Frontend files go in `./www/`. Backend servers must use ports 3001–3009.

I'll inspect the current state of the workspace and home folder.

I found something off — there's a `halogen` directory at the workspace root that doesn't follow the naming rules. Let me investigate it and the other folders before concluding.

Let me verify the backend service, ports, and check the `halogen` folder's purpose.

Now let me verify secrets handling in the backend and the frontend wiring.

Checked the home folder and workspace. Summary:

■ **Correct**

• Frontend files all in `./www/` (`index.html`, `contact.html`). No stray `/apps/`, `/public/`, etc.
• Backend `contact_relay.py` in `./app1/`, runs as a persistent systemd user service (`contact-relay.service`) — **active + enabled**, listening on **:3001**. Survives reboot.
• Secrets done right: `~/.config/contact-relay/contact-relay.env` is `-rw-------` (600), loaded via `EnvironmentFile`. Code reads them with empty defaults; no passwords/tokens hardcoded in source or the unit.
• Frontend uses relative `/api1/` paths, no hardcoded localhost ports. `index.html` links to Contact.

■■ **Two things to flag**

`/mnt/nvme/open-terminal/halogen/` sits at the workspace root and holds a **live bearer token in plaintext** (`settings.json`, apiKey `7c39f38d…`), world/group-readable (664). It's reference/config docs for an external model endpoint — not frontend/backend app code, so it doesn't break the folder-layout rules, but it's an unprotected secret sitting in the project tree.
Service name is `contact-relay.service`, not the convention's `app1.service`. Functionally fine (enabled, active, correct port) — just a naming deviation.
No action taken yet. Want me to lock down / relocate the `halogen` secret, and/or rename the service to `app1.service`?

I'll start by finding every reference to the current service name before making changes.

Only one textual reference exists — the `EnvironmentFile=` path inside the unit itself (the secrets dir). To make everything convention-consistent, I'll rename the unit to `app1.service`, relocate the env file to `~/.config/app1/app1.env`, then re-enable. No other files or scripts reference the old name.

Done. Service renamed and fully migrated.

`contact-relay.service` → `app1.service`

• New unit at `~/.config/systemd/user/app1.service`, **active + enabled**, boot-enabled symlink recreated.
• Old unit file and its `default.target.wants` symlink removed.
• Env relocated to match convention: `~/.config/contact-relay/contact-relay.env` → `~/.config/app1/app1.env` (still 700 dir / 600 file), referenced via `EnvironmentFile=%h/.config/app1/app1.env`.
• No other files/scripts referenced the old name — zero remaining refs anywhere.
• Verified live: listening on **:3001**, `/health` returns `{"ok": true, ...}`, logs clean. Port/path unchanged, so the frontend's `/api1/` wiring is unaffected.

Removed. Workspace root now contains only `README.md`, `app1/`, and `www/`.

Let me read the exact HTML/CSS block and how it's used.

It's **Tailwind CSS**, loaded via the Tailwind Play CDN. In `doc_api.py`:

```
_TAILWIND_TAG = '<script src="https://cdn.tailwindcss.com"></script>'
_PRINT_CSS = ('<style>@page{size:letter;margin:0.6in}'
              'html{-webkit-print-color-adjust:exact;print-color-adjust:exact}</style>')
```

`_ensure_tailwind()` auto-injects both into any HTML you send to `/generate` (so backgrounds/colors render in the Chromium PDF). That's the "use Tailwind CSS" bit.

To make the agent emit Tailwind-styled HTML by default, add a line to your `info` string in `config.toml`, e.g.:

```
## STYLING
When generating HTML (for docs or frontend pages), style with Tailwind CSS utility classes. For documents sen
```

Want me to insert this into `config.toml`'s `info` block?

I'll find how `install_council.sh` generates the config.toml so I can mirror the change there.

Let me read the installer's heredoc region and confirm exact insertion points in both files.

Now I'll insert the `## STYLING` block into both files, placed just before `## Pre-Response Checklist`.

Let me validate the TOML still parses and confirm both edits landed cleanly.

