Audion Hub Manager — how it is built

Русский · About · Console command · Decisions

Contents

- The parts
- The window
- The Git canon
- The engine
- The registry and the layers
- The mirror
- Profiles
- Access and Forgejo
- Storage
- Commit the marked
- The lab
- Checks
- The builder
- Documents

The parts

```
Start.exe ──► App\AudionHubManager.exe  (the window: C#, Avalonia, .NET 10 inside the
folder)
                  |                                  |
                  | git — by itself, by the canon    | hub_cli.cmd — one JSON per
command
                  ▼                                  ▼
          the repository of the chosen layer   system_core\cli.py →
system_core\core\*  (Python 3.12 of runtime\)
```

The window is the face of the console command: the registry, the mirror, the checks, the batches, the lab and Forgejo go through `hub_cli.cmd`, and the same can be done without the window. Git the window calls by itself: the command it shows is the lesson, and the state of a repository (status, graph, branches) is read often.

The window

Where	What
`Source\MainWindow.*.cs`	the sections: Changes, History, Branches, Sync, Mirror, Projects, Service; the header and the layer — Project, the column of projects — Side, commit the marked — Marked, the folder panel — Folders, the terminal — Command, Run
`Source\Views\FadeBox.cs`	a text or a row that does not fit fades out over its last 16 px instead of an ellipsis
`Source\Core\Git.cs`	quiet reading for drawing: `status --porcelain=v2 -b`, `log --format`, `for-each-ref`
`Source\Core\Canon.cs`	the canon from `hub_cli canon`: captions, tips, risk
`Source\Core\Hub.cs`	calling `hub_cli.cmd` and reading its JSON; a stdin line (a token) goes only into the process
`Source\Core\Runner.cs`, `ChildJob.cs`	running commands with live output; a Windows job: STOP and closing the window end the whole tree of processes
`Source\Views\GraphCell.cs`	the graph of commits: branch lanes by parents

Where	What
`Source\App.axaml`	four themes and the colours of Git: in the index, changed, new, deleted, conflict, renamed, outside the profile
`Source\Ps\`	the program's PowerShell 7, `App\AudionPs.exe` (Microsoft.PowerShell.SDK)

Every Git command of a button runs with this environment: `GIT_TERMINAL_PROMPT=0` (nothing waits for a password), `GIT_EDITOR=true` and `GIT_SEQUENCE_EDITOR=true` (an editor the window could not show is not opened), the colours through `GIT_CONFIG_COUNT` (`color.ui=always`, `core.quotepath=false`), so the command shown stays the one typed by hand. Git's messages on the error stream are not errors: the exit code tells the outcome.

Dangerous operations ask with the whole command, what is not committed and how to undo it. A line typed by hand is split into its Git calls — the GIT mode, or every `git` between `&&`, `||`, `;`, `|` and line breaks of PS and CMD — and each is judged by its arguments in any order (`Source\Core\TypedCommand.cs`): `reset --hard` and the other modes of `reset`, `commit --amend`, `restore` of the working folder, `checkout --`, `switch -f`, `clean -f`, `push` with force, deletion, mirror or a `+`/`:` refspec, `branch -D`, `stash drop`, `rebase`, `filter-branch`, `reflog expire`, `gc --prune`, `update-ref -d`, `rm` without `--cached`, `tag -d`. What has an entry of the canon asks with its words, `reset --hard` with the typed RESET; recursive deletion of PS and CMD is caught by a filter of words.

The build: `install\Build-App.cmd` — the tests of the engine and the canon, publishing the PowerShell and the window (win-x64, with .NET), then the self-check of the built window (`--self-test --just-built`): the console command answers, the canon reads and every operation has its words in both languages, the theme has the colours of Git, the number of the exe is that of `config\version.json`. The build trusts only the line `[OK] all checks passed` of the report.

The Git canon

`system_core\core\git_canon.py` — 63 entries. The fields of an entry: `id`, the command with places (`{path}`, `{commit}`, `{branch}`, `{remote}`, `{message}`, `{tag}`, `{paths}` — a list), risk, mode, the window's section, the Russian and English words "what it does" and "how to undo it". The caption of a button (`short`) is the command without its places and quiet flags: `git restore --staged`, `git switch -c`.

