Audion Hub Manager — устройство

English · О программе · Консольная команда · Решения

Содержание

- Части
- Окно
- Канон Git
- Движок
- Реестр и слои
- Зеркало
- Профили
- Вход и Forgejo
- Хранилище
- Коммит отмеченных
- Лаборатория
- Проверки
- Сборщик
- Документы

Части

```
Start.exe ──► App\AudionHubManager.exe  (окно: C#, Avalonia, .NET 10 внутри папки)
                  |                                  |
                  | git — сам, по канону             | hub_cli.cmd — один JSON на
команду
                  ▼                                  ▼
          репозиторий выбранного слоя     system_core\cli.py → system_core\core\*
(Python 3.12 из runtime\)
```

Окно — лицо консольной команды: реестр, зеркало, проверки, пакеты, лаборатория и Forgejo идут через `hub_cli.cmd`, и то же самое делается без окна. Git окно зовёт само: показанная команда и есть урок, а состояние репозитория (status, граф, ветки) читается часто.

Окно

Где	Что
`Source\MainWindow.*.cs`	разделы: Changes, History, Branches, Sync, Mirror, Projects, Service; заголовок и слой — Project, колонка проектов — Side, коммит отмеченных — Marked, панель папок — Folders, терминал — Command, Run
`Source\Views\FadeBox.cs`	текст или строка, которые не помещаются, гаснут на последних 16 px вместо многоточия
`Source\Core\Git.cs`	тихое чтение для рисования: `status --porcelain=v2 -b`, `log --format`, `for-each-ref`
`Source\Core\Canon.cs`	канон из `hub_cli canon`: подписи, подсказки, риск
`Source\Core\Hub.cs`	вызов `hub_cli.cmd` и чтение его JSON; строка stdin (токен) уходит только в процесс
`Source\Core\Runner.cs`, `ChildJob.cs`	запуск команд с живым выводом; задание Windows: СТОП и закрытие окна гасят всё дерево процессов
`Source\Views\GraphCell.cs`	граф коммитов: полосы веток по родителям
`Source\App.axaml`	четыре темы и цвета Git: в индексе, изменён, новый, удалён, конфликт, переименован, вне профиля
`Source\Ps\`	PowerShell 7 программы, `App\AudionPs.exe` (Microsoft.PowerShell.SDK)

Каждая Git-команда кнопки идёт с окружением: `GIT_TERMINAL_PROMPT=0` (ничего не ждёт пароля), `GIT_EDITOR=true` и `GIT_SEQUENCE_EDITOR=true` (редактор, которого окно не покажет, не открывается), цвета — через `GIT_CONFIG_COUNT` (`color.ui=always`, `core.quotepath=false`), поэтому показанная команда остаётся той, что набирают руками. Сообщения git в потоке ошибок — не ошибка: исход говорит код выхода.

Опасные операции спрашивают с командой целиком, незакоммиченным и способом отмены. Набранная руками строка делится на вызовы Git — режим GIT или каждый `git` между `&&`, `||`, `;`, `|` и переводами строки в PS и CMD, — и каждый судится по доводам в любом порядке (`Source\Core\TypedCommand.cs`): `reset --hard` и прочие режимы `reset`, `commit --amend`, `restore` рабочей папки, `checkout --`, `switch -f`, `clean -f`, `push` с силой, удалением, зеркалом или refspec с `+` / `:`, `branch -D`, `stash drop`, `rebase`, `filter-branch`, `reflog expire`, `gc --prune`, `update-ref -d`, `rm` без `--cached`, `tag -d`. Что есть в каноне, спрашивает его словами, `reset --hard` — набранным RESET; рекурсивное удаление PS и CMD ловит фильтр слов.

Сборка: `install\Build-App.cmd` — тесты движка и канона, публикация PowerShell и окна (win-x64, вместе с .NET), затем самопроверка собранного окна (`--self-test --just-built`): консольная команда отвечает, канон читается и у каждой операции есть слова на двух языках, у темы есть цвета Git, номер exe совпадает с `config\version.json`. Сборка верит только строке `[OK] all checks passed` отчёта.

Канон Git

`system_core\core\git_canon.py` — 63 записи. Поля записи: `id`, команда с местами (`{path}`, `{commit}`, `{branch}`, `{remote}`, `{message}`, `{tag}`, `{paths}` — список), риск, режим, раздел окна, слова RU и EN «что делает» и «как отменить». Подпись кнопки (`short`) — команда без мест и тихих ключей: `git restore --staged`, `git switch -c`.