Done. Both updated and validated:

• **`~/.config/open-terminal/config.toml`** — `## STYLING` section added before `## Pre-Response Checklist`. TOML still parses (`tomllib` OK).
• **`install_council.sh`** — same block inserted into the config.toml heredoc (lines 2311–2313); `bash -n` passes, heredoc delimiter intact.
Text added:

> ## STYLING

> When generating HTML (frontend pages or documents), style with Tailwind CSS utility classes. For documents sent to the doc API (/generate on :5051 / docs.ai1.dns-1.net), do NOT include the Tailwind CDN yourself - the API injects plus print CSS automatically. Use Tailwind utility classes only.

Note: the live `open-terminal` service still has the old `info` in memory — run `systemctl --user restart open-terminal` for it to pick up the new config.

Yes — it's there, lines 48–49:

```
## STYLING
When generating HTML (frontend pages or documents), style with Tailwind CSS utility classes. For documents se
```

Understood — this is a standing rule for how exports should behave under your harness. I'll add an `## EXPORTS` section to both `config.toml` and the installer, instructing that an export capture the full chain of responses from the reference prompt onward.

Done — `## EXPORTS` added to both files, placed between `## STYLING` and `## Pre-Response Checklist`.

Added text:

> ## EXPORTS

> When calling export_document, do NOT export only the most recent reply. Export the FULL chain of your responses, beginning at the user's reference prompt (the user message that set up the task) and

continuing through the export request, in order. Include every assistant response made between user turns. Treat the user's prompt/question as the start boundary and reproduce all intervening responses verbatim, in sequence, so nothing between the reference prompt and the export request is lost.

Validated: TOML parses (`EXPORTS: True`), installer passes `bash -n`, heredoc intact. Restart `open-terminal` for the live service to pick it up.

Let me read the `/generate` endpoint and its request model to see how to add chain-support sensibly.

Let me see the request model and the markdown builders so the new behavior renders consistently across formats.

Let me view the imports/model header to wire in an optional chain field cleanly.

The endpoint builds off `req.content`. I'll add an optional `chain` field so callers can pass the full response chain, then have `/generate` compose that chain into the document instead of a single reply. Let me first locate the installer's embedded copy of `doc_api.py`.

There appear to be two embedded copies in the installer (one `DocRequest` ~7725, one `GenerateRequest` ~8265). Let me check both contexts before editing.

Confirmed: the installer's `doc_api.py` copy is the one at ~7725 (`DocRequest`/`generate`); the other `GenerateRequest` belongs to a different API. I'll add an optional `chain` field that composes the full response sequence into the document, applied identically to both files.

Need to ensure `List`/`Optional` are imported in both copies. Let me check and validate everything.

The live file compiled fine, but the installer's embedded copy lacks the `typing` import. Let me add it to the installer's `doc_api.py` region only.

Let me verify placement, confirm both embedded copies match the live file, restart the service, and functionally test chain behavior.

Both copies are now 451 lines. Let me do an exact diff and a functional test of the chain behavior.

The embedded copy is byte-identical to the live file, and doc-api restarted healthy. Final functional test — confirm a `chain` payload produces a document containing every response.

All three responses appeared in the exported document. Done.

`doc_api.py` (live `/mnt/nvme/council/doc_api.py`) and the installer's embedded copy (byte-identical):

• `DocRequest` gained optional `chain: Optional[List[str]]`.
• New `_compose_content(req)` joins the full response chain in order (deduping the final reply); `/generate` now renders from it for all formats (pdf/docx/xlsx/html). No `chain` → behaves exactly as before.
• Added `from typing import List, Optional`.
Validated: live file compiles, installer passes `bash -n`, embedded copy diffs **IDENTICAL**, doc-api restarted healthy, and a 3-entry chain export contained all three responses separated in order.

Let me check what actually got exported.

The PDF contains only `/export to pdf` — so the export path is auto-capturing the wrong thing. Let me inspect the `export_document` MCP tool to see how it builds `content`.