Risk		Mode	
`read`	changes nothing	`run`	at once
`local`	changes this repository	`ask`	after a question
`net`	talks to a server	`template`	into the terminal's line, to be finished
`danger`	can lose work		

The canon is read by the window (`hub_cli canon`), the tests (`tests\test_git_canon.py` runs every entry on a fresh copy of the lab) and the documents (`hub_cli canon --docs` writes the tables between `<!-- canon:… -->` and `<!-- /canon -->`; `tests\test_canon_docs.py` fails when a table falls behind). The program has no other Git command.

The engine

Module of `system_core\core\`	What
`hub_ops.py`	registry and layers, the profile check of paths, batches, workspace, addresses and `remotes.json`, the lab in the registry, the machine's tools, the Recycle Bin
`project_registry.py`, `project_importer.py`	reading the registry; finding project roots in a folder and merging them in
`projection_engine.py`, `scan_engine.py`, `mask_engine.py`, `copy_engine.py`	the mirror: the walk by profile, masks, copying, BLAKE3 sums, reports
`docs_view.py`	the DOCS layer: the Markdown and texts of the SOURCE by the profile `markdown_only_projection`
`safety.py`	the scan for secrets and heavy files
`storage_layout.py`	the storage roots and their roles
`git_engine.py`	`git status` for batches, version tags of a series, the check of a remote address
`git_credentials.py`	the `git credential` protocol: fill, approve, reject

Module of `system_core\core\`	What
`forgejo_api.py`, `forgejo_service.py`	the Forgejo and Gitea API, sign-in by token, repositories
`auth_doctor.py`	the access check: Git, name and e-mail, gh, glab, SSH, remote addresses
`external_apps.py`	the path of this machine's VS Code
`output_decode.py`	decoding output (UTF-8, OEM, Cyrillic)
`lab.py`, `git_canon.py`, `canon_docs.py`	the lab, the canon, the canon tables of the documents

The registry and the layers

`<root of the projects>\hub_projects.json` (`project_registry.registry_path`): each root has a registry of its own. `active_project_id` and `projects` — records with `id`, `title`, `source_path`, `projection_path`, `docs_path`, `profile`, `default_branch`, `notes`. A relative path counts from the folder of the registry; paths inside the root are written from it, the others in full, and the root moves together with its registry. A record without a GIT COPY gets `Hub Data\<title>` beside the program — as a scan gives it. A broken file does not stop the start: the surviving records are read, the answer names the trouble (`load_error`). The registry is this machine's data and does not go into git. When it is missing the program makes it: from `config\projects.json` when the registry lies there (the paths are rebased from the program folder to the root, the old file is removed); for the shipped root from `config\projects.example.json`; for a root of one's own empty.

`projects export FILE` writes the registry into a file with full paths; `projects import FILE` rebases the paths of the file from where it lies to the root and adds the projects the registry does not have (`--replace` takes the file's list whole).

Each layer (`source`, `mirror`, `docs`) has a repository state: `own`, `parent` (a folder inside another repository: the commands would go there), `none`.

The mirror

```
walk the SOURCE by the profile → walk the GIT COPY → compare the paths
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 (`preserve_empty_dirs`); the
profiles shipped put none
```

The SOURCE never changes; `.git/**` is protected from the walk, deletion and upkeep. A plan is checked again before writing: a stale or edited plan deletes nothing outside its area. Neither walk follows a symbolic link or a junction (`lstat`, the reparse tag): such a folder or file is skipped, and every write, copy and delete resolves its path and refuses one that leads out of its layer. A folder or file a walk could not read becomes an `error` item of the plan (`source_unreadable`, `mirror_unreadable`), so step 4 deletes nothing; verification counts it as an error. Reports — `logs\<project>\YYYYMMDD_HHMMSS_microseconds_kind.json`.

Compare	How
`quick`	size and time
`safe`, `safe_blake3`	sums only when the size is the same and the time differs
`strict`, `strict_blake3`, `metadata_then_blake3`	sums for every file of the same size, even with the same time; `metadata_then_blake3` is the default

Sums carry their algorithm: `blake3:…`; `sha256:…` is the fallback where BLAKE3 is missing. Any apply with errors or differences returns exit code 1.

