Authentication Architecture

QODY Authentication Architecture

Status: LIVE (demo + prod: 2026-06-30, commit e63ab88)
Revisions:
• Demo: qody-api--0000060 / qody-admin--0000033 / qody-staff-kitchen--0000016
• Prod: qody-api-prod--0000009 / qody-admin-prod--0000005 / qody-staff-kitchen-prod--0000004
MC: #104547 (cookie migration), #104553 (prod deploy), #104554 (rate-limiting)
Author: Securion (Parisa Tabriz) | FlowForge (Kelsey Hightower) deploy

---

Overview

QODY uses a **custom JWT authentication system** with httpOnly cookie transport, cryptographic CSRF protection, login rate-limiting, and optional TOTP 2FA step-up. The system deliberately **does NOT use Microsoft Entra** (unlike ALAI's other products like Bilko) — QODY is a self-contained BiH product requiring no Azure AD integration.

Authentication Stack:

- **JWT library:** auth0/java-jwt with HMAC256 signing
- **Password hashing:** argon2id (OWASP-recommended)
- **2FA:** TOTP (Time-based One-Time Password, RFC 6238)
- **Session transport:** httpOnly cookies (SameSite=Lax, Secure)
- **CSRF protection:** Signed double-submit cookie pattern
- **Rate-limiting:** Dual-key sliding-window (per email+IP and per IP)

Why NOT Entra: QODY targets BiH restaurants (Bosnia and Herzegovina market) where Azure/Microsoft SSO adds no value and creates unnecessary external dependencies. Self-contained JWT auth allows offline merchant onboarding, faster login flows, and complete control over session management. The architecture decision is documented in BookStack "QODY Architecture Decisions".

Login Rate-Limiting

QODY implements dual-key sliding-window rate limiting to protect staff and super-admin login endpoints against brute-force attacks and password spraying.

Implementation

Service: `LoginRateLimiter.kt` (in-memory, per-replica)

Thresholds:

- **5 failed attempts per (email + IP) in 15-minute window** — protects individual accounts
  - Resets on successful login
  - Prevents targeted account attack
- **20 failed attempts per IP in 15-minute window** — protects against spray attacks
  - Does NOT reset on success (spray defense needs longer persistence)
  - Prevents attacker trying many accounts from single IP

Response:

- Before threshold: HTTP 401 with vague message (no email-existence leak)
- After threshold: HTTP 429 Too Many Requests
  - Header: `Retry-After: <seconds>`
  - Body: `{"error":"Too many failed attempts","code":"RATE_LIMITED"}`

Endpoints Protected:

- `POST /staff/auth/login`
- `POST /superadmin/auth/login`

Known Limitation:
In-memory storage means limits are per-replica. For multi-replica deployments, the effective limit multiplies (5 attempts × N replicas). For true global limits, migrate to Redis (MC #104549 — same task covers WebSocket ticket store).

Live Verification

Curl test on https://api.qody.ba/staff/auth/login with invalid credentials (2026-06-30):

- Attempts 1-5: HTTP 401
- Attempt 6: HTTP 429 (threshold enforced)
- Subsequent attempts: HTTP 429 with Retry-After header

Evidence: `/Users/makinja/system/evidence/104554/prod-ratelimit-live-verify.txt`

Production Deployment

QODY authentication hardening (httpOnly cookie + CSRF + WebSocket ticket + rate-limiting) is now live on **production** and demo.

Demo Environment

Status: LIVE (2026-06-30, commit e63ab88)
Resource Group: rg-qody-demo (swedencentral)
Domains: demo.app.qody.ba | demo.admin.qody.ba | demo.kuhinja.qody.ba | demo.api.qody.ba
Cookie Domain: `.qody.ba`

Note: Demo topology changed on 2026-06-30 — `qody.alai.no` DECOMMISSIONED. All demo traffic migrated to `demo.*.qody.ba` with `COOKIE_DOMAIN=.qody.ba` (same as prod).