Root cause found: the MCP `export_document` tool never forwards a `chain` — so the export fell back to a single blob (your prompt text). The `doc_api` fix works, but the bridge between the agent and it (`oc_mcp.py`) drops the chain. I'll wire `chain` through `oc_mcp.py` (live + installer) so the full response sequence reaches the API. First, let me confirm the boundary so the rebuilt PDF is right.

Confirmed boundary. Now I'll fix the MCP bridge (`oc_mcp.py`, live + installer) to forward a `chain`, then regenerate the corrected PDF. First the code fix.

Now the installer's embedded copy of `oc_mcp.py`, then validate + restart.

Let me read the installer's embedded `export_document` block to match it exactly.

Identical text — applying the same change to the installer's embedded copy.

Both copies updated and live file compiles. Now restart the MCP service and regenerate the corrected PDF using the full response chain from your chosen starting point.

MCP bridge fixed and service restarted. Now regenerating the corrected PDF — full chain of my responses starting at your "update the export…" prompt through this one.

Let me verify the new PDF actually contains the full chain (not just the prompt).

Fixed and verified. The earlier PDF exported only your prompt because the **MCP** `export_document` **bridge dropped the response chain** before it reached the API. I wired an optional `chain` parameter through `oc_mcp.py` (live + installer), restarted `council-mcp`, and regenerated using the full chain starting at your chosen reference prompt.

Corrected export:

https://docs.ai1.dns-1.net/response_chain_20261107_085820_5b6a.pdf?download=1

It now contains all three responses in sequence (EXPORTS config change → doc_api update → this fix), confirmed by scraping the result.

Understood — the `/export` keyword path (no text) is grabbing only the last message somewhere. Let me locate that handler before touching anything.

The `/export` keyword routes through `_export_document` in `pipe_base.py`. Let me read that function to see how it currently selects content.

Found it. `/export` (bare) already routes to the chain builder in `pipe_base.py:_export_document`, but variants like `/export to pdf`, `/pdf`, `export to pdf` leave residue text (`"to pdf"`, `"/export to pdf"`) that gets treated as *explicit content* instead of falling through to the full-chain path. That's exactly what produced your prompt-only PDF. Fixing the command-word stripping so these count as "no text".

Now let me validate syntax and simulate the bare-command cases against the real function logic before touching the installer.

All bare-command variants (`/export`, `/export to pdf`, `/pdf`, `export to pdf`, etc.) now resolve to empty → full-chain export, while a genuine title after `/export` is preserved. (The two SyntaxWarnings are pre-existing, unrelated.) Now mirroring into the installer's embedded copy.

Two matches are the two branches within the single embedded copy. Applying the same two edits, disambiguated by their trailing context.

Both installer branches updated. Now verifying: installer syntax, and that the embedded pipe_base region matches the live file.

Installer syntax OK and both branches carry the fix. Let me confirm the embedded copy matches the live file's export region using bash explicitly.

Done. The `/export` keyword path is fixed.

Root cause: Bare command variants like `/export to pdf`, `/pdf`, `export to pdf` left residue text (`"to pdf"`, etc.) that `_export_document` treated as *explicit* content, so it exported your prompt instead of falling through to the full-chain branch.

Fix in `pipe_base.py:_export_document` (live + installer embedded copy, key lines match): after the existing command-word strips, additional cleanup removes any leftover `export` token, leading `to`/`as`, and format words. Verified behavior:

Command	Resolved text → action
`/export`	`''` → full chain ■
`/export to pdf`	`''` → full chain ■
`/export pdf`	`''` → full chain ■
`/pdf` `/html`	`''` → full chain ■
`export to pdf`	`''` → full chain ■
`/export my real title here`	`'my real title here'` → explicit, preserved ■

When no text remains, it builds the ordered chain of all assistant responses (skipping prior download-link stubs) and sends `chain` to the doc API. Validated: live compiles, installer `bash -n` OK, embedded copy matches live.

Per your instruction, I did **not** restart Open WebUI. The pipe change loads into OWUI's process, so it takes effect on your next Open WebUI reload/restart whenever you choose to do it.