Profiles

`config\projection_profiles.json`:

Profile	What
`audion_python_project_projection`	a project of any kind; 476 include masks, 243 exclusions, 89 folders not walked

Profile	What
`audion_python_project_projection_strict`	extends the project profile (`extends`), the strict check
`markdown_only_projection`	Markdown and texts only — the DOCS layer

The project profile lets through the sources, documents and settings of the common languages and build systems: Python, the web (JS, TS, frameworks), C and C++, .NET, the JVM, Go, Rust, Swift, Dart and Flutter, Ruby, PHP, Elixir, Haskell, R, Julia, Lua, shells and PowerShell, SQL, Terraform, Nix, Docker and Kubernetes, CMake, Bazel, Meson, Qt (forms, resources, QML), WiX, Inno Setup and NSIS installers, translations, templates, diagrams as text; dependency lock files, CI (GitHub, GitLab, Forgejo, Gitea and others), linter and formatter settings, the shared `.vscode\tasks.json`, `launch.json`, `extensions.json`, `.devcontainer`; the text files of Unity and Godot. Pictures and fonts of the sources pass up to a megabyte. It keeps out dependencies and caches (`node_modules`, `.venv`, `target`, `obj`, `.gradle`, `.next` and dozens more), builds and packages, archives and disk images, media and the files of design programs, databases, data sets and trained models, logs and temporary files, `.env`, private keys, keys and tokens kept in text files (`api_key_*.txt`, `token.json`), password stores, the state of Terraform, machine-local `*.local.*` and `*.user`; for Audion programs, the built window `App\`, its state, the licence pack and the release manifest.

The project's own `.gitignore` stands over the profile: what it hides the mirror neither walks nor copies (`respect_gitignore`). A project with a repository of its own is asked as it is — what it has committed is never hidden; any other folder through a throwaway git folder, so that only its own `.gitignore` counts, not the one of a repository it lies in. Hidden folders of the first two levels are cut off before the walk: a build tree in `build-ui\` is not walked at all.

Hundreds of masks are checked in a fraction of a second: exact names by a set, endings like `*.py` by one check of the ending, the rest by one compiled expression (`mask_engine._matcher`); the answer is that of the masks one by one (`tests\test_mask_engine.py`).

Key of a profile	What it does
`dry_run_default: true`	an apply without `--apply` only shows
`require_include_filter: true`, `min_include_globs`	with no masks the plan fails
`allow_unfiltered_full`	the only way to allow a full copy; no profile the program ships sets it
`mirror_scope: "filtered"`	the GIT COPY is brought to the SOURCE only by the masks

Key of a profile	What it does
`compare_mode`	the way of comparing; the profiles shipped use `strict_blake3`
`small_include_globs`, `small_include_max_file_bytes`	small files that pass up to their limit (pictures and fonts — up to 1 MB)
`hide_dirs`	folders the walk never enters
`exclude_dir_contents`	folders that stay empty with a marker: service folders, builds, outputs
`extends`	a profile takes another and changes its own keys
`preserve_empty_dirs`, `marker_file`	keep empty folders by a marker; off in the profiles shipped — a program's folders are made by its `install\init_folders.cmd`
`respect_gitignore`	do not copy what the project's `.gitignore` hides; on by default
`output_dirs`	folder guesses: builds and dependencies in most projects, sources in some (`app\` of Next.js, `build\` of Electron, `bin\` of scripts, `vendor\` of Go); kept out only when the project has no `.gitignore` of its own
`extensionless_text_bytes`	a file without an extension the list does not name (`install`, `UNLICENSE`, the hook `.husky\pre-commit`) passes when it is within the limit and text: no zero byte in its first 8 KB

A folder rule is a name anywhere (`logs`), a path from the root (`install/download`) or, with a leading `/`, a name at the root only (`/App` — the built window, not an `app` folder of sources deeper down). A name may be a mask: `cmake-build-*`.

The same profile decides what goes into a commit: `hub_cli check-paths` before `add` and `commit`, a file outside the profile is not put into the index.

Access and Forgejo

The program keeps no passwords, tokens or private keys. Access goes by the usual means: SSH keys and ssh-agent, Git Credential Manager, GitHub CLI, GitLab CLI. The checks never wait for input: `GIT_TERMINAL_PROMPT=0`, SSH with `BatchMode=yes`.