Production Environment

Status: LIVE (2026-06-30, commit e63ab88)
Resource Group: rg-qody-prod (gentlecliff-98883162, swedencentral)
Domains: app.qody.ba | admin.qody.ba | kuhinja.qody.ba | api.qody.ba
Cookie Domain: `.qody.ba`
Image Tag: 20260630-httponly-prod-104553

Revisions:

- qody-api-prod--0000009 (100%)
- qody-admin-prod--0000005 (100%)
- qody-staff-kitchen-prod--0000004 (100%)

Environment Variables:

- `COOKIE_DOMAIN=.qody.ba`
- `DEV_AUTH_MODE` — NOT SET (cookie-only mode)
- `CORS_ORIGINS=https://app.qody.ba,https://admin.qody.ba,https://kuhinja.qody.ba,...`

Production Verification Evidence

Health Check:

```
curl https://api.qody.ba/health
```

 PASS — RLS check: `bypassRls=false`, `status=PASS`

Frontends: All serving HTTP/2 200

- https://app.qody.ba
- https://admin.qody.ba
- https://kuhinja.qody.ba

Login Endpoint: Live and processing requests

- Rate-limiting confirmed (429 after 5 failed attempts)
- Application logs show proper auth flow
- CORS headers present for all origins

Rate-Limit Verification: Direct curl test on live prod:

- 5 sequential failed login attempts → HTTP 401
- 6th attempt → HTTP 429 + Retry-After header
- Evidence: `/Users/makinja/system/evidence/104554/prod-ratelimit-live-verify.txt`

Known Gap: Full Browser UAT on Production

Status: PENDING

The production database is clean (no demo seed). Super-admin account `asmirmc@gmail.com` exists and is active (role SUPERADMIN, not deleted — verified in prod DB 2026-08-02).
⚠ **CORRECTION (2026-08-02):** an earlier revision of this page stated the prod password was `PLACEHOLDER_PW`. That was true when written but was **remediated on 2026-07-06** — the literal placeholder was a live, guessable SUPERADMIN credential and has been rotated (argon2id via rehash-on-login; Bitwarden "QODY Prod Secrets" → `super_admin_password` updated). Do not treat the placeholder string as current.

What's Verified:

- Code deployed (same commit as demo)
- Health checks PASS
- Frontends serving 200
- Login endpoint live
- Rate-limiting active (429 confirmed)
- COOKIE_DOMAIN=.qody.ba set
- CORS includes all prod domains
- Demo browser UAT PASSED (Proveo 10/10 on demo.*.qody.ba with identical code)

What's Pending:

- ☐ Full browser login flow on prod (password rotated 2026-07-06; browser UAT still not run end-to-end)
- ☐ Playwright verification of Set-Cookie headers on prod

Equivalence Confidence:
Demo and prod run **identical code** (same commit e63ab88, same build args, same COOKIE_DOMAIN=.qody.ba). Demo passed full Proveo browser UAT (10/10 cross-role scenarios). Prod differs only in database content (clean vs seeded). Cookie flow proven on demo → high confidence in prod.

Next Step for Full Clearance:
Run Proveo browser UAT on prod → verify Set-Cookie headers contain `Domain=.qody.ba; HttpOnly; Secure; SameSite=Lax`. The password blocker is resolved (rotated 2026-07-06).

Known Gap: no self-service password reset for SUPERADMIN (open)

Status: OPEN — tracked as MC #106666 (H).

Verified live on 2026-08-02:

- `POST https://api.qody.ba/superadmin/auth/forgot-password` → **HTTP 404** (endpoint does not exist).
- `POST https://api.qody.ba/auth/forgot-password` without `venueSlug` → **HTTP 400** `{"error":"email and venueSlug are required"}`.
- The merchant flow resolves staff via `VenueTable.slug` → `StaffTable(email, venueId)`, so a venue-less SUPERADMIN can never be found by it. The admin UI "Zaboravili ste lozinku?" form (Slug lokala + Email) is therefore a dead end for super-admins.

Consequence: every forgotten SUPERADMIN password currently requires a manual `password_reset_tokens` INSERT against the prod database by an operator with `qody_flyway` access. Done once on 2026-08-02 (evidence: `~/system/evidence/106666/A-manual-reset-2026-08-02.md`).

