📁 PROJECT LAYOUT & GLOBAL REQUIREMENTS *for Forum Backup*

1. Directory tree (root-level, no `src/` folder)

```
forum_mirror/          # Git / workspace root
├─ config/
│   ├─ __init__.py
│   ├─ defaults.yaml
│   └─ settings.py
├─ core/
│   ├─ __init__.py
│   ├─ fetcher.py
│   ├─ throttle.py
│   ├─ state.py
│   ├─ pathutils.py
│   ├─ redirects.py
│   └─ adblock.py
├─ crawler/
│   ├─ __init__.py
│   ├─ discover.py
│   └─ scheduler.py
├─ downloader/
│   ├─ __init__.py
│   ├─ assets.py
│   └─ workers.py
├─ processor/
│   ├─ __init__.py
│   ├─ orchestrator.py
│   └─ rewrite/
│       ├─ __init__.py
│       ├─ links.py
│       └─ assets.py
├─ cli/
│   ├─ __init__.py
│   ├─ __main__.py          # run via:  python -m cli
│   └─ auth.py
├─ utils/
│   ├─ __init__.py
│   ├─ files.py
│   └─ logging.py
├─ requirements.txt
└─ README.md
```

---

2. `requirements.txt`

```
aiohttp
beautifulsoup4
PyYAML
python-slugify
```

```
tqdm
ijson
```

---

3. Config package (`config/`)

3.1 `config/defaults.yaml`

Generated on first run then editable by user.

```
yaml

folder_mapping:
  f: categorias
  t: topicos
  u: users
  g: grupos
  admin: admin
  privmsg: privmsg
  profile: profile

ignored_prefixes: ["/admin", "/modcp", "/profile"]
blacklist_params: ["vote","mode","friend","foe","profil_tabs"]

max_asset_kb: null       # null = unlimited
slug_max_len: 120

ad_hosts: []             # merged with StevenBlack list
tracker_patterns: []

ad_sources:
  - url: https://raw.githubusercontent.com/StevenBlack/hosts/master/hosts
    cache_days: 7
```

3.2 `config/settings.py`

Function / Const	Purpose
`init(backup_root: Path, user_yaml: Path | None)`	
Globals produced	`FOLDER_MAPPING`, `IGNORED_PREFIXES`, `BLACKLIST_PARAMS`, `AD_HOSTS`, `TRACKER_PATTERNS`, `AD_SOURCES`, `MAX_ASSET_KB`, `SLUG_MAX_LEN`, `BACKUP_ROOT`

Imports: `Path` (pathlib), `yaml`, `os`.

---

4. Core package (`core/`)

Below: **public functions / classes** each file must expose, plus the key imports.

4.1 `core/fetcher.py`

Reusable aiohttp session with adaptive throttle.

Class	Methods	Notes
`Fetcher(cfg, throttle, cookies)`	`fetch_text(url, allow_redirects=True) -> (status:int, text:str	None, final_url:str) fetch_bytes(url) -> (status:int, bytes

Imports: `aiohttp`, `asyncio`, `ClientTimeout`, `TCPConnector`.

4.2 `core/throttle.py`

Adaptive delay & worker scaling.

Class	Purpose
`ThrottleController(cfg)`	Tracks RTT, 2xx streaks, 429/5xx; exposes `delay`, `workers`, and API: `before_request()` (async sleep) `after_response(status)` (update logic).

4.3 `core/state.py`

Two compact JSON databases + queue semantics.

Part	Public API	Key points

crawl_state	`add_url(path, rel_path)`
	`get_next(phase:str)` (`phase` = "discover" or "download")
	`mark_discovered` / `mark_downloaded` / `mark_redirect_source`
	`update_after_fetch(success, err="")`
	`pending_count()`
	`save()` (stream-write via `ijson`)
	`load()`
		Record format = `["rel_path", redirect_flag, status, retries, error]`.

assets_cache	`get_asset(url)` `add_asset(url, rel_path)`	Stored as `"url": "assets/…/file.ext"`.