Риск		Режим	
`read`	ничего не меняет	`run`	сразу
`local`	меняет этот репозиторий	`ask`	после вопроса
`net`	говорит с сервером	`template`	в строку терминала, дописать самому
`danger`	может потерять работу		

Канон читают окно (`hub_cli canon`), тесты (`tests\test_git_canon.py` гоняет каждую запись на свежей копии лаборатории) и документы (`hub_cli canon --docs` пишет таблицы между `<!-- canon:… -->` и `<!-- /canon -->`; `tests\test_canon_docs.py` падает, если таблица отстала). Другой Git-команды у программы нет.

Движок

Модуль `system_core\core\`	Что
`hub_ops.py`	реестр и слои, проверка путей профилем, пакеты, workspace, адреса и `remotes.json`, лаборатория в реестре, инструменты машины, Корзина
`project_registry.py`, `project_importer.py`	чтение реестра; поиск корней проектов в папке и слияние с реестром
`projection_engine.py`, `scan_engine.py`, `mask_engine.py`, `copy_engine.py`	зеркало: обход по профилю, маски, копирование, суммы BLAKE3, отчёты
`docs_view.py`	слой DOCS: Markdown и тексты ИСХОДНИКА по профилю `markdown_only_projection`
`safety.py`	поиск секретов и тяжёлых файлов
`storage_layout.py`	корни хранилища и их роли
`git_engine.py`	`git status` для пакетов, теги версий серии, проверка адреса remote
`git_credentials.py`	протокол `git credential`: fill, approve, reject
`forgejo_api.py`, `forgejo_service.py`	API Forgejo и Gitea, вход токеном, репозитории
`auth_doctor.py`	проверка доступа: Git, имя и почта, gh, glab, SSH, адреса remote
`external_apps.py`	путь к VS Code этой машины
`output_decode.py`	раскодировка вывода (UTF-8, OEM, кириллица)
`lab.py`, `git_canon.py`, `canon_docs.py`	лаборатория, канон, таблицы канона в документах

Реестр и слои

`<корень проектов>\hub_projects.json` (`project_registry.registry_path`): у каждого корня свой реестр. `active_project_id` и `projects` — записи с `id`, `title`, `source_path`,

`projection_path`, `docs_path`, `profile`, `default_branch`, `notes`. Относительный путь считается от папки реестра; пути внутри корня программа пишет от него, остальные — полностью, и корень переезжает вместе со своим реестром. Запись без GIT-КОПИИ получает `Hub Data\<название>` рядом с программой — как при поиске. Испорченный файл не мешает запуску: уцелевшие записи читаются, о проблеме сказано в ответе (`load_error`). Реестр — данные этой машины, в git он не идёт. Если его нет, программа заводит его: из `config\projects.json`, если реестр лежит там (пути пересчитываются от папки программы к корню, старый файл удаляется); для корня поставки — из `config\projects.example.json`; для своего корня — пустым.

`projects export ФАЙЛ` пишет реестр в файл с полными путями; `projects import ФАЙЛ` пересчитывает пути файла от его места к корню и дописывает проекты, которых в реестре нет (`--replace` — список файла целиком).

У каждого слоя (`source`, `mirror`, `docs`) — состояние репозитория: `own` (свой), `parent` (папка внутри чужого: команды ушли бы туда), `none`.

Зеркало

```
обойти ИСХОДНИК по профилю → обойти GIT-КОПИЮ → сравнить пути
1 создать разрешённые папки
2 скопировать и обновить файлы
3 проставить времена изменения
4 если ошибок и расхождений нет — удалить лишнее в GIT-КОПИИ
5 убрать опустевшие папки
6 маркеры пустых папок — только если профиль велит (`preserve_empty_dirs`);
поставляемые профили их не ставят
```

ИСХОДНИК не меняется; `.git/**` защищён от обхода, удаления и обслуживания. План перепроверяется перед записью: устаревший или поправленный план не удалит ничего за пределами своей области. Ни один обход не идёт по символической ссылке или junction (`lstat`, метка reparse): такая папка или файл пропускаются, а каждая запись, копия и удаление раскрывают свой путь и отказывают тому, что ведёт за пределы своего слоя. Папка или файл, которые обход не смог прочитать, становятся пунктом `error` плана (`source_unreadable`, `mirror_unreadable`), и шаг 4 ничего не удаляет; сверка считает это ошибкой. Отчёты — `logs\<проект>\ГГГГММДД_ЧЧММСС_микросекунды_вид.json`.

Сравнение	Как
`quick`	размер и время
`safe`, `safe_blake3`	суммы — только при совпавшем размере и разном времени
`strict`, `strict_blake3`, `metadata_then_blake3`	суммы для всех файлов одного размера, даже при совпавшем времени; `metadata_then_blake3` — значение по умолчанию

Суммы помечены алгоритмом: `blake3:…`; `sha256:…` — запасной вариант, если BLAKE3 нет. Любое применение с ошибками или расхождениями возвращает код 1.

