Audion Hub Manager — the console command

Русский · About · User guide · Git wiki

Contents

- The canon and the number
- Projects
- The profile and the check of paths
- The mirror and DOCS
- Checks
- Batches and the workspace
- The lab and versions
- Remotes
- Forgejo and Gitea

`hub_cli.cmd` in the program folder is everything the window does beyond Git: the registry, the mirror, the checks, the batches, the lab, Forgejo. A button of the window and this command do the same thing. Git itself the window calls directly, by the canon; `hub_cli.cmd canon` gives its table.

The command runs `system_core\cli.py` on the program's Python (`runtime\python.exe`) and prints one JSON document. It always has `ok`; on an error also `error` and `kind`: `usage` (a wrong call) or `failed`. A call the command does not understand — an unknown command or option, a missing argument — answers the same way, with `kind: usage`.

Exit code	Means
0	done
1	done with a bad result (the mirror differs, a check found a blocker or could not read a file) or failed
2	a wrong call: no such project or profile, a missing argument; nothing was changed

The reports of the mirror, the checks and the batches go as dated JSON files into `logs\<project>\` (for the registry and the batches, `logs\registry\`); the answer names the report (`report`).

The canon and the number

```
hub_cli.cmd version                    the number of the program and its folder
hub_cli.cmd canon                      the Git canon: every operation of the window
(JSON)
hub_cli.cmd canon --docs               write the canon tables of docs\*.md and
SKILL.md anew
```

An entry of the canon: `id`, `args` (the command with places `{path}`), `display` (with places `‹path›`), `short` (the caption of a button), `places`, `risk` (`read`, `local`, `net`, `danger`), `mode` (`run`, `ask`, `template`), `section`, `ru` and `en` with `what` and `undo`.

Projects

```
hub_cli.cmd projects list [--fast]                          the registry and the state
of each layer
hub_cli.cmd projects active ID                              make a project the active
one
hub_cli.cmd projects set ID [--title T] [--source P] [--mirror P] [--docs P] [--
profile X] [--branch B] [--notes N]
hub_cli.cmd projects scan FOLDER [--import] [--hub-root P] [--docs-root P] [--depth
N] [--dry-run]
hub_cli.cmd projects remove ID [--dry-run]                  take a record out of the
registry; folders are not touched
hub_cli.cmd projects clean [--dry-run]                      drop records without
folders and duplicates
hub_cli.cmd projects export FILE                            the registry of the root
into a file, every path in full
hub_cli.cmd projects import FILE [--replace] [--dry-run]    the projects of a file
into the registry of the root
```

The registry is `hub_projects.json` in the root of the projects (`storage`): each root has its own. Without it the program makes it: for the default root from `config\projects.example.json`, for a root of one's own empty; a registry in `config\projects.json`, if there is one, moves into the root when first read. `import` reads the paths of the file from where it lies and writes them from the root;

projects whose `id` the registry has stay as they are, `--replace` takes the file's list whole. The answer names the added (`added`) and the passed over (`skipped`).

`list` gives each project its `layers`: `source`, `mirror`, `docs` with `path`, `exists` and `repo` — `own` (a repository of its own), `parent` (a folder inside another one) or `none`. `--fast` skips asking Git.

`set` edits the fields of an existing record; paths inside the root of the projects are written from it (the default root is the folder beside the program, where the program and `Hub Data` both lie), elsewhere or on another disk in full. `scan` makes new records: without `--import` it only shows the candidates, with `--import` it adds them to the registry. A folder that is a project itself (it has `.git`, `README.md`, `pyproject.toml` or the like) finds itself; a shared folder without such marks finds the projects in its subfolders down to `--depth` (3). A candidate's GIT COPY is `--hub-root\<name>`, without it `Hub Data` beside the program folder.

The profile and the check of paths

```
hub_cli.cmd profile ID                                  the project's mirror
profile: what it lets through, what not
hub_cli.cmd check-paths ID --layer source|mirror PATH...   which paths the profile
lets into a commit
hub_cli.cmd check-paths ID --layer source|mirror --stdin   the same, the paths on the
input, a line each
```

`check-paths` answers `allowed` (paths that may go into the index), `blocked` (a path and the reason: a mask of the profile or a folder the profile does not walk) and `ignored` (what the repository's own `.gitignore` hides: the project's word on what never to commit stands over the profile). A folder counts as every file under it. Exit code 1 when something is blocked. The window asks this before `add` and `commit`; an answer without `allowed` (the profile could not be read) means the check did not happen, and the window takes no file. The thousands of paths of a big project do not fit one command line of Windows: they go on the input (`--stdin`), and a Git command of the canon with them takes them from its input by itself — `git add --pathspec-from-file=- --pathspec-file-nul`, the terminal names their number.