Uses `asyncio.Lock`, temp-file + `os.replace()` for atomic writes.

4.4 `core/pathutils.py`

Map URL → desktop filepath.

Function	Description
`url_to_local_path(path:str) -> str`	*Lowercase* → `slugify()` tokens (max `SLUG_MAX_LEN`) → choose folder via `FOLDER_MAPPING` (fallback *misc*) → detect collisions (`-dupN`).

4.5 `core/redirects.py`

Redirect graph manager, persisted to `redirects.json`.

Class / Func	Purpose
`RedirectMap`	`add(src,dst)` (avoid self-loops) • `resolve(path)` (visited-set + depth≤16) • stats helpers.
`redirects`	Singleton instance used project-wide.

4.6 `core/adblock.py`

Download StevenBlack hosts & expose quick blocker.

Function	Purpose
`update_hosts(backup_root)` (async)	Download each source in `AD_SOURCES` if cache expired; merge hostnames into `config.settings.AD_HOSTS`.
`is_blocked_host(host:str) -> bool`	Membership test.

Requires `aiohttp`, `re`, `time`.

---

5. Utils package (baseline)

5.1 `utils/files.py` *(already outlined)*

`safe_file_write`, `safe_file_read` — atomic I/O.

5.2 `utils/logging.py`

`setup()` → basic coloured / timestamped logging (optional).

📁 6 · Package `crawler/`

6.1 `crawler/discover.py`

Element	Responsibility	Must import
`IGNORED_PREFIXES`, `BLACKLIST_PARAMS`, `BASE_URL`, `BASE_DOMAIN`	filter rules loaded from `config.settings`	`from config.settings import …`
`_strip_fragment(url)`	remove `#anchor` once and for all	`str.split("#",1)`
`_is_valid_link(href)`	**internal** filter: same domain, not ignored prefix, allowed	`urllib.parse.urlparse`, `urljoin`

Element	Responsibility	Must import
	query params, no `mailto:` etc.	
`_path_plus_query(url)`	return canonical key `/path?query`	…
`handle_redirect(worker_id, src_url, dst_url, state)`	log to `redirects`, mark source as redirect, enqueue dest	`core.redirects.redirects`, `core.state.State`
`class LinkDiscoverer`	one async worker	
`__init__(cfg,state,fetcher,worker_id)`	store refs + create `AssetManager` for prefetch (optional)	
`run()`	pop **l** URLs via `state.get_next("discover")`; terminate after 15×0.5 s empty loops	
`_process(path)`	1) fetch HTML (`allow_redirects=False`) 2) deal with 301/302 via `handle_redirect` 3) save raw HTML (`utils.files.safe_file_write`) 4) call `_parse_links()` then `state.mark_discovered(path)`	`url_to_local_path`, `safe_file_write`
`_parse_links(html)`	use `BeautifulSoup` to find `<a href>`; for each valid link → `state.add_url(key, rel_path)`; **returns count**	`bs4.BeautifulSoup`

Concurrency note: you may spawn `AssetManager.fetch()` as fire-and-forget tasks to pre-download images/CSS in discovery. They use the same asset cache so there is no duplication.

6.2 `crawler/scheduler.py`

API	Behaviour
`run_discovery_phase(cfg, state, fetcher)`	• start **1** `LinkDiscoverer`. • poll `state.pending_count()` every second; when pending ≥ 20 (or first worker finishes) spawn up to

API	Behaviour
	`cfg.workers` discoverers. • `await gather(*tasks)`; returns when all workers exit and **no `'1'` left**.
`run_download_phase(cfg, state, fetcher)`	• create `DownloadWorker` instances = `cfg.workers`. • `await gather()`. • returns when every worker exhausts `'d'` queue.

Imports required: `asyncio`, `LinkDiscoverer`, `DownloadWorker`.

