NAS (HDD Browser via FastAPI) — Comprehensive, Code-Level Technical Report

The NAS project delivers a self-hosted web application that provides authenticated browsing, preview, and streaming of local or mounted storage. The backend is written in Python using FastAPI and Starlette building blocks, and it renders server-side HTML via Jinja2 templates while serving static JavaScript and CSS assets. It is designed to be especially friendly to Windows users by including a robust Windows batch launcher and optional integration with Tailscale for secure remote access. The project also prioritizes safe filesystem operations and multi-user role-based authorization without relying on a database; instead, it stores its users and configuration in a .env file, with functionality to read, validate, and even update that file at runtime from an admin panel.

At the top level of the repository, files like the README and a "NAS-Deep-Dive-Technical-Report.pdf" provide user-facing setup and overview. The real engine of the application lives under the hdd_browser/app directory. The FastAPI application is constructed in main.py, which mounts several sub-routers and provides all the page routes and most API endpoints. The security model is enforced both by a global authentication middleware and by explicit authentication checks on the protected endpoints. Static and templated pages give you a clean, straightforward UI for login, browsing filesystem roots, search, and an admin dashboard. The admin panel powers management of users stored in the .env file and offers quick statistical views of storage utilization and thumbnail cache usage, making it useful for ongoing operation and housekeeping.

The runtime configuration model is centered around the .env file that is auto-discovered and loaded, with Pydantic v2 used to define and validate the application settings. The code defends against weak defaults through validation and emits warnings for risky situations like an unchanged session secret. Many aspects of the behavior—allowed filesystem roots, whether uploads and deletions are permitted, whether thumbnails are generated, and more—are toggled through environment variables read from that .env. The implementation includes careful utilities to preserve JSON and Windows path escaping semantics inside .env content, avoiding subtle corruption when persisting configuration.

Because the UI emphasizes visual navigation, thumbnail generation is a first-class concern. Images are handled through Pillow, including EXIF-based orientation, and optional HEIC/HEIF support is wired up via pillow-heif. Video poster thumbnails are generated with ffmpeg when available. Thumbnails are cached on disk using content and parameter hashes so that repeated browsing is snappy while also reacting to file changes without confusion. Where thumbnails cannot be generated, a deliberately gray placeholder is returned to keep the UI responsive and predictable.

Below, we cover each main module in detail, explain the governance of authentication and authorization, and delve into the core file operations that make the system tick. We also unpack the launcher batch script and point out a Windows-only keyboard utility present in the module tree that is not part of the web app itself. Finally, we discuss the templates and static resources that compose the user interface and the images folder that contains screenshots and diagrams referenced in the README.

The code-level narrative is organized by key components and files, with links provided to the canonical source in the repository for easy cross-reference.

Application composition and top-level assets

At the repository root are several files that frame the project and provide a bridge into the application:

- The README is a deep, operational guide that walks you from cloning the repository to configuring .env, starting the server, and enabling optional remote access via Tailscale. It is thorough, including a full example .env. It also documents a set of API endpoints, many of which are implemented in the current codebase via the main application module and its routers. The README also shows screenshots and an architecture diagram. See the README here: README.md.
- The NAS.bat file is a Windows batch launcher. It is designed to run the system in a Conda environment and optionally integrate with Tailscale so that the browser automatically opens a .ts.net URL. The batch file also pre-creates a thumbnail cache directory, sets default environment values for the app, and checks for conda availability before launching uvicorn inside the chosen Conda virtual environment. From a developer-experience standpoint, this script significantly reduces startup friction on Windows. Its content is straightforward to read and adapt: NAS.bat.
- The serve-config.json file exists but is empty in the current repository state. It likely serves as a placeholder for future runtime configuration in JSON form, but the current implementation does not rely on it because the Pydantic settings model is strongly tied to environment variables and the .env file.
- The .thumb_cache directory exists to hold JPEG thumbnails that the application generates on demand to accelerate future navigations and keep front-end loading fast and predictable.
- The images directory contains static screenshots and diagrams referenced in the documentation, such as the login view, browse view, admin dashboard, and an architecture diagram. See the images (for example): login.png, browse.png, admin.png, architecture.png. There is also a one-byte login.html placeholder file under images, likely accidental or a stub.

