_same_mtime_different_content
test_safe_mode_keeps_disk_auditor_fast_path_same_size_same_mtime
test_delete_phase_is_skipped_when_copy_phase_fails
test_projection_mirror_preserves_empty_service_dirs
test_projection_preserves_empty_dirs_with_gitkeep
```

A profile may keep empty folders by a `.gitkeep` marker: it goes into every empty folder, is taken out when real files appear there, and is never put into hidden technical folders. The profiles shipped put no markers: a program makes its service folders itself (`install\init_folders.cmd`), and empty placeholder files only clutter the history.

3. The program keeps no passwords

Decision

Hub Manager does not become a store of tokens and passwords. The program is portable, and Git credentials do not live inside a portable folder.

Access is left to the usual means:

- SSH keys and ssh-agent;
- the credential manager for HTTPS (Git Credential Manager);
- the command-line tools of GitHub and GitLab;
- VS Code and GitKraken — for signing in by hand and visual work.

The program runs real `git` commands and shows their output whole. If you have signed in through any of the above, push and pull work, and the program knows no secret.

What it means in practice

A remote is a named address:

```
git remote add github git@github.com:audion/Audion_Hub.git
git remote add gitlab git@gitlab.com:audion/Audion_Hub.git
git remote add local_nas file:///Z:/git-mirrors/Audion_Hub.git
```

An address with a password or a token inside is not saved. The access check recognises the kind of address and tells what you will be known by — but it stays a check, not an editor of credentials.

An exception that is not one

For own Forgejo and Gitea servers the program takes a personal access token: it checks it by a request to the server and hands it to your credential helper. The token gets into neither the program's files nor the log, and after that `git push` takes it from the store by itself. "Forget the token" removes it from this machine's store.

The known addresses of servers lie in the settings openly — without tokens.

4. One table of Git commands

Decision

All the Git commands of the program are one table, the canon: the command with its places, risk, mode, the words in two languages "what it does" and "how to undo it". The window writes no Git

command of its own: a button is an entry of the canon, and its caption is the command as typed in a terminal.

Why

The program teaches while it is used. One sees which command went and reads what it did and how to undo it. A month later one types it oneself.

One place, one truth. A test runs every entry of the canon on a fresh copy of the Git lab; the tables of the documents are written from the canon and checked by a test. A button, a document and a check cannot part.

The risk is seen before the click. Reads, changes, network, danger — a dot at the button. Danger asks with the whole command, what is not committed now and how to undo it; the rare and history-rewriting (`rebase -i`) is a template one finishes oneself.

What follows

`pull` goes by fast-forward only: the merge is made and seen by the person. Only what the project's profile lets through goes into the index and a commit — no `git add .`: otherwise the point of the filter is lost on the first commit.

5. The window is the face of the console command

Decision

Everything the window does beyond Git — the registry, the mirror, the checks, the batches, the lab, Forgejo — goes through `hub_cli.cmd` with one JSON per answer. Git the window calls by itself.

Why

What the window can do, a script, an agent and a person in a console can do too — without the window and without differences. The Python engine is held by its tests and is not rewritten for the

window. Git, though, the window calls directly: the command it shows is the lesson, and the state of a repository is read too often to go for it through a middleman.

6. A lab instead of examples

Decision

The repositories to learn on do not lie in the program ready-made — `lab.py` builds them: ten repositories in known states, with fixed authors, dates and bytes of files.

Why

One learns on what can be broken. The broken is built anew with one button, with the same hashes on any machine, so the lesson, the test and the description speak of the same commits. The same repositories are the test bench of the canon.