📁 7 · Package `downloader/`

7.1 `downloader/assets.py` → `AssetManager`

Member	Purpose	Implementation hints

`__init__(fetcher, state)`	store references; mkdir folders:
	`BACKUP_ROOT/assets/imagens/internal`
	`BACKUP_ROOT/assets/files/internal`
	`BACKUP_ROOT/external_files`
		`pathlib.Path.mkdir(parents=True, exist_ok=True)`

`fetch(url:str, kind_hint:str)->str`	None`	main async method: 1. **Block check**: if host in `AD_HOSTS` → return None. 2. **Cache hit**: `state.get_asset(url)`. 3. **Download** via `fetcher.fetch_bytes`. 4. **Size guard**: if `MAX_ASSET_KB` and len > limit → return None. 5. **Extension** via `_choose_ext()`. 6. Write file atomically with `safe_file_write` (mode `"wb"`). 7. Add to asset cache; return relative path.
`_choose_ext(url, kind_hint)`	fallback logic when URL has no suffix; e.g. `"fonts"` → `.woff`	also consult `Content-Type` header if needed.

Constants: `IMAGE_EXTS` (png, jpg, …)

Imports: `hashlib`, `mimetypes`, `os`, `pathlib.Path`, `urllib.parse.urlparse`, `config.settings`, `core.state`, `utils.files`.

7.2 `downloader/workers.py` → `DownloadWorker`

Stage	What it does	Notes
`run()`	infinite loop: `path = await state.get_next("download")`; if None → break; else `_process(path)`	
`_process(path)`	1. Build absolute URL. 2. Fetch HTML **with redirects allowed**. 3. If final ≠ requested → `handle_redirect` and return. 4. On 200: `process_html(final_url, html, fetcher, state)`. 5. Save resulting HTML to `url_to_local_path(final_url)`. 6. `state.mark_downloaded(path)` and update `tqdm` progress (optional).	Catches exceptions → `state.update_after_fetch(False,"download error")`.

Imports: `urllib.parse.urljoin`, `traceback`, `processor.orchestrator`, `core.pathutils`.

📂 8 · Package `processor/`

(Only interfaces here; full code in Part 3.)

File	Public function / class
`rewrite/links.py`	`rewrite_links(soup, cur_file:str, state)`
`rewrite/assets.py`	`_rewrite_head(), _rewrite_body_assets()`
`orchestrator.py`	`process_html(url, html, fetcher, state) -> str`

📂 9 · Package `cli/`

File	Functions (new)	Description
`auth.py`	`handle_authentication()`	loads cookies → `is_logged_in()` → prompt wizard if needed.
`__main__.py`	See Part 3: interactive prompt + banners.	

📁 10 · Package `utils/`

File	Functions
`files.py`	`safe_file_write`, `safe_file_read` (atomic)
`logging.py`	`setup()` (optional colourful logging)

How everything connects (textual walk-through)

1. **CLI** asks for URL / backup folder; calls `config.settings.init()`.
2. Downloads StevenBlack hosts → merges into `AD_HOSTS`.
3. Builds `State`, `ThrottleController`, `Fetcher`.
4. **Phase 1** – `run_discovery_phase()`
   LinkDiscoverers: fetch raw HTML → save → extract links → queue them; prefetch assets.
   Ends when no `"l"` left.
5. **Phase 2** – `run_download_phase()`
   DownloadWorkers: pull `"d"` → fetch final HTML → `processor.orchestrator.process_html()` → rewrite assets & links → save final file → mark `"p"`.
   Ends when no `"d"` left.
6. Save state, copy to `crawl_state_final.json`, exit.