The FastAPI application entrypoint and request lifecycle

The application is constructed in the hdd_browser/app/main.py module. The code begins by building a FastAPI app with a title sourced from the settings. It adds a Starlette SessionMiddleware with a secret key and cookie parameters and then registers Jinja2 templates and a static files mount. The session middleware is configured even though authentication in this codebase is governed by a signed cookie named hdd_session that is created and verified by code in the auth module. In that sense, the middleware's presence is more about aligning with the typical Starlette pattern and ensuring sessions are available if needed than it is about holding the authenticated user's identity. The source for main.py is here: main.py.

Templates are wired via a directory path pointing to hdd_browser/app/templates, and the static files mount exposes hdd_browser/app/static at the /static URL path, ensuring that CSS, JavaScript, and assets load for unauthenticated pages like the login screen. A small but useful construct installs a dynamic settings proxy into the template environment so templates always see up-to-date settings without requiring a server restart, which is helpful when settings are edited at runtime via the admin panel.

There are two middlewares that govern request handling. The first is an authentication gate. This gate checks whether a given path is public using security.is_public_path. If the path is public (for example, /auth/login, /static, or /favicon.ico), the request passes through. Otherwise, the middleware queries the current user by reading and verifying the hdd_session cookie via the auth module's current_user function. If no user is present, unauthenticated_response is invoked, which redirects the browser to /auth/login for normal page requests or returns a 401 JSON for /api/* endpoints. This approach gives a uniform protection mechanism to all protected endpoints without repeating authentication checks everywhere. The second middleware adds standard security headers such as X-Frame-Options and X-Content-Type-Options. There is a commented-out Content-Security-Policy header with a starter value that can be tightened for production if desired.

The page routes for HTML rendering are straightforward. The root path / redirects to the login if a user is not present and otherwise renders index.html. The /browse endpoint renders the browsing UI (browse.html) and, importantly, passes template flags indicating whether the current user can upload or delete. This is computed using the role-aware functions in main.py that check both the global enablement flags and the user's roles. The /search route renders the search UI (search.html). These page routes operate as the user-facing shell around the API endpoints described below.

The API endpoints in main.py are the heart of the browsing, previewing, downloading, streaming, searching, and file operations. The pattern is consistent: every API handler begins by requiring a user, computing the current allowed roots for that user, resolving the drive root associated with the drive_id, and then joining a relative path onto that root using file_ops.safe_join. The use of safe_join is critical because it normalizes and resolves the candidate path and then verifies it starts with the intended root, which prevents path traversal attacks that attempt to escape the allowed roots via sequences like ../. Only after this validation does the code perform the operation.

Drive discovery and listing utilize drive_discovery.discover_drives and resolve_drive_root. Searching is implemented with a recursive os.walk over the allowed tree and performs case-insensitive name token matching with early exit once a configured result cap is reached. Previewing a file checks MIME type and returns either text content (subject to maximum bytes) or metadata, allowing the front-end to decide how to present the file. The download path returns a FileResponse for the file, setting the filename for a better browser experience. Inline viewing is implemented by a dedicated /api/view endpoint that responds with an inline content disposition so browsers render compatible types (such as PDFs or images) inside iframe containers.

Large file streaming is handled by a first-class implementation of HTTP range requests in /api/stream. The main.py module defines a parse_range helper that accepts a Range header and parses single-range specifications in one of the standard forms—bytes=start-end, bytes=start-, and bytes=-suffix. It returns a validated (start, end) tuple within file bounds or None for invalid or multi-range requests. The handler then decides between a 206 Partial Content StreamingResponse with appropriate Content-Range and Content-Length headers or a full stream response if no valid Range header is present. The body producers are implemented as async generators that read the file in chunks and bail out early if the client disconnects, which is considerate both of server resources and of user experience when scrubbing video.

Deletion and upload have explicit checks tethered to configuration and roles. The delete endpoint checks that deletes are globally enabled, that the current user has admin or deleter in their role set if roles exist, and then refuses to delete the drive root for safety. It now supports recursive deletion via a flag, calling into file_ops.delete_path with recursive True to descend through directories. Upload is gated similarly: if global uploads are disabled, it returns a 403; otherwise it ensures the target directory exists and is a directory, then writes the uploaded file to disk and returns the saved path. Both routes rigorously validate the requested path against allowed roots, which is the essential protection that upholds the project's safety guarantees.