Note for implementers: `PasswordResetService.resetPassword` is already venue-independent (token_hash → staff_id, `purpose='password_reset'`), and `https://admin.qody.ba/reset-password?token=…` renders correctly for a super-admin token. Only the *initiation* half needs building.

Cookie Transport

Cookie Attributes

QODY sets **two cookies** on successful login:

Cookie Name	HttpOnly	Secure	SameSite	Domain	Path	Max-Age
`qody_auth`	✅ Yes	✅ Yes	Lax	`.qody.alai.no` (demo) / `.qody.ba` (prod)	`/`	28800 (8 hours)
`qody_csrf`	❌ No	✅ Yes	Lax	Same as above	`/`	28800 (8 hours)

Environment-driven domain: The `COOKIE_DOMAIN` env var controls the `Domain` attribute:

- Demo: `COOKIE_DOMAIN=.qody.alai.no`
- Prod: `COOKIE_DOMAIN=.qody.ba`

Why SameSite=Lax (not Strict)

QODY's deployment topology:

```
Demo:  admin.qody.alai.no  → fetch →  api.qody.alai.no
       kuhinja.qody.alai.no → fetch →  api.qody.alai.no

Prod:  admin.qody.ba → fetch →  api.qody.ba
       kuhinja.qody.ba → fetch →  api.qody.ba
```

The frontend (admin MFE) and API are **different subdomains** but share the same eTLD+1 (`alai.no` for demo, `qody.ba` for prod). Under browser same-site rules, these are considered **same-site** despite the subdomain difference.

SameSite values comparison:

- `Strict`: Would block cookies on cross-site **top-level navigations** (e.g., clicking a link from email → qody.ba would not send cookie → forced re-login). Too restrictive for normal use.
- `Lax`: Allows cookies on same-site cross-subdomain fetches AND top-level GET navigations. **Blocks** cookies on cross-site POST/PUT/DELETE (where CSRF attacks originate). This is the sweet spot.
- `None`: Would send cookies on **all** cross-site requests, including from attacker origins. Requires additional CSRF layer (which we have, but defense-in-depth says use Lax).

Result: `SameSite=Lax` provides CSRF protection against external attackers while allowing natural user flows.

Cookie-Only Mode vs Dev Mode

Production/demo (cookie-only):

- Login endpoint returns `{"token": "", ...}` (empty string)
- JWT lives **only** in the `qody_auth` httpOnly cookie (JavaScript cannot read it)
- All API calls use `credentials: 'include'` to send cookies automatically

Local dev (bearer token fallback):

- Environment variable `DEV_AUTH_MODE=bearer` enables dual-write mode
- Login endpoint returns **both** the cookie AND `{"token": "eyJ...", ...}` in response body
- Frontend with `VITE_AUTH_MODE=bearer` reads the body token and sends `Authorization: Bearer` header
- Allows developers to run localhost without reverse-proxy/custom-domain setup

This is detected automatically — no developer config needed in production builds.

CSRF Protection

QODY uses the **Signed Double-Submit-Cookie** pattern (OWASP-recommended stateless CSRF defense).

How It Works

On login (server):

1. Issue JWT in `qody_auth` cookie (httpOnly)
2. Compute `csrf_token = HMAC-SHA256(JWT_SECRET, staffId + ":" + exp)` (hex-encoded, 64 chars)
3. Issue second cookie: `qody_csrf=<csrf_token>` (NOT httpOnly, so JavaScript can read it)

On every mutating request (POST/PUT/PATCH/DELETE) — server:

1. Read `qody_auth` cookie → validate JWT as usual
2. Read `X-QODY-CSRF-Token` request header
3. Re-derive `expected = HMAC-SHA256(JWT_SECRET, sub + ":" + exp)` from JWT claims
4. Constant-time compare `X-QODY-CSRF-Token` to `expected`
5. Return `403 CSRF_TOKEN_INVALID` if mismatch or header missing

GET and HEAD requests are exempt (must remain idempotent).

