Audion Hub Manager — консольная команда

English · О программе · Руководство · Git-вики

Содержание

- Канон и номер
- Проекты
- Профиль и проверка путей
- Зеркало и DOCS
- Проверки
- Пакеты и рабочая область
- Лаборатория и версии
- Remote
- Forgejo и Gitea

`hub_cli.cmd` в папке программы — всё, что окно делает вне Git: реестр, зеркало, проверки, пакеты, лаборатория, Forgejo. Кнопка окна и эта команда делают одно и то же. Git окно зовёт само, по канону; его таблицу отдаёт `hub_cli.cmd canon`.

Команда запускает `system_core\cli.py` на Python программы (`runtime\python.exe`) и печатает один документ JSON. В нём всегда есть `ok`; при ошибке — `error` и `kind`: `usage` (неверный вызов) или `failed`. Вызов, которого команда не понимает, — неизвестная команда или ключ, нехватка довода — отвечает так же, с `kind: usage`.

Код выхода	Значит
0	сделано
1	сделано с плохим итогом (зеркало расходится, проверка нашла блокирующее или не смогла прочитать файл) или не удалось

Код выхода	Значит
2	неверный вызов: нет такого проекта или профиля, не хватает довода; ничего не изменено

Отчёты зеркала, проверок и пакетов ложатся датированными JSON в `logs\<проект>\` (у реестра и пакетов — `logs\registry\`); путь отчёта есть в ответе (`report`).

Канон и номер

```
hub_cli.cmd version                    номер программы и её папка
hub_cli.cmd canon                      канон Git: все операции окна (JSON)
hub_cli.cmd canon --docs               переписать таблицы канона в docs\*.md и
SKILL.md
```

Запись канона: `id`, `args` (команда с местами `{path}`), `display` (с местами `‹path›`), `short` (подпись кнопки), `places`, `risk` (`read`, `local`, `net`, `danger`), `mode` (`run`, `ask`, `template`), `section`, `ru` и `en` с `what` и `undo`.

Проекты

```
hub_cli.cmd projects list [--fast]                          реестр и состояние каждого
слоя
hub_cli.cmd projects active ID                              сделать проект активным
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 ПАПКА [--import] [--hub-root P] [--docs-root P] [--depth N]
[--dry-run]
hub_cli.cmd projects remove ID [--dry-run]                  убрать запись из реестра;
папки не трогаются
hub_cli.cmd projects clean [--dry-run]                      убрать записи без папок и
дубли
hub_cli.cmd projects export ФАЙЛ                            реестр корня — в файл,
каждый путь полностью
hub_cli.cmd projects import ФАЙЛ [--replace] [--dry-run]    проекты из файла — в
реестр корня
```

Реестр — `hub_projects.json` в корне проектов (`storage`): у каждого корня свой. Без него программа заводит его: для корня по умолчанию — из `config\projects.example.json`, для

своего корня — пустым; реестр в `config\projects.json`, если он есть, переезжает в корень при первом чтении. `import` читает пути файла от его места и пишет их от корня; проекты, чей `id` в реестре уже есть, остаются как есть, `--replace` берёт список файла целиком. Ответ называет добавленные (`added`) и пропущенные (`skipped`).

`list` даёт у каждого проекта `layers`: `source`, `mirror`, `docs` с `path`, `exists` и `repo` — `own` (свой репозиторий), `parent` (папка внутри чужого) или `none`. `--fast` пропускает опрос Git.

`set` правит поля существующей записи; пути внутри корня проектов пишутся от него (корень по умолчанию — папка рядом с программой, где лежат и программа, и `Hub Data`), в другом месте или на другом диске — полностью. Новые записи заводит `scan`: без `--import` он только показывает кандидатов, с `--import` — дописывает их в реестр. Папка, которая сама проект (есть `.git`, `README.md`, `pyproject.toml` и подобное), находит себя; общая папка без таких признаков — проекты в подпапках до глубины `--depth` (3). GIT-КОПИЯ кандидата — `--hub-root\<имя>`, без него — `Hub Data` рядом с папкой программы.

Профиль и проверка путей

```
hub_cli.cmd profile ID                                  профиль зеркала проекта:
что пропускает, что нет
hub_cli.cmd check-paths ID --layer source|mirror ПУТЬ...   какие пути профиль пускает
в коммит
hub_cli.cmd check-paths ID --layer source|mirror --stdin   то же, пути — на входе, по
строке
```

`check-paths` отвечает `allowed` (пути, которые можно класть в индекс), `blocked` (путь и причина: маска профиля или папка, которую профиль не обходит) и `ignored` (то, что прячет `.gitignore` самого репозитория: слово проекта о том, что не коммитить, стоит над профилем). Папка считается всеми файлами под ней. Код 1, если что-то заблокировано. Окно спрашивает это перед `add` и `commit`; ответ без `allowed` (профиль не прочитан) значит, что проверки не было, и окно не берёт ни одного файла. Тысячи путей большого проекта не помещаются в одну командную строку Windows: их дают на вход (`--stdin`), а Git-команда канона с ними сама берёт их со входа — `git add --pathspec-from-file=- --pathspec-file-nul`, терминал называет число путей.