Thumbnail generation is exposed by /api/thumb. The request receives a drive, relative path, and a requested size. The code chooses an effective max dimension by favoring the provided size when in sensible bounds or falling back to the configured THUMB_MAX_DIM. It then calls thumbnailer.get_thumbnail, which in turn determines whether the path is an image, a video, or something else. If generation succeeds, the result is cached for future reuse and a "image/jpeg" payload is returned with Cache-Control headers encouraging the browser to cache the content for a week. If generation fails or the file type is unknown, a gray placeholder is synthesized with Pillow and returned, explicitly marked with an X-Thumb-Placeholder header so the frontend can differentiate a real preview from a fallback. This approach keeps browsing stable even when rare decoders are missing; users still see a uniform grid or table without broken images.

For HEIC and other image rendering scenarios, an /api/render_image endpoint can convert HEIC/HEIF to JPEG and optionally resize large images. It registers the HEIC opener from pillow-heif when available and respects a setting to disable HEIC conversion. It uses EXIF transpose to honor orientation and performs format conversions as needed to ensure the output matches what the UI expects. This allows in-browser displays without relying on clients to have niche codecs.

Finally, main.py includes a trivial /healthz endpoint that returns a JSON ok. By default it is public only if you edit the public path configuration; otherwise, it is protected and checked behind authentication. The module also defines a run() function that runs uvicorn with parameters from settings. This enables python -m execution or wiring from the batch file.

Authentication and session management

The authentication responsibilities live in hdd_browser/app/auth.py. The module creates an APIRouter at /auth and defines endpoints for GET /auth/login to render the login form, POST /auth/login to authenticate, and POST /auth/logout to clear a cookie and redirect back to the login page. When a user submits the login form, the handler first calls into env_utils.read_users_json to get the most up-to-date representation of users as stored in .env. If a match for username and password is found, it constructs a signed token using itsdangerous.URLSafeTimedSerializer, embedding the username, an issue time, optional roles, and optional allowed roots for that user. This token is written to a cookie named hdd_session with a max age of eight hours and the response redirects to the home page. If no valid USERS_JSON exists, the module offers a legacy fallback where it compares the submitted credentials against AUTH_USERNAME and AUTH_PASSWORD from settings, and on success grants an admin role and sets the cookie the same way. The cookie is HTTPOnly to protect against trivial client-side script access. The code that implements these flows is here: auth.py.

For each request, other modules can determine the current user by calling current_user(request), which reads and verifies this signed cookie. A user_info helper returns the unpacked payload as a convenient dict with normalized fields u for username, r for roles, and ar for per-user allowed roots. The require_user function is a gate that raises a 401 HTTPException if the user is missing, which the API endpoints use to simplify route logic. Because verification is time-bounded (through itsdangerous' max_age) and salted, the design remains robust without a backing user database; the cookie itself is authoritative for the session state.

The module delegates template rendering for the login form to Jinja2Templates. It passes the settings object into the template context so the login screen has access to data like the app name and debug mode flags. There is an anecdotal detail that logout simply deletes the cookie and redirects to the login page; this is appropriate here because there is no server-maintained session store that needs invalidation.

Public path governance and unauthenticated responses

It is useful to understand how the code decides which paths are public and what happens when a protected resource is requested unauthenticated. This policy is expressed in hdd_browser/app/security.py. The module defines a tuple of path prefixes that are considered public, including /auth/login and /static so that the login page's CSS and scripts are accessible. A small helper is_public_path(path) checks the incoming path against this list, explicitly marking the root path / as protected. If a request targets an API path beginning with /api and the user is not authenticated, unauthenticated_response returns a JSON 401 response with a small detail message. For page paths, it instead returns a RedirectResponse to the login page with a next query parameter set to the original target path, allowing the app to send users back to their intended destination after successful login. The code that implements these mechanisms is here: security.py.

While straightforward, this bifurcation of behavior is essential for clean user experience and for programmatic clients that should not be redirected when they are expecting JSON. The main middleware in main.py relies on these helpers to actually carry out the enforcement decision.