Frontend Requirements

- On login, read the `qody_csrf` cookie value (readable — no httpOnly)

- Store in memory (module-level variable, NOT localStorage)
- Attach `X-QODY-CSRF-Token: <value>` to every non-GET fetch in `apiFetch()` and `saFetch()`
- On logout, clear in-memory value (server clears both cookies via `Max-Age=0`)

CORS header allowlist: `X-QODY-CSRF-Token` is added to `Cors.kt` `allowHeader()` alongside `X-Step-Up-Token`.

Why This Pattern

- **Stateless:** No server-side session store required (QODY API is stateless, JWT-only)
- **HMAC-signed:** The CSRF token is cryptographically bound to the session; an attacker cannot forge it without knowing `JWT_SECRET`
- **Defense-in-depth:** SameSite=Lax blocks most CSRF, but if a browser has SameSite bugs, the CSRF token catches it

---

WebSocket Authentication

The Old Way (Removed)

Before MC #104547: WebSocket connections used `wss://api.qody.alai.no/staff/stream?token=<jwt>`.

Problem: The 8-hour JWT appeared in:

- Browser history (`wss://...?token=eyJ...`)
- Server access logs (query params are logged)
- Any intermediate proxy logs

This is a **credential leak** (the token survives browser restarts via history autocomplete).

The New Way (Ticket-Based)

Current flow (2026-06-30 deploy):

1. **Obtain a WebSocket ticket:**

```
POST /staff/auth/ws-ticket
Cookie: qody_auth=<jwt>
X-QODY-CSRF-Token: <csrf>
```

```
Response: {"ticket": "abc123...", "expiresAt": "..."}
```

2. **Connect WebSocket with the ticket:**

```
wss://api.qody.alai.no/staff/stream?ticket=abc123...
```

Ticket properties:

- **60-second TTL** (not 8 hours)
- **Single-use** (consumed on first connection, invalid after)
- **Stored in-process** (current implementation uses in-memory map — see "Known Follow-Ups" below)

Why this is secure:

- The ticket is short-lived (60s) — even if leaked via browser history, it expires fast
- It's single-use — replaying the URL doesn't work
- The long-lived JWT never appears in the URL

Why not just read the cookie on WebSocket upgrade?
The WebSocket API in browsers cannot send custom headers on the initial upgrade request. Cookies **are** sent, but the `WebSocket` constructor doesn't expose them. The `?ticket=` query param is the standard workaround (used by Socket.IO, etc.) — we just made the ticket ephemeral.

Authentication Endpoints

Staff & Admin Login

Endpoint	Role	Request Body	Response (Cookie-Only)	Response (Dev Mode)
`POST /staff/auth/login`	STAFF / OWNER	`{"venueSlug": "...", "email": "...", "password": "..."}`	`{"token": "", "staffId": "...", "name": "...", "role": "...", ...}` + Set-Cookie headers	Same JSON but `token` field contains JWT
`POST /superadmin/auth/login`	SUPER_ADMIN	`{"email": "...", "password": "..."}`	Same pattern	Same pattern

Cookies set:

- `qody_auth=<jwt>; HttpOnly; Secure; SameSite=Lax; Domain=<COOKIE_DOMAIN>; Path=/; Max-Age=28800`

- `qody_csrf=<hmac>; Secure; SameSite=Lax; Domain=<COOKIE_DOMAIN>; Path=/; Max-Age=28800`

Logout

Endpoint	Effect
`POST /staff/auth/logout`	Clears `qody_auth` and `qody_csrf` cookies (Max-Age=0)
`POST /superadmin/auth/logout`	Same as above

Frontend action on logout:

1. `POST /staff/auth/logout` (with `credentials: 'include'` so cookies are sent)
2. Clear in-memory CSRF token variable
3. Purge legacy localStorage keys (`qody_admin_token`, `qody_staff_token`, etc.) — these may persist from before the migration

WebSocket Ticket

Endpoint	Role	Response
`POST /staff/auth/ws-ticket`	STAFF / OWNER	`{"ticket": "abc123...", "expiresAt": "2026-06-30T22:00:00Z"}`

Then connect: `wss://api.qody.alai.no/staff/stream?ticket=abc123...`

Password Reset (Out of Band)

Endpoints:

- `POST /auth/forgot-password` — sends email with single-use token
- `POST /auth/reset-password` — validates token, sets new password

These are NOT cookie-based — the reset token is a URL parameter emailed to the user. Out of scope for this doc.