7 · Package processor/

7.1 `processor/rewrite/assets.py`

```
python

"""
Download & rewrite <head> and <body>-level external resources.

Public coroutines
-----------------
* _rewrite_head(soup, page_url, asset_mgr)
* _rewrite_body_assets(soup, page_url, asset_mgr)
```

```
Both modify the BeautifulSoup object **in-place** and never return HTML
text.
"""

from __future__ import annotations
import re, asyncio
from urllib.parse import urljoin, urlparse
from bs4 import BeautifulSoup

from downloader.assets import AssetManager
from core.adblock      import is_blocked_host
from config.settings   import BASE_URL, BASE_DOMAIN

# ─────────────────────────── helpers ────────────────────────────
CSS_URL_RE = re.compile(r"url\\(['\\"]?(.*?)['\\"]?\\)")
IMG_TAGS   = ("img", "input")       # tags that normally carry 'src'

async def _download_and_replace(tag, attr, mgr: AssetManager):
    url = urljoin(BASE_URL, tag[attr])
    rep = await mgr.fetch(url, "images")
    if rep:
        tag[attr] = rep

# ─────────────────────────── <head> ─────────────────────────────
async def _rewrite_head(soup: BeautifulSoup, page_url: str, mgr:
AssetManager):
    head = soup.head
    if not head:
        return
    # stylesheet / preload / icon
    for link in head.find_all("link", href=True):
        rels = {r.lower() for r in link.get("rel", [])}
        href = urljoin(BASE_URL, link["href"])
        host = urlparse(href).netloc.lower()
        if is_blocked_host(host):
            link.decompose()
            continue

        if "stylesheet" in rels:
            repl = await mgr.fetch(href, "css")
            if repl:
                link["href"] = repl

        elif rels & {"preload", "prefetch"}:
            repl = await mgr.fetch(href, "misc")
            if repl:
                link["href"] = repl

        elif "icon" in rels:
            repl = await mgr.fetch(href, "images")
            if repl:
                link["href"] = repl

    # external <script src="…">
    for script in head.find_all("script", src=True):
        src_abs = urljoin(BASE_URL, script["src"])
        if is_blocked_host(urlparse(src_abs).netloc.lower()):
            script.decompose()
            continue
        repl = await mgr.fetch(src_abs, "js")
        if repl:
```

```
            script["src"] = repl

    # inline <style> with font URLs
    for style in head.find_all("style"):
        if not style.string:
            continue
        css = style.string
        for orig in CSS_URL_RE.findall(css):
            abs_u = urljoin(page_url, orig)
            repl  = await mgr.fetch(abs_u, "fonts")
            if repl:
                css = css.replace(orig, repl)
        style.string.replace_with(css)

# ─────────────────────── <body> & inline ──────────────────────
async def _rewrite_body_assets(soup: BeautifulSoup, page_url: str, mgr:
AssetManager):
    # <img>, <input type="image">
    await asyncio.gather(*[
        _download_and_replace(tag, "src", mgr)
        for tag in soup.find_all(IMG_TAGS, src=True)
    ])

    # <script src> inside body
    await asyncio.gather(*[
        _download_and_replace(script, "src", mgr)
        for script in soup.find_all("script", src=True)
    ])

    # <source srcset="a.jpg 1x, b.jpg 2x">
    for src in soup.find_all("source", srcset=True):
        newset = []
        for part in src["srcset"].split(","):
            url = part.split()[0]
            repl = await mgr.fetch(urljoin(BASE_URL, url), "images")
            if repl:
                newset.append(part.replace(url, repl))
        if newset:
            src["srcset"] = ",".join(newset)

    # inline style="background:url(...)"
    for tag in soup.find_all(style=True):
        style = tag["style"]
        for orig in CSS_URL_RE.findall(style):
            repl = await mgr.fetch(urljoin(page_url, orig), "images")
            if repl:
                style = style.replace(orig, repl)
        tag["style"] = style
```