Configuration discovery, validation, and settings model

The configuration system in hdd_browser/app/config.py is both pragmatic and robust. It starts by discovering a .env file path. The discovery priority is explicit via an HDD_ENV_FILE environment variable; next it checks the current working directory for a .env; failing that, it walks up a few directories from the module's location looking for a .env, with a depth limit to avoid scanning the whole filesystem. The discovered path is stored and optionally logged once for visibility. If python-dotenv is available, it loads the .env with an override behavior controlled by HDD_ENV_OVERRIDE, which when set allows the .env values to override process environment variables. If python-dotenv is not installed, the code relies on Pydantic's env_file capability to read the file on Settings instantiation. This dual strategy ensures configuration is loaded consistently across environments and doesn't silently ignore the .env. The module logs warnings when a .env is not found or when python-dotenv is missing, giving the operator useful feedback.

The Settings model itself is declared with Pydantic v2 support in mind. It defines core parameters like APP_NAME, HOST, PORT, and DEBUG; legacy single-user authentication defaults for AUTH_USERNAME and AUTH_PASSWORD; a SESSION_SECRET with a validation rule that enforces a minimum length to avoid extremely weak secrets; and operational flags and parameters such as ENABLE_UPLOAD, ENABLE_DELETE, ENABLE_THUMBNAILS, ENABLE_HEIC_CONVERSION, ALLOWED_ROOTS, maximum preview bytes, search limits, and thumbnail parameters. There is an allowed_roots property that parses ALLOWED_ROOTS into a list of Path objects that exist and are directories, which is later used to filter drive exposure and path access. A verify method emits warnings when defaults are still present and could indicate insecure or unconfigured deployments, such as when AUTH_USERNAME and AUTH_PASSWORD remain "admin" and USERS_JSON is empty. The module exposes a get_settings function that is lru_cache'd, stabilizing repeated access to the settings object across requests while still allowing explicit cache clearing when runtime updates occur, such as after changing .env content via the admin panel. The implementation is here: config.py.

This configuration plumbing underpins the rest of the system. It governs which directories can be browsed, whether sensitive actions like deletion are available, and whether features like HEIC conversion or thumbnail generation are enabled. It also enforces the sine qua non of session security by requiring an adequately strong session secret. The template environment retains a dynamic reference to settings values so that templates always reflect the current truth—very useful given the admin panel can persist changes without restarting the app.

Environment utilities for persistent .env editing and USERS_JSON handling

The files hdd_browser/app/env_utils.py implement precise logic for reading and writing .env content and, critically, for working with a USERS_JSON variable that encodes users, roles, and per-user allowed roots. The design problem here is subtle: Windows paths use backslashes, and JSON strings require those backslashes to be escaped. Many helpers that read .env files unescape values, which can break the exact JSON string a user intended to store. To avoid this, the utilities eschew python-dotenv for reading USERS_JSON and instead parse .env lines manually to obtain literal values whenever necessary. For other keys, the code is happy to use python-dotenv if available to simplify reads and writes. The module also implements a small quoting strategy: when writing values to .env, it wraps them in single quotes and escapes internal single quotes so that the resulting file persists in a form that is both readable and robust across tooling.

The utils expose functions to determine the .env path, preferring the path discovered by the configuration module but falling back to a .env in the current directory if needed. The read_env_var function prefers process environment variables if set, then falls back to reading strictly from .env; whereas read_env_file_var_only reads only from the .env file, ignoring the process environment. This distinction matters for admin panel behavior, where one wants to reflect fresh on-disk state and avoid a situation where the process has stale environment variables masking the persisted configuration.

The USERS_JSON lifecycle is addressed via read_users_json and write_users_json. The reading function tries, in order, the value in the .env file, the process environment, and finally the current settings value and parses that JSON into a list of user dictionaries, normalizing roles to a list and roots to absolute, resolved path strings, skipping invalid entries. The writing function updates .env with a compact JSON serialization, mirrors the new value into os.environ to keep process-level reads consistent, and clears and refreshes the Pydantic settings cache so that get_settings returns up-to-date values. The module also defines a function that seeds a USERS_JSON value if none is present and legacy AUTH_USERNAME and AUTH_PASSWORD exist, thereby helping first-time setups transition into the preferred multi-user model without editing a JSON string by hand. The implementation details for this essential plumbing are here: env_utils.py.