Own Forgejo and Gitea servers — `config\forgejo_hosts.json`: the address, the SSH user and port, the kind of address, the login; no tokens. It is data of the machine, as the registry is: the file does not

go into git, the release cleaner removes it. The account is a personal access token with the header `Authorization: token …`: it is checked by a request to `/api/v1/user` and handed to Git's credential helper (`git credential approve`); after that `git push` over HTTPS takes it by itself. OAuth2 sign-in is not used: Forgejo's OAuth2 has no scopes yet, such a token carries the account's administrative rights, lives briefly and needs a separate mechanism for `git push`.

Storage

`config\storage_layout.json` — the roots and their roles; `${APP_ROOT}` is the program folder:

Root	Role	As shipped
`manager_root`	the program itself	`${APP_ROOT}`
`hub_data_root`	the GIT COPIES of projects	`${APP_ROOT}/../Hub Data`
`docs_root`	the DOCS layer	`${APP_ROOT}/../Hub Docs`
`full_projects_root`	the live projects, the source of truth	`${APP_ROOT}/..`

`hub_cli storage` checks the roots are there, none lies inside another, the GIT COPY of each project is under `hub_data_root` and its DOCS under `docs_root`. DOCS is optional: the docs root is needed only when at least one project keeps the layer. The demo project and the Git lab keep their copies inside the program and are not held to the layout.

The root of the live projects can be chosen for one machine: `hub_cli storage --projects-root FOLDER` writes it into `config\storage_layout.local.json` (beside the program it is kept as `${APP_ROOT}/../…`, elsewhere in full; inside the GIT COPIES or DOCS it is refused), `--reset-projects-root` gives the shipped root back. The file is not in git and the release cleaner takes it (`*.local.json`): a release carries the layout of the program, not of the machine it was made on. A change of the root answers with exit code 0 when the root is written; what of the projects of the new root the layout does not hold is told by `off_layout` (`source`, `projection`, `docs` — lists of `id`), and the window names them in a line under the command.

Commit the marked

`Source\MainWindow.Marked.cs`: a checkpoint of every project marked in the column, by `SKILL.md` — `mirror apply --apply`, `mirror verify`, `safety --layer mirror`, `check-paths --layer mirror --stdin` over the changed paths, `add` and `commit` of the canon in the

GIT COPY. The steps go through the same runner as the buttons; the next step is put in when the one before has ended whole (`Done` → `Dispatcher.Post`). A project without a repository of its own in the GIT COPY is passed over, a step that says no stops only its project, STOP the whole chain.

The lab

`system_core\core\lab.py` builds ten repositories with a clock of its own: the author, the date and the bytes of every file are fixed, so the commit hashes are the same on any machine and in every run. `_servers\` holds bare repositories, the "server" of the lesson on remotes.

Checks

What	How
engine, console command, canon	`runtime\python.exe -m pytest -q tests` — 172 tests; the canon on a fresh copy of the lab
window	`install\Build-App.cmd` — the self-check of the built window
release	`install\verify_portable_env.cmd` — the environment (`system_core\doctor.py`), the console command, `Start.exe`, the window by its report

The builder

`builder_main.cmd` — a menu on FZF:

Step	What
01, 02	the Python environment: `runtime\` and `wheelhouse\` from `install\requirements_full.in` (CMD or PowerShell)
03	`system_core\fzf.exe`
05	the BLAKE3 package
06	the program's folders
07	the lock file of dependencies

Step	What
09	an offline install from `runtime\` and `wheelhouse\`
70	cleaning the install cache and what a build of the window leaves
71	the release check
72	the encoding of `.cmd`: UTF-8 without BOM, CRLF
95–98	open `install\`, `runtime\`, `wheelhouse\`, `licenses\`

The window is built apart: `install\Build-App.cmd`. `cleanup_project.cmd` brings the project back to its sources: it removes the runtime, caches, logs, the state of the window and what a build leaves; `App\` and `Start.exe` stay.

Documents

Markdown is the source. The canon tables of the documents are written by `hub_cli canon --docs`, never by hand. The DOCS layer is not checked by BLAKE3: its technical copy is in the GIT COPY already.