7.2 **processor/rewrite/links.py**

```
python

"""
Rewrite every internal <a href> so it points to the correct local file.
"""

import os
from urllib.parse import urljoin, urlparse
```

```
from bs4 import BeautifulSoup

from config.settings import BASE_URL, BASE_DOMAIN
from core.pathutils   import url_to_local_path
from core.redirects   import redirects
from core.state       import State, REL   # index 0 in the compact record

def rewrite_links(soup: BeautifulSoup, cur_file: str, state: State):
    cur_dir = os.path.dirname(cur_file)

    for a in soup.find_all("a", href=True):
        href = a["href"]
        if href.startswith(("mailto:", "javascript:", "#")):
            continue

        abs_url = urljoin(BASE_URL, href)
        base, *frag = abs_url.split("#", 1)
        netloc = urlparse(base).netloc
        if netloc and netloc != BASE_DOMAIN:
            # external link: leave unchanged
            continue

        key = urlparse(base).path + (f"?{urlparse(base).query}" if
urlparse(base).query else "")
        key = redirects.resolve(key)

        rec = state.urls.get(key)
        if not rec:
            continue
        target_file = rec[REL]
        rel_link = os.path.relpath(target_file, cur_dir).replace(os.sep,
"/")
        if rel_link.endswith("/index.html"):
            rel_link = rel_link[:-len("index.html")] or "./"
        if frag:
            rel_link += "#" + frag[0]
        a["href"] = rel_link
```

7.3 `processor/orchestrator.py`

```
python

"""
High-level coordinator that converts raw HTML into fully localised HTML.
Exported coroutine:  process_html(url, html, fetcher, state) -> str
"""

import asyncio, pathlib
from bs4 import BeautifulSoup

from downloader.assets            import AssetManager
from processor.rewrite.assets     import _rewrite_head,
_rewrite_body_assets
from processor.rewrite.links      import rewrite_links
from core.pathutils               import url_to_local_path
from core.state                   import State

async def process_html(page_url: str, html: str, fetcher, state: State) ->
str:
```

```
    """
    • Parse HTML with BeautifulSoup
    • Localise head assets, body assets
    • Rewrite internal anchors
    • (Future) Pass through optimiser hook
    """
    soup = BeautifulSoup(html, "html.parser")
    mgr  = AssetManager(fetcher, state)
    save_path = pathlib.Path(url_to_local_path(page_url))

    await _rewrite_head(soup, page_url, mgr)
    await _rewrite_body_assets(soup, page_url, mgr)
    rewrite_links(soup, str(save_path), state)

    return await maybe_optimize(soup)

# ─────────────────────── future hook ──────────────────────
async def maybe_optimize(soup: BeautifulSoup) -> str:
    """
    Stub for a future HTML minifier / template injector.
    For now, returns the str(soup) unchanged.
    """
    return str(soup)
```

---

8 · Package cli⁄

8.1 `cli/auth.py`

```
python

"""
Cookie-based authentication flow.
"""

import json, aiohttp
from pathlib import Path
from urllib.parse import urljoin
from bs4 import BeautifulSoup

from utils.files     import safe_file_write
from config.settings import BACKUP_ROOT

COOKIES_FILE = BACKUP_ROOT / "cookies.json"  # overwritten later at runtime

async def load_cookies() -> dict:
    if COOKIES_FILE.exists():
        return json.loads(COOKIES_FILE.read_text("utf-8"))
    return {}

async def is_logged_in(session: aiohttp.ClientSession, forum_url: str) ->
bool:
    r = await session.get(forum_url)
    html = await r.text()
    soup = BeautifulSoup(html, "html.parser")
    return any(urljoin(forum_url,
a["href"]).split("/",3)[3].startswith("profile")
               for a in soup.find_all("a", href=True))
```

```
async def handle_authentication() -> tuple[dict,bool]:
    """
    Returns (cookies, logged_in_flag).
    Prompts user to re-enter cookies if not authenticated.
    """
    import aiohttp, asyncio
    cookies = await load_cookies()
    async with aiohttp.ClientSession(cookies=cookies) as s:
        ok = await is_logged_in(s, BACKUP_ROOT.meta["forum_url"])
        if ok:
            print("☑ [Cookies] Logged in")
            return cookies, True

    ans = input("Reconfigure cookies? (y/N): ").lower()
    if ans != "y":
        return cookies, False

    # wizard
    domain = BACKUP_ROOT.meta["domain"]
    ck1 = f"fa_{domain.replace('.','_')}_data"
    ck2 = f"fa_{domain.replace('.','_')}_sid"
    print("Paste cookie values:")
    cookies = {ck1: input(f"{ck1}: ").strip(),
               ck2: input(f"{ck2}: ").strip()}
    await safe_file_write(COOKIES_FILE, json.dumps(cookies, indent=2), "w")
    return cookies, True
```

(CLI banner+prompt code was delivered in previous messages and should remain unchanged.)

---

9 · Package utils✓

9.1 `utils/logging.py`

```
python

import logging, sys
from datetime import datetime

def setup(level="INFO"):
    logging.basicConfig(
        level=getattr(logging, level),
        format="%(asctime)s | %(levelname)7s | %(message)s",
        datefmt="%H:%M:%S",
        handlers=[logging.StreamHandler(sys.stdout)]
    )
```

Call `utils.logging.setup("INFO")` inside `cli/__main__.py` if you want colour-free logging.

---

10 · `README.md` (starter)

````
# Forum-Mirror

Offline spider that clones phpBB / Forumotion boards with assets, link
rewriting,
ad-blocking and resume support.

## Quick start

```bash
git clone https://github.com/you/forum_mirror.git
cd forum_mirror
python -m pip install -r requirements.txt
python forum_mirror.py          # interactive wizard
````

By default the mirror lands in `~/Desktop/<forum-slug>/`.

Features

- **Two-phase crawl** – discovery + download with adaptive throttling.
- **Asset deduplication** – every image/font/CSS saved once.
- **Ad & tracker blocking** – integrates StevenBlack hosts list.
- **Resume safely** – state is saved every N pages.
- **Per-forum YAML** – customise folder mapping & filters.

11 · Testing package `tests/`

```
tests/
├─ __init__.py
├─ test_pathutils.py
├─ test_redirects.py
└─ conftest.py
```

11.1 `tests/conftest.py`

```
python

import pytest, tempfile, shutil, os
from pathlib import Path

@pytest.fixture
def tmp_root(tmp_path):
    """
    Create an isolated BACKUP_ROOT folder for each test.
    Auto-cleaned by pytest.
    """
    return tmp_path
```
```

11.2 `tests/test_pathutils.py`

```
python

from core.pathutils import url_to_local_path

def test_basic_mapping(tmp_root, monkeypatch):
    monkeypatch.setattr("config.settings.BACKUP_ROOT", tmp_root)
    out = url_to_local_path("/f17-something")
    assert out.endswith("categorias/f17-something.html")
```

11.3 `tests/test_redirects.py`

```
python

from core.redirects import RedirectMap

def test_chain_resolution():
    rm = RedirectMap(":memory:")           # in-RAM
    rm.map.update({"/a":"/b", "/b":"/c"})
    assert rm.resolve("/a") == "/c"
```

Run with `pytest -q`.

---

12 · Lint & Format

12.1 `pyproject.toml`

```
toml

[tool.black]
line-length = 88
target-version = ["py310"]

[tool.ruff]
line-length = 88
select = ["E", "F", "I"]           # errors, flakes, isort
ignore = ["E501"]                  # already handled by black
```

12.2 `.editorconfig`

```
ini

[*]
charset = utf-8
indent_style = space
indent_size = 4
end_of_line = lf
trim_trailing_whitespace = true
insert_final_newline = true
```

---

14 · Future HTML optimizer module

```
processor/
└─ optimize/
   ├─ __init__.py
   └─ htmlmin.py
```

14.1 `processor/optimize/htmlmin.py`

```
python

"""
Pluggable HTML minifier – currently no-op.

Swap `run()` with any real compressor (htmlmin, bs4 tweaks, etc.)
and orchestrator.maybe_optimize() will import & use it automatically
if the module exists.
"""

def run(html: str) -> str:
    # placeholder – return unchanged
    return html
```

Modify orchestrator hook:

```
python

async def maybe_optimize(soup):
    try:
        from processor.optimize.htmlmin import run
        return run(str(soup))
    except ModuleNotFoundError:
        return str(soup)
```

CI pipeline runs the same checks on every push.