The net effect is a persistent, file-based user store that is resilient to JSON escaping pitfalls on Windows and that can be safely manipulated from the admin UI. By intersecting per-user roots with global allowed roots in the main app logic, the design permits granular sharing while maintaining a global safety boundary.

Drive discovery, allowed root filtering, and path joining

Drive enumeration and mount resolution are done in hdd_browser/app/drive_discovery.py, which uses psutil.disk_partitions with all=True to include bind mounts, mapped network drives, and removable volumes—a key detail for a practical NAS-like browser on Windows and heterogeneous environments. The module optionally filters discovered mountpoints to only those that are within the configured allowed roots (or a per-request override of allowed roots supplied by main.py after consulting the user's roles and per-user roots). This filter uses path resolution and prefix checking to identify whether a mountpoint is within any allowed root. Where no partitions match because the allowed roots define arbitrary directories rather than real mountpoints, the code falls back to exposing the allowed roots themselves as drives, computing disk usage for those paths so that drive data like free and total space remain informative. A resolver function then validates drive IDs by comparing against discovered drives and returns a resolved Path for the chosen drive, raising a ValueError if the chosen ID is not recognized. This behavior ensures that the "drive list" a user sees and can select from is always consistent with their allowed roots. The code can be found here: drive_discovery.py.

The other half of safe filesystem operations is in hdd_browser/app/file_ops.py. The safe_join function is used across the API handlers to combine a validated root path and a user-supplied relative path and to reject results that would escape the root after resolution. Directory listing retrieves file metadata, including sizes, modified times, and best-effort MIME guesses for files. It sorts entries with directories first and then files lexicographically, which aligns with user expectations in most browsers and operating systems. A preview_file routine returns summary information for a file, including textual content when the file is small enough and text-like; to avoid blowing up the UI, it truncates and marks content when necessary and decodes with error resilience. The search function walks the tree up to a maximum depth and returns results with flags indicating whether an entry is a directory or file, allowing the client to present results consistently. For deletion, the code supports both a recursive mode that descends into directories (without following symlinks) and a safer non-recursive mode that only deletes empty directories. Uploads are saved via save_upload, which ensures that the resolved target remains within the target directory and avoids clobbering existing files by returning a conflict if the file already exists. The rigorous use of path resolution and prefix checking prevents directory traversal and other path-based attacks. Inspect these helpers here: file_ops.py.

Thumbnail generation and caching strategy

The thumbnailing subsystem in hdd_browser/app/thumbnailer.py begins with a clear distinction between image-like and video-like files using both file extensions and MIME type guesses. The effective ffmpeg path is determined either from a configuration variable or by searching the system PATH, making the system robust on environments where ffmpeg is installed in common locations. To identify a stable cache key for any input file, the code computes a SHA-256 hash over the file path, last modified nanoseconds, file size, the requested maximum dimension, and a marker indicating whether the file was an image, video, or something else. This combination ensures that when a file changes, prior cached thumbnails are no longer considered valid; at the same time, thumbnails at different sizes do not collide and risk mismatched thumbnails. The chosen cache directory is resolved from settings.THUMB_CACHE_DIR and created if missing.

For images, Pillow is used to open the source and transpose based on EXIF so images shot in various orientations display correctly. The image is then thumbnail-resized to fit within a square of max_dim and re-encoded as JPEG with quality tuning and RGB conversion when needed. For videos, the code uses ffmpeg to extract a frame one second into the file and applies a scaling filter that preserves aspect ratio while fitting the larger dimension into max_dim. The video thumbnail is also encoded as JPEG and returned. If any of these steps fail, a neutral gray placeholder square is synthesized and returned to avoid gaps in the UI. The code caches only genuine thumbnails; placeholders are not cached so that future requests can produce a real thumbnail once the environment problem is fixed (for example, after installing ffmpeg or enabling HEIC support). Examining this implementation will show a careful balance between performance, determinism, and graceful degradation: thumbnailer.py.

The module hdd_browser/app/heic_init.py complements thumbnailing and image rendering by attempting to register HEIC/HEIF support if pillow-heif is installed. It is idempotent and intentionally silent on failure so that endpoints can surface explicit errors when a user attempts to render a HEIC file but support is not available. This small module keeps imports localized and optional: heic_init.py.