Зеркало и DOCS

```
hub_cli.cmd mirror plan ID                  план ИСХОДНИК → GIT-КОПИЯ, ничего не пишет
hub_cli.cmd mirror apply ID                 применить; при dry_run_default профиля —
только показать
hub_cli.cmd mirror apply ID --apply         настоящая запись
hub_cli.cmd mirror apply ID --dry-run       принудительный предпросмотр
hub_cli.cmd mirror verify ID                сверка ИСХОДНИКА и GIT-КОПИИ по BLAKE3;
код 1, если расходятся или файл
                                            не прочитан
hub_cli.cmd docs plan ID                    план слоя DOCS: Markdown и тексты
ИСХОДНИКА
hub_cli.cmd docs apply ID [--dry-run]       записать слой DOCS
```

Все поставляемые профили — с `dry_run_default`: `mirror apply` без `--apply` ничего не пишет. `docs apply` пишет сразу, если не сказано `--dry-run`; код 1, если что-то не записалось.

Зеркало не ходит ни по символическим ссылкам, ни по junction — ни в ИСХОДНИКЕ, ни в GIT-КОПИИ: такая папка или файл пропускаются, и сквозь них ничего не пишется. Файл или папка, которые обход не смог прочитать, — ошибка плана: пока она стоит, `apply` ничего не удаляет в GIT-КОПИИ.

Проверки

```
hub_cli.cmd safety ID [--layer source|mirror]      секреты и тяжёлые файлы; код 1 при
блокирующих находках
                                                   и при файлах, которые не удалось
прочитать (`errors`)
hub_cli.cmd storage                                корни программы, GIT-КОПИЙ, DOCS и
проектов
hub_cli.cmd storage --projects-root ПАПКА          сделать ПАПКУ корнем живых
проектов этой машины
hub_cli.cmd storage --reset-projects-root          вернуть корень по умолчанию (папку
рядом с программой)
hub_cli.cmd blake3                                 настоящий ли BLAKE3 или запасной
SHA-256
hub_cli.cmd tools                                  git, gh, glab, ssh, BLAKE3, VS
Code этой машины
hub_cli.cmd auth [--repo P] [--no-mirrors]         доступ: Git, имя и почта, gh,
glab, SSH, адреса remote
```

`storage --projects-root` и `--reset-projects-root` отвечают кодом 0, когда корень записан (в `config\storage_layout.local.json`); раскладку проектов нового корня говорят `result.ok` и `off_layout`: `source` — нет папки ИСХОДНИКА, `projection` — GIT-КОПИЯ не в корне GIT-КОПИЙ, `docs` — DOCS не в своём корне. Демо и лаборатория раскладке не подчиняются. Корень внутри GIT-КОПИЙ или DOCS не принимается (код 2).

Пакеты и рабочая область

```
hub_cli.cmd batch status          git status по всем проектам реестра
hub_cli.cmd batch mirror-plan     план зеркала каждого проекта
hub_cli.cmd batch safety          проверка секретов во всех проектах
hub_cli.cmd batch verify          сверка BLAKE3 всех проектов
hub_cli.cmd workspace [ID]        .code-workspace для VS Code: слои проекта или весь
реестр
```

У пакета есть `summary` с `errors`, где посчитаны и ошибки внутри ответа каждого проекта; код 1, если ошибки были, у `batch safety` — и при блокирующих находках, у `batch verify` — при расхождениях. `batch status` читает слой, где живёт история проекта: GIT-КОПИЮ, если в ней свой `.git`, иначе ИСХОДНИК; какой слой прочитан, говорит `layer` у каждого проекта. Папка без своего репозитория — состояние, а не ошибка: `repo: none` и счёт `no_git` (так у первого

урока лаборатории); у папки внутри чужого репозитория `inside` называет его, а git её не спрашивают — он ответил бы за внешний.

Лаборатория и версии

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

`lab create` строит десять учебных репозиториев в `examples\lab`; существующую папку меняет только с `--replace`, прежняя уходит в Корзину. `--register` вносит их в реестр (`lab_01` … `lab_10`).

`next-version` читает теги серии (`projection` по умолчанию) и отвечает `version` (`v0.1.2`), `tag` (`projection-v0.1.2`) и `command` — командой git, которой читал.

Remote

```
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` собирает адрес: `github`, `gitlab`, `codeberg` — облачные; `forgejo`, `gitea`, `custom` — со своим `--server` и портом SSH. `save` пишет remote в `config\remotes.json`; адрес с паролем или токеном внутри отклоняется.

Forgejo и 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` читает токен из stdin — одной строкой, не доводом командной строки, чтобы он не остался в истории команд и в журналах. Токен проверяется запросом `/api/v1/user` и уходит в хранилище учётных данных Git; в файлы программы он не пишется. `whoami`, `repos` и `create` берут его из хранилища; нет токена — `message: forgejo_token_missing`, код 1.

Области токена: `read:user` — кто я и список репозиториев, `write:user` — создать репозиторий, `read:repository` и `write:repository` — `git clone` и `git push` по HTTPS.