The mirror and DOCS

```
hub_cli.cmd mirror plan ID                  the plan SOURCE → GIT COPY, writes nothing
hub_cli.cmd mirror apply ID                 apply; with the profile's dry_run_default
only show
hub_cli.cmd mirror apply ID --apply         a real write
hub_cli.cmd mirror apply ID --dry-run       a forced preview
hub_cli.cmd mirror verify ID                SOURCE against GIT COPY by BLAKE3; exit
code 1 when they differ or a file
                                            could not be read
hub_cli.cmd docs plan ID                    the plan of the DOCS layer: the Markdown
and texts of the SOURCE
hub_cli.cmd docs apply ID [--dry-run]       write the DOCS layer
```

Every profile the program ships has `dry_run_default`: `mirror apply` without `--apply` writes nothing. `docs apply` writes at once unless `--dry-run` is given; exit code 1 when something could not be written.

The mirror walks neither symbolic links nor junctions, on either side: such a folder or file is skipped, and nothing is written through it. A file or folder the scan could not read is an error of the plan: while it stands, `apply` deletes nothing in the GIT COPY.

Checks

```
hub_cli.cmd safety ID [--layer source|mirror]      secrets and heavy files; exit code
1 on blocking findings
                                                   and on files it could not read
(`errors`)
hub_cli.cmd storage                                the roots of the program, the GIT
COPIES, DOCS and the projects
hub_cli.cmd storage --projects-root FOLDER         make FOLDER the root of the live
projects of this machine
hub_cli.cmd storage --reset-projects-root          give the shipped root back (the
folder beside the program)
hub_cli.cmd blake3                                 real BLAKE3 or the SHA-256
fallback
hub_cli.cmd tools                                  git, gh, glab, ssh, BLAKE3, VS
Code of this machine
hub_cli.cmd auth [--repo P] [--no-mirrors]         access: Git, name and e-mail, gh,
glab, SSH, the remote addresses
```

`storage --projects-root` and `--reset-projects-root` answer with exit code 0 when the root is written (into `config\storage_layout.local.json`); the layout of the projects of the new root is told by `result.ok` and `off_layout`: `source` — no SOURCE folder, `projection` — a GIT COPY outside the root of the GIT COPIES, `docs` — DOCS outside its root. The demo and the lab are not held to the layout. A root inside the GIT COPIES or DOCS is refused (exit code 2).

Batches and the workspace

```
hub_cli.cmd batch status          git status of every project of the registry
hub_cli.cmd batch mirror-plan     the mirror plan of each project
hub_cli.cmd batch safety          the scan for secrets in every project
hub_cli.cmd batch verify          the BLAKE3 check of every project
hub_cli.cmd workspace [ID]        a .code-workspace for VS Code: the layers of a
project or the whole registry
```

A batch has a `summary` with `errors`, the errors inside each project's answer counted too; exit code 1 when there were errors, for `batch safety` also on blocking findings and for `batch verify` on differences. `batch status` reads the layer the history of a project lives in: the GIT COPY when it has a `.git` of its own, otherwise the SOURCE; `layer` of each project says which. A folder without a repository of its own is a state, not an error: `repo: none` and the count `no_git` (the first lesson of

the lab is such); for a folder inside another repository `inside` names that one, and git is not asked — it would answer for the outer one.

The lab and versions

```
hub_cli.cmd lab create [--dir P] [--replace] [--register]
hub_cli.cmd next-version --repo P [--series S] [--bump patch|minor|major]
```

`lab create` builds the ten repositories to learn on in `examples\lab`; an existing folder is replaced only with `--replace`, and the former one goes to the Recycle Bin. `--register` enters them into the registry (`lab_01` … `lab_10`).

`next-version` reads the tags of the series (`projection` by default) and answers `version` (`v0.1.2`), `tag` (`projection-v0.1.2`) and `command` — the git command it read them with.

Remotes

```
hub_cli.cmd remote url --platform X --owner O --repo R [--type ssh|https] [--server
S] [--ssh-port N]
hub_cli.cmd remote list
hub_cli.cmd remote save --name N --url U [--platform X] [--repo R] [--disabled]
```

`url` builds an address: `github`, `gitlab`, `codeberg` are clouds; `forgejo`, `gitea`, `custom` take their own `--server` and SSH port. `save` writes a remote into `config\remotes.json`; an address with a password or a token inside is refused.

Forgejo and Gitea

```
hub_cli.cmd forgejo doctor [--offline]
hub_cli.cmd forgejo check --url U
hub_cli.cmd forgejo sign-in --url U [--login L] [--ssh-port N] [--type ssh|https]
hub_cli.cmd forgejo whoami --url U [--login L]
hub_cli.cmd forgejo repos --url U [--login L]
hub_cli.cmd forgejo create --url U --name N [--login L] [--public] [--description D]
[--branch B]
hub_cli.cmd forgejo sign-out --url U [--login L]
```

`sign-in` reads the token from stdin — one line, not an argument of the command line, so it stays out of the history of commands and the logs. The token is checked by a request to `/api/v1/user` and goes to Git's credential store; it is never written into the program's files. `whoami`, `repos` and `create` take it from the store; with no token the answer is `message: forgejo_token_missing`, exit code 1.

The scopes of a token: `read:user` — who I am and the list of repositories, `write:user` — create a repository, `read:repository` and `write:repository` — `git clone` and `git push` over HTTPS.