Admin dashboard, configuration persistence, and storage statistics

The administrative functionality implemented in hdd_browser/app/admin.py is designed to be both practical and safe. The admin page at GET /admin renders an admin.html template only for users with the admin role. Admin APIs under /api/admin require admin privileges as enforced by a small helper that checks the decoded session token's roles set. One group of endpoints handles users: listing, creating, updating, and deleting users by persisting changes into USERS_JSON within .env. The list endpoint returns users with redacted password details so you can safely display this in the UI. The create and update endpoints validate role names against a shortlist and normalize per-user roots by resolving absolute paths and ensuring they lie within the global allowed roots when global restrictions are configured. Any errors yield clear HTTP errors for the UI to surface. After any write, the system updates .env and the in-process environment and then refreshes the settings cache so changes take effect immediately.

The admin panel also governs feature toggles for uploads, deletes, thumbnailing, and HEIC conversion. The update endpoint accepts form values for these booleans, parses truthiness with tolerant rules (accepting yes, on, true, and numeric 1), writes to the environment, and reads back the settings to return a current snapshot of features in the API response. This allows the UI to show the active state reliably even after a page reload.

Finally, the admin stats endpoint provides a time-boxed view into storage. It constructs a deduplicated list of allowed roots, checks each root's reachability quickly with a timeout to avoid hanging on offline mounts, and then performs limited scans to count files and directories and optionally compute a cumulative byte count with an explicit cap to prevent runaway scanning. It uses psutil to compute true device capacity, returning total, used, and free for each unique device backing the configured roots. It also computes the size of the thumbnail cache with a time budget to maintain UI responsiveness. The endpoint emits a partial flag to indicate when any of the scans were truncated to respect the time or count budgets, ensuring the UI can annotate approximate numbers. This time-aware reporting is a nice example of designing for responsiveness in administrative reporting. You can inspect the admin code here: admin.py and the capacity helper here: capacity.py.

Page templates and static assets

The UI templates are located in hdd_browser/app/templates and include base.html, index.html, login.html, browse.html, search.html, and admin.html. The base template anchors common layout, while the others render specific UI views. When rendering, the app passes both the current user and a live snapshot of settings, as well as permissions flags to conditionally expose actions like upload and delete. While we have not printed the inside of these templates here, their presence and naming conventions map directly to routes in main.py, which confirms the server-side rendering strategy. They are referenced as follows: base.html, index.html, login.html, browse.html, search.html, and admin.html.

Static assets are served from hdd_browser/app/static. The favicon is explicitly present and will be served even on the login page when not authenticated: favicon.ico. Subdirectories for CSS and JS exist. The README and module documentation in hdd_browser/Doucmentation.md indicate that the frontend logic is primarily vanilla JavaScript in a single app.js and that additional vendor libraries can be loaded lazily from CDNs for rendering DOCX and spreadsheets. The template and static directory organization aligns with common FastAPI + Jinja2 patterns and provides a predictable structure for adding or modifying frontend behavior.

API router in app/api.py and its relationship to the main application

There is a file hdd_browser/app/api.py which defines an APIRouter at the /api prefix and implements directory listing with pagination, simple uploads, renaming, a trash-and-restore workflow, zipping selections, and multi-file uploads that honor a relative path header. It addresses many of the same functional needs as main.py's API endpoints, but it is not imported or included in main.py in the current code. Moreover, some configuration references in api.py, such as settings.DRIVES, do not exist in the current Settings model, suggesting that this router represents an earlier or alternative approach not currently wired into the running application. From a maintenance perspective, it appears to be legacy or experimental code that was left in the repository for reference. If you decide to use it, you would need to add app.include_router(router) to the FastAPI setup in main.py and align its configuration model with the current settings system. You can review its content here: api.py.

Auxiliary Windows batch launcher and Tailscale integration

The Windows batch file at the repository root is more than a simple launcher. It prepares the environment by changing to the script's directory, sets a default conda environment name, and constructs an application command that invokes uvicorn to serve hdd_browser.app.main:app at a HOST and PORT defaulting to 0.0.0.0 and 8000 respectively. It defines a CUSTOM_URL variable that can point to a Tailscale-provided HTTPS URL including a next query parameter to land at the home route after authentication. Before running the app, the script ensures the thumbnail cache directory exists. It attempts to start the Tailscale Windows service and tray UI, tolerating the case where they are already running, and locates the Tailscale CLI across typical installation paths. If an environment variable TAILSCALE_AUTHKEY is present, it uses the CLI to bring up the node. It then probes the Tailscale IPv4 address a number of times within a loop, reporting status and gracefully skipping this step if it cannot acquire an IP in time.

