API Authentication & Permissions

Status: Alpha documentation snapshot (2026-03-12).

This note explains how Atho secures its HTTP API, how passwords/permissions work, and why this
matters for protecting your node and funds.

Model overview
- Every API request must include:
- X-API-Key ? the token from Src/Config/Api_Keys.json
- X-User ? username associated with the key (default atho)
- X-Pass ? password for the user (hashed in the key file)
- Optional HMAC: when enabled, requests also carry X-TS and X-Sig to sign method/path/body wit
a derived secret.
- Permissions are per-key; missing scopes yield 403s. Keys are stored in
Src/Config/Api_Keys.json.

Permissions
- Required scopes include:
- read ? general info endpoints
- explorer ? block/tx explorer-like data
- send_tx ? submit transactions
- mining ? mining control/status
- wallet_admin ? wallet/key mutation endpoints (/wallet/create, /wallet/recover_mnemonic,
/wallet/import, /wallet/export, rename/delete/default)
- node_admin ? node/config/security mutation endpoints
- Keys can have a subset; defaults are full permissions if a password is set, or read-only if
password is provided.
- The code migrates existing keys to include required scopes on startup.

Passwords and safety
- Passwords are hashed (SHA3-256 with a salt) in the key file; the plain password is never
stored.
- Without X-Pass, a key cannot authorize sensitive actions even if the token is known.
- If you leave the password blank on creation, the key is restricted to read-only scopes,
reducing blast radius for non-admin usage.

HMAC (optional but recommended when exposed)
- If REQUIRE_HMAC is enabled, every request must include X-TS and X-Sig; the server verifies
freshness (–60s) and integrity over method/path/body.
- HMAC secrets are derived per key (or you can store an explicit secret).
- Protects against replay and tampering; use when exposing APIs beyond localhost.

File locations
- API keys: Src/Config/Api_Keys.json
- Auto-creation: the entrypoint (docker-entrypoint.sh) calls ensure_api_key_auto() if no keys
exist; cliui/runnode call interactive setup.

Why this matters
- The API can submit transactions, manage mining, and expose node data. Without authentication
anyone could double-spend, drain wallets, or disrupt your node.
- Strong passwords + per-key scopes + optional HMAC provide layered defense. Keep Api_Keys.jso

private (chmod 600) and rotate keys periodically.
- API authentication protects request authorization. Release integrity checks (checksums/signatures) are a separate control layer and should be used for distributed binarie

Operational tips
- Local dev: HMAC off, HTTPS off, bind to localhost; use strong passwords anyway.
- Exposed/production: enable HMAC, run behind TLS/reverse proxy, limit IPs via firewall, use least-privilege keys (separate read-only vs. send/mine).
- Back up Api_Keys.json securely; losing it locks you out, leaking it compromises your node.

Performance behavior (wallet endpoints)
- POST /wallet/create now supports mnemonic-backed key creation parameters (word_count, passphrase, account, index, iterations) and defaults to 24-word mnemonic generation.
- POST /wallet/recover_mnemonic recovers deterministic keys from phrase+path.
- POST /wallet/export exports one key bundle to .txt.
- POST /wallet/import imports key bundles (.txt current format + legacy format support).
- API boundaries normalize sender/recipient addresses to canonical HPK before send-path logic runs.
- GET /wallet/balance uses a short in-memory cache and wallet snapshot path to reduce repeated index scans during GUI polling.
- POST /wallet/balance_batch accepts multiple addresses and returns one combined response so G wallet tables do one HTTP round-trip instead of N per-address calls.
- GET /wallet/history applies short TTL caching and per-request normalization memoization to reduce repeated work during rapid refresh loops.
- GET /wallet/recent supports cursor pagination (limit, offset, cursor) with bounded caps to avoid unbounded scans under load.

Network hashrate telemetry endpoints
Read-scoped endpoints now expose live and chart-ready hashrate telemetry:
- GET /network/hashrate/live
- returns current aggregate hashrate snapshot + per-reporter share
- GET /network/hashrate/chart?window_seconds=&bucket_seconds=&include_reporters=
- returns chart buckets from persisted history
- GET /network/hashrate/hourly?hours=&include_reporters=
- returns hourly historical buckets (default 168 hours / 7 days)
- points update hourly and include peer-shared reporter maps by default
- GET /network/hashrate/peer-shared-hourly?hours=&include_reporters=
- alias of /network/hashrate/hourly for dashboards expecting peer-shared naming
- GET /network/hashrate?window_seconds=&bucket_seconds=&include_reporters=
- combined live + chart response

Backing files (per network):
- logs/<network>/network/network_hashrate_live.json
- logs/<network>/network/network_hashrate_state.json
- logs/<network>/network/network_hashrate_history.jsonl

Network peer-count telemetry endpoint
- GET /network/node_count?active_window_seconds=&sample_limit=
- returns observed peer counts and sampled peer records from this node’s view
- includes observed_counts (known_peers, active_peers, healthy_peers, bucket stats)
- used by the GUI Web Explorer bridge/header stats to show network-facing peer estimates

Notes:
- This endpoint is still an observed/local network view, not global consensus truth.
- sample_limit is bounded server-side and intended for dashboard/reporting use.

Node/admin operations
- POST /network/switch
- set active network preference (mainnet|testnet|regnet)
- requires admin permission + sensitive approval token
- rejects while full/miner/wallet nodes are running (unless overridden)
- GET /nodes/active
- returns active process snapshot by role
- GET /storage/root
- returns effective storage-root info and persisted storage-path config
- POST /storage/root
- move/set block storage root; requires nodes stopped + admin approval token
- GET /emission/status
- reads emission tracker state (emission_state.json)
- GET /burn/status
- reads burn tracker state (burn_state.json)

Send idempotency (duplicate-payment protection)
- POST /tx/send now requires a client-provided intent_id (idempotency key).
- The node persists send intents in a local SQLite database:
- default path: logs/api/send_intents.sqlite3
- override path: ATHO_SEND_INTENTS_DB
- Intent lifecycle states:
- pending ? request accepted, send in progress
- broadcast ? transaction(s) built and submitted to mempool
- confirmed ? previously broadcast tx set later observed as confirmed
- failed ? send attempt failed
- If the same intent_id is submitted again with the same payload, the API returns the prior saved result (idempotent_replay=true) instead of creating new transactions.
- Reusing an intent_id with a different payload is rejected.
- GET /tx/intent/{intent_id} returns current intent state so clients can poll send progress without resubmitting the payment payload.