Профили

`config\projection_profiles.json`:

Профиль	Что
`audion_python_project_projection`	проект любого рода; 476 масок включения, 243 исключения, 89 папок вне обхода
`audion_python_project_projection_strict`	наследует проектный профиль (`extends`), сверка строгая
`markdown_only_projection`	только Markdown и тексты — слой DOCS

Профиль проекта пропускает исходники, документы и настройки распространённых языков и сборок: Python, веб (JS, TS, фреймворки), C и C++, .NET, JVM, Go, Rust, Swift, Dart и Flutter, Ruby, PHP, Elixir, Haskell, R, Julia, Lua, оболочки и PowerShell, SQL, Terraform, Nix, Docker и Kubernetes, CMake, Bazel, Meson, Qt (формы, ресурсы, QML), установщики WiX, Inno Setup и NSIS, переводы, шаблоны, схемы текстом; lock-файлы зависимостей, CI (GitHub, GitLab, Forgejo, Gitea и другие), настройки линтеров и форматтеров, общие `.vscode\tasks.json`, `launch.json`, `extensions.json`, `.devcontainer`; текстовые файлы Unity и Godot. Картинки и шрифты исходников проходят до мегабайта. Не пропускает зависимости и кэши (`node_modules`, `.venv`, `target`, `obj`, `.gradle`, `.next` и десятки других), сборки и пакеты, архивы и образы дисков, медиа и файлы графических программ, базы данных, наборы данных и обученные модели, журналы и временные файлы, `.env`, закрытые ключи, ключи и токены в текстовых файлах (`api_key_*.txt`, `token.json`), хранилища паролей, состояние Terraform, машинные `*.local.*` и `*.user`; у программ Audion — собранное окно `App\`, его состояние, пакет лицензий и манифест выпуска.

Над профилем стоит `.gitignore` самого проекта: что он прячет, зеркало не обходит и не копирует (`respect_gitignore`). Проект со своим репозиторием спрашивается как есть — закоммиченное не прячется никогда; любая другая папка — через временную пустую папку git, чтобы считался только её `.gitignore`, а не репозитория, внутри которого она лежит. Спрятанные папки первых двух уровней отсекаются до обхода: дерево сборки в `build-ui\` не обходится вовсе.

Сотни масок сверяются за доли секунды: точные имена — по набору, окончания вида `*.py` — одной проверкой окончания, остальные — одним общим выражением (`mask_engine._matcher`); ответ тот же, что у перебора масок по одной (`tests\test_mask_engine.py`).

Ключ профиля	Что делает
`dry_run_default: true`	применение без `--apply` только показывает
`require_include_filter: true`, `min_include_globs`	без масок планирование падает
`allow_unfiltered_full`	единственный способ разрешить полную копию; ни один поставляемый профиль его не ставит
`mirror_scope: "filtered"`	GIT-КОПИЯ приводится к ИСХОДНИКУ только по маскам
`compare_mode`	способ сравнения; поставляемые профили — `strict_blake3`
`small_include_globs`, `small_include_max_file_bytes`	маленькие файлы, которые проходят до своего предела (картинки и шрифты — до 1 МБ)
`hide_dirs`	папки, в которые обход не заходит
`exclude_dir_contents`	папки, которые остаются пустыми с маркером: служебные, сборки, выводы
`extends`	профиль берёт другой и меняет свои ключи
`preserve_empty_dirs`, `marker_file`	держать пустые папки маркером; в поставляемых профилях выключено — папки программы восстанавливает `install\init_folders.cmd`
`respect_gitignore`	не копировать спрятанное `.gitignore` проекта; по умолчанию включено

Ключ профиля	Что делает
`output_dirs`	папки-догадки: сборки и зависимости у большинства проектов, исходники у некоторых (`app\` Next.js, `build\` Electron, `bin\` сценариев, `vendor\` Go); не идут в копию, только если у проекта нет своего `.gitignore`
`extensionless_text_bytes`	файл без расширения, которого нет в списке (`install`, `UNLICENSE`, хук `.husky\pre-commit`), проходит, если он не больше предела и это текст: в первых 8 КБ нет нулевого байта

Правило папки — имя где угодно (`logs`), путь от корня (`install/download`) или имя только у корня с `/` впереди (`/App` — собранное окно, а не папка `app` с исходниками глубже). Имя может быть маской: `cmake-build-*`.

Тот же профиль решает, что идёт в коммит: `hub_cli check-paths` перед `add` и `commit`, файл вне профиля в индекс не кладётся.

Вход и Forgejo

Программа не хранит паролей, токенов и закрытых ключей. Вход — обычными средствами: ключи SSH и ssh-agent, Git Credential Manager, GitHub CLI, GitLab CLI. Проверки не интерактивны: `GIT_TERMINAL_PROMPT=0`, SSH с `BatchMode=yes`.

Свои серверы Forgejo и Gitea — `config\forgejo_hosts.json`: адрес, пользователь и порт SSH, вид адреса, логин; без токенов. Это данные машины, как реестр: в git файл не идёт, чистильщик выпуска его убирает. Учётная запись — личный токен доступа с заголовком `Authorization: token …`: он проверяется запросом `/api/v1/user` и отдаётся помощнику учётных данных Git (`git credential approve`); дальше `git push` по HTTPS берёт его сам. Вход OAuth2 не используется: в Forgejo у OAuth2 ещё нет областей, такой токен даёт права администратора учётной записи, живёт недолго и требует отдельного механизма для `git push`.

Хранилище

`config\storage_layout.json` — корни и их роли; `${APP_ROOT}` — папка программы:

Корень	Роль	Как поставляется
`manager_root`	сама программа	`${APP_ROOT}`

Корень	Роль	Как поставляется
`hub_data_root`	GIT-КОПИИ проектов	`${APP_ROOT}/../Hub Data`
`docs_root`	слой DOCS	`${APP_ROOT}/../Hub Docs`
`full_projects_root`	живые проекты, источник правды	`${APP_ROOT}/..`

`hub_cli storage` проверяет, что корни есть, ни один не лежит внутри другого, GIT-КОПИЯ каждого проекта — под `hub_data_root`, а DOCS — под `docs_root`. DOCS необязателен: корень DOCS нужен, только когда слой есть хотя бы у одного проекта. Демонстрационный проект и учебная лаборатория держат копии внутри программы и раскладкой не проверяются.

Корень живых проектов можно выбрать для одной машины: `hub_cli storage --projects-root ПАПКА` пишет его в `config\storage_layout.local.json` (рядом с программой — как `${APP_ROOT}/../…`, в другом месте — полностью; внутри GIT-КОПИЙ или DOCS — отказ), `--reset-projects-root` возвращает корень поставки. Файл не идёт в git, чистильщик выпуска его убирает (`*.local.json`): выпуск везёт раскладку программы, а не машины, где его собирали. Смена корня отвечает кодом 0, когда корень записан; что из проектов нового корня раскладка не держит, говорит `off_layout` (`source`, `projection`, `docs` — списки `id`), и окно называет их строкой под командой.

Коммит отмеченных

`Source\MainWindow.Marked.cs`: контрольная точка каждого отмеченного в колонке проекта по `SKILL.md` — `mirror apply --apply`, `mirror verify`, `safety --layer mirror`, `check-paths --layer mirror --stdin` по изменённым путям, `add` и `commit` канона в GIT-КОПИИ. Шаги идут через тот же исполнитель, что и кнопки; следующий шаг ставится, когда предыдущий закончен целиком (`Done` → `Dispatcher.Post`). Проект без своего репозитория в GIT-КОПИИ пропускается, отказ шага останавливает только свой проект, СТОП — всю цепочку.

Лаборатория

`system_core\core\lab.py` строит десять репозиториев со своими часами: автор, дата и байты каждого файла постоянны, поэтому хэши коммитов одинаковы на любой машине и в каждом прогоне. `_servers\` — bare-репозитории, «сервер» урока о remote.

Проверки

Что	Как
движок, консольная команда, канон	`runtime\python.exe -m pytest -q tests` — 172 теста; канон — на свежей копии лаборатории
окно	`install\Build-App.cmd` — самопроверка собранного окна
выпуск	`install\verify_portable_env.cmd` — окружение (`system_core\doctor.py`), консольная команда, `Start.exe`, окно по его отчёту

Сборщик

`builder_main.cmd` — меню на FZF:

Шаг	Что
01, 02	окружение Python: `runtime\` и `wheelhouse\` из `install\requirements_full.in` (CMD или PowerShell)
03	`system_core\fzf.exe`
05	пакет BLAKE3
06	папки программы
07	lock-файл зависимостей
09	установка без сети из `runtime\` и `wheelhouse\`
70	чистка кэша установки и того, что оставляет сборка окна
71	проверка выпуска
72	кодировка `.cmd`: UTF-8 без BOM, CRLF
95–98	открыть `install\`, `runtime\`, `wheelhouse\`, `licenses\`

Окно собирается отдельно: `install\Build-App.cmd`. `cleanup_project.cmd` возвращает проект к исходникам: убирает рантайм, кэши, журналы, состояние окна и то, что оставляет сборка; `App\` и `Start.exe` остаются.

Документы

Markdown — источник. Таблицы канона в документах пишет `hub_cli canon --docs`, руками их не правят. Слой DOCS не сверяется по BLAKE3: его техническая копия уже в GIT-КОПИИ.