The script chooses which URL to open in the default browser. If CUSTOM_URL is set, it will use that; otherwise, it defaults to a localhost URL using the configured port. It prints context lines such as the chosen conda environment and allowed roots to provide visible confirmation of key settings. Before executing the app command, it verifies that conda is available and instructs the user to use the Anaconda Prompt if not. Finally, it uses conda run -n "%ENV_NAME%" to run the app command in the chosen environment, avoiding global activation leakage and making the script idempotent for repeated use. After the server exits, it emits a message and pauses so the window stays open, allowing the user to read any final logs. This script is a model of user-friendly Windows integration: NAS.bat.

Optional HEIC and video dependencies and Python requirements

The Python dependencies are declared in hdd_browser/requirements.txt and pin precise versions of FastAPI, uvicorn, itsdangerous, psutil, jinja2, python-multipart, Pydantic v2 and its settings library, packaging, Pillow, and pillow-heif. The choice to pin versions makes deployments more deterministic and reduces variation across environments. FastAPI 0.111 with Pydantic v2 implies modern typing and configuration patterns and access to newer features. The presence of pillow-heif is optional because the code guards its import and enables conversion only when installed. You can review the requirements here: requirements.txt.

The templates and the user experience

Although we have not pasted the template HTML here, the files are named and sized in a way that strongly suggests a conventional structure. base.html probably provides the layout, navigation header, and footer scaffolding. login.html renders the username and password form and shows an error message when credentials are invalid. index.html likely acts as an entry point that redirects or presents a welcome panel linked to browse/search. browse.html likely includes a drive picker, a file list or grid that loads lazily via calls to /api/list and /api/thumb, and UI elements for preview, download, delete, and upload conditioned on permissions. search.html likely exposes a query input box, a drive selection, and a list view of hits. admin.html probably renders various cards showing statistics, feature toggle switches, and a user management table with forms to add or update users. The back-end's decision to send a fresh settings object into each render means toggles and user lists are always accurate on reload.

The static assets include a favicon and CSS/JS directories. The documentation in hdd_browser/Doucmentation.md explains that the front-end relies on vanilla JavaScript and may lazy-load vendor libraries like Mammoth.js and SheetJS from CDNs. This implies that the initial payloads remain lean and the UI loads additional capabilities only when needed, such as when previewing DOCX or spreadsheet files. The pages are server-rendered and progressively enhanced by JavaScript that fetches JSON from the API routes; this hybrid approach blends SEO-friendly templates with a responsive client experience.

The Windows keyboard utility script

There is a Python file hdd_browser/keyboard.py that is not directly used by the application but is included in the repository. This script is a Windows-only utility that manipulates PnP keyboard devices via PowerShell commands. It can list keyboard devices and enable or disable them based on substring matches in the device's InstanceId or FriendlyName. The script must be run as Administrator and uses commands like Get-PnpDevice and Disable-PnpDevice or Enable-PnpDevice. It acts as a useful example of invoking PowerShell from Python and parsing JSON output for device manipulation. Because it is not part of the core FastAPI application, you can consider it an auxiliary or experimental tool kept for convenience. The code is here: keyboard.py.

Project documentation within the repository

In addition to the root README, there is a Markdown document within the hdd_browser directory named "Doucmentation.md" that provides a comprehensive narrative of the project's architecture, flow, backend and frontend components, and configuration. It aligns with the code we have examined and reinforces how the system is meant to be understood and operated. Even small UX details are documented, such as the removal of a legacy "Open" action in favor of "Download" adjacent to "Preview" and "Delete." It provides further discussion of client-side viewers and the lazy-loading approach used on the front end. This document is a valuable complement to this report for developers diving into the code: Doucmentation.md.