TOTP 2FA Step-Up Compatibility

QODY supports TOTP 2FA for high-privilege actions (e.g., refunds, merchant suspend).

The JWT-to-cookie migration does NOT affect TOTP. Verification:

1. The TOTP flow issues a **step-up nonce** (`stepUpToken`), not a new session JWT.

2. This nonce travels in the `X-Step-Up-Token` request header (already in CORS `allowHeader`).
3. The TOTP code itself travels in the POST body of `/admin/mfa/step-up` (`{"code": "NNNNNN"}`).
4. The session JWT (now in `qody_auth` cookie) continues to satisfy the `authenticate("staff-jwt")` guard on all TOTP routes.

No change needed to the TOTP flow logic — only the JWT **extraction point** changed (from `Authorization` header to `qody_auth` cookie).

Recovery codes also travel in POST body. Unchanged.

CORS Configuration

Moving to cookie-based auth requires `credentials: 'include'` on all frontend fetch calls. Browsers enforce:

- `Access-Control-Allow-Credentials: true` (already set in `Cors.kt`)
- `Access-Control-Allow-Origin` **MUST NOT be** `*` (wildcard rejected when credentials are included)

QODY CORS_ORIGINS (explicit allowlist):

Demo environment:

```
https://qody.alai.no,https://admin.qody.alai.no,https://kuhinja.qody.alai.no,https://demo.app.qody.ba,https://demo.admin.qody.ba,https://demo.kuhinja.qody.ba
```

Prod environment:

```
https://app.qody.ba,https://admin.qody.ba,https://kuhinja.qody.ba
```

Local dev (hard-coded in Cors.kt):

```
http://localhost:5173, http://localhost:5174, http://localhost:5175
```

Critical: The `CORS_ORIGINS` env var is populated in `DEPLOY-MAP.md` secrets table and set in Azure Container Apps configuration. The code reads this env var and splits on comma. No wildcard fallback.

Critical Ops Lesson: Edge Cookie Incident

Cross-reference: `technical_bilko_entra_edge_cookie` (BookStack / MEMORY.md)

Bilko failure pattern (2026-06-23): Bilko's edge middleware checked for a `refresh-cookie` **or** an `Authorization` header to detect auth state. When Bilko switched to cookie-only auth, the middleware didn't know about the new cookie name → treated all requests as unauthenticated → redirect loop.

QODY guard requirement:
Any future route guard, middleware, reverse proxy rule, or Ktor plugin that checks "is this request authenticated?" **MUST read the `qody_auth` cookie by name.** It MUST NOT:

- Inspect `Authorization: Bearer` as primary path (frontend no longer sends this in production)
- Inspect `qody_admin_token` from localStorage (server cannot see localStorage)
- Use any other cookie name

Specific guard locations in QODY:

1. `authenticate("staff-jwt")` plugin in `Application.configureSecurity()` — token extraction changed from `Authorization` header to `qody_auth` cookie (with local-dev `Bearer` fallback)
2. `verifyStaffJwt()` in WebSocket handler — now validates the `?ticket=` param (which is itself issued via a cookie-authenticated endpoint)
3. `requireStepUp()` in TOTP routes — reads `X-Step-Up-Token` header (not session cookie; no change needed)

If you add a new middleware/ingress rule: it MUST check `qody_auth` cookie. Otherwise, all users will appear unauthenticated.

Known Follow-Ups

1. WebSocket Ticket Store (MC #104549)

Current state: The WS ticket store is **in-process** (ConcurrentHashMap in `RealtimeHub.kt`).

Problem: If the API scales to multiple replicas (horizontal scaling), a ticket issued by replica A won't be visible to replica B → connection fails.

Solution: Move ticket store to **Redis** (shared state across replicas). TTL handled by Redis `SETEX` (60s auto-expiry). Single-use handled by Redis `GET` + `DEL` atomic operation.

Impact: Non-blocking. Single-replica demo works fine. Defer until multi-replica deployment.

2. CORS Origin List (Dynamic Management)

Current state: CORS origins are hardcoded in `CORS_ORIGINS` env var.

Future improvement: Store allowed origins in the database (per-venue or system-wide config table). This allows adding new custom domains (e.g., `order.my-restaurant.ba`) without redeploying the API.

Impact: Low priority. Current env-var approach works for all known domains.

Recurring Build-Arg Trap (Cross-Reference)

CRITICAL for MFE rebuilds: All three MFEs (guest, admin, staff-kitchen) **MUST** receive `VITE_API_BASE_URL` at Docker build time. Vite bakes env vars into the bundle.

Symptom of missing build-arg:

- Deploy succeeds (200 OK)
- Page loads fine
- API calls fail (browser tries `localhost:8080` or `undefined`)

Evidence of correct build:

```
===== QODY <MFE> BUILD: API base resolves to https://api.qody.alai.no =====
```

How to verify live:
Open browser console → check network tab → API calls should go to `api.qody.alai.no`, NOT `localhost`.

This has broken QODY 3 times (see MEMORY.md "Talas 2B", "Talas 4", "UAT fix-wave A"). The fix is durable in the Dockerfile (build-arg default), but **always verify** the build log when deploying MFEs.

Security Properties After Implementation

Threat	Before (localStorage + Bearer)	After (httpOnly cookie)
XSS exfiltrates session JWT	❌ Trivial — JWT readable via `localStorage.getItem()`	✅ Eliminated — httpOnly cookie not readable by JS
CSRF (state-mutating requests)	✅ Not a concern (Bearer token not auto-sent)	✅ Mitigated by SameSite=Lax + HMAC-signed double-submit cookie
XSS injects CSRF token	N/A	✅ Mitigated — CSRF token is HMAC-signed with server secret; forgery requires `JWT_SECRET`
Cookie stolen via non-HTTPS	✅ N/A (already HTTPS-only)	✅ Mitigated — Secure flag, HSTS deployed on Azure ACA
Cookie sent to wrong subdomain	N/A	✅ Scoped to `Domain=.qody.alai.no` or `.qody.ba` — not sent to unrelated ALAI products
JWT exposed in WS URL	❌ Present (8-hour token in `?token=` query param)	✅ Eliminated — 60s single-use ticket instead
Session fixation	✅ Not applicable (JWT stateless)	✅ Not applicable (JWT stateless)

Net result: The httpOnly cookie migration **eliminates** the XSS credential exfiltration vector (the primary threat) while maintaining CSRF protection via defense-in-depth (SameSite + HMAC).

Non-Goals (Out of Scope)

- **Refresh tokens:** Out of scope. QODY uses 8-hour access tokens (matches a restaurant shift). Refresh token rotation would require server-side token store. Deferred.
- **Guest MFE (`qody-guest`):** Guest flow is anonymous (QR-token scoped, no login). This spec covers only authenticated staff/admin flows.
- **Password reset tokens:** Single-use URL tokens emailed to user. Do not use cookies. Out of scope.
- **OAuth / Social Login:** QODY is BiH-market only, no Google/Facebook login. Out of scope.

Live Verification (2026-06-30 Deploy Evidence)

From: `/Users/makinja/system/evidence/104547/flowforge-deploy-evidence-20260630.txt`

Deployment Metadata

- **Branch:** `feat/qody-httponly-cookie-104547`
- **Commit:** `b15efa2`
- **Environment:** `rg-qody-demo` (swedencentral, Azure)
- **Images:** `qodydemoacr.azurecr.io/qody-{api,admin,staff-kitchen}:20260630-httponly-104547`
- **Revisions:**
  - `qody-api--0000058` (100% traffic)
  - `qody-admin--0000031` (100% traffic)
  - `qody-staff-kitchen--0000014` (100% traffic)
  - `qody-guest--0000032` (unchanged, not rebuilt)

Environment Configuration (qody-api)

```
COOKIE_DOMAIN=.qody.alai.no
CORS_ORIGINS=https://qody.alai.no,https://admin.qody.alai.no,https://kuhinja.qody.alai.no,https://demo.app.qody.ba,https://demo.admin.qody.ba,https://demo.kuhinja.qody.ba
DEV_AUTH_MODE=(not set — cookie-only mode active)
```

Verification Tests

1. Frontend endpoints (200 OK):

```
 https://qody.alai.no → HTTP/2 200
 https://admin.qody.alai.no → HTTP/2 200
 https://kuhinja.qody.alai.no → HTTP/2 200
 https://demo.app.qody.ba → HTTP/2 200
 https://demo.admin.qody.ba → HTTP/2 200
 https://demo.kuhinja.qody.ba → HTTP/2 200
```

2. API health check:

```
 https://api.qody.alai.no/health → HTTP/2 200
{
  "status": "ok",
  "version": "0.1.0",
  "db": {
    "connected": true,
```

```
    "rlsRoleCheck": {
      "role": "qody_app",
      "bypassRls": false,
      "status": "PASS"
    }
  }
}
```

3. HttpOnly cookie auth verification:

```
POST /staff/auth/login
Body: {"venueSlug":"qody-demo","email":"owner@demo.qody","password":"demo1234"}

Response Headers:
set-cookie: qody_auth=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...; HttpOnly; Secure; SameSite=Lax;
Domain=.qody.alai.no; Path=/; Max-Age=28800
set-cookie: qody_csrf=14b7a79615f0f46a51a5c38753b306be6fb3b0301ef256237d6971c73bbd7c16;
Secure; SameSite=Lax; Domain=.qody.alai.no; Path=/; Max-Age=28800

Response Body:
{"token":"","staffId":"00000000-0000-0000-0000-000000000080","name":"Demo
Owner","role":"OWNER","venueId":"00000000-0000-0000-0000-000000000002","venueName":"QODY Demo
Bistro"}
```

 VERIFIED:

- `Set-Cookie` header contains `qody_auth` with `HttpOnly`, `Secure`, `SameSite=Lax`, `Domain=.qody.alai.no`
- `Set-Cookie` header contains `qody_csrf` with `Secure`, `SameSite=Lax`, `Domain=.qody.alai.no` (no HttpOnly)
- Response JSON `"token"` field is **EMPTY** (`""`) — no bearer token in body

4. CSRF protection verification:

```
PATCH /admin/settings
Cookie: qody_auth=<valid jwt>
(WITHOUT X-QODY-CSRF-Token header)

Response: HTTP/2 403
{"error":"CSRF token required","code":"CSRF_TOKEN_INVALID"}
```

 VERIFIED: Mutation endpoint correctly rejects requests without CSRF token.

Summary

QODY authentication is production-grade httpOnly-cookie-based JWT auth:

- Session credential not readable by JavaScript (XSS-proof)
- CSRF protected by SameSite=Lax + HMAC-signed double-submit cookie
- WebSocket auth uses ephemeral 60s single-use tickets (no JWT in URL)
- TOTP 2FA step-up compatible (unchanged)
- CORS properly configured (explicit origin allowlist, credentials: true)
- Live on demo (2026-06-30, revisions verified)
- Edge-cookie-trap lesson integrated (guards check `qody_auth` cookie by name)

Remaining work:

- MC #104549: Move WS ticket store to Redis (multi-replica support)
- Proveo validation (MC #104547 gates)

Documentation sources:

- Design spec: `/Users/makinja/business/ALAI-Holding-AS/products/qody/docs/auth-cookie-spec.md` (Securion)
- Deploy evidence: `/Users/makinja/system/evidence/104547/flowforge-deploy-evidence-20260630.txt` (FlowForge)
- Live verification: curl + browser UAT on https://api.qody.alai.no

---

Last updated: 2026-06-30
Next review: Before production cutover (qody.ba merchant onboarding)