Subtle implementation notes and security posture

There are a few subtle points in the code worth emphasizing because they reveal the project's defensive design. Path handling is carefully done with resolution and prefix checking, and file operations respect allowed roots both globally and per-user. The system federates per-user allowed roots with global allowed roots by intersecting them, which means that a user's view and permissions cannot widen the global scope; at most, they can narrow it. Delete operations refuse to remove the drive root and require explicit recursive intent to remove directories with contents. Uploads are only permitted when globally enabled and, if roles are present, when the user has an uploader or admin role. Session cookies are signed and time-bounded, and the session secret is validated for minimum length, discouraging default or trivial secrets. API routes return JSON 401s rather than redirects when unauthenticated, which is the correct behavior for programmatic clients. Security headers are added to all responses to raise the bar for common web security issues.

The thumbnail caching scheme's hash includes both size and mtime so it does not return stale thumbnails after a file change, which can be a subtle source of confusion in less careful implementations. By not caching placeholders, the system keeps a path open to producing real thumbnails once dependencies are corrected without waiting for manual cache purges. Where HEIC conversion is disabled or pillow-heif is absent, the render endpoints return clear errors, allowing the operator to respond by toggling the feature or installing missing components.

Lastly, although SessionMiddleware is installed, the actual user identity is conveyed in a manually set cookie using itsdangerous. This is a legitimate and secure pattern that avoids server-side session storage. A vestigial docstring in main.py mentions request.session["user"] but the code has clearly moved to the signed-cookie approach; the behavior aligns with the auth module's implementation and should be considered the single source of truth for authentication state.

Running and operating the application

From a practical perspective, the system can be launched directly by calling uvicorn hdd_browser.app.main:app and setting the HOST and PORT via environment variables or .env. Windows users can simply use NAS.bat with an appropriate Conda environment name. The .env can be created at hdd_browser/app/.env as suggested in the README, and the application will discover and load it. The admin panel allows you to manage users and quickly toggle features. If thumbnails or HEIC previews are not appearing, the admin stats and capacity reporting, along with the application logs and the dedicated troubleshooting section in the README, give actionable pointers. For secure remote access, you can use Tailscale to expose the app inside a private tailnet, and the batch script even opens the right URL if configured.

Because the design is explicitly local-first, if you wish to expose the app on a public network, you should place it behind an HTTPS reverse proxy, set SESSION_SECRET to a long random value, and consider disabling deletion or restricted roles. The security checklist in the README covers these concerns, and the admin feature toggles and per-user roots give you the tools to enforce the appropriate risk posture for your environment.

Notes on ancillary and legacy elements

The repository contains a small initializer file hdd_browser/*init*.py that simply marks the directory as a Python package. Its size suggests it may only contain a version string or be empty save for a docstring. There is also an empty serve-config.json at the root that is not consumed by the code in its present form and likely serves as a placeholder for future expansion of configuration mechanisms. The hdd_browser/app/api.py router appears to be legacy or alternative code that is not wired into the main app, and it references configuration not present in the current settings model. While it demonstrates additional operations like rename, trash, and zip workflows, its direct use would require integration work and should be treated as an example rather than a production path in the current codebase.

Conclusion

NAS is a cohesive, full-stack application built around a carefully implemented FastAPI backend and a straightforward server-rendered UI with progressive enhancement. It exhibits careful attention to safe path handling, consistent authentication enforcement, and sensible operational features like thumbnail caching and HEIC conversion. The configuration system is robust and supports runtime updates via the admin panel thanks to precise .env reading and writing utilities. The Windows batch script smooths the developer and operator experience, and optional Tailscale support gives a secure and convenient avenue for remote access.

From code clarity to operational ergonomics, the repository presents a well-thought-out solution for browsing and serving local storage through a web interface. It has a strong foundation for multi-user environments through per-user allowed roots and role-based permissions and exposes essential APIs and UI affordances for browsing, preview, download, streaming, upload, and deletion. Should you wish to extend the app toward new file types or new workflows, the modular organization of thumbnailing, file operations, and authentication will support those developments. Tightening security headers, adopting HTTPS everywhere in remote deployments, and continuing to pin dependencies will help maintain the project's strong security posture over time.

For continued development, keeping the documentation in sync with the current authentication mechanism and pruning unused legacy routers will prevent confusion and ensure that future contributors find the codebase as understandable as it is functional. The current structure, however, already supports a smooth mental model for how requests flow, how configuration is accessed and updated, and how UI and API layers collaborate to provide a responsive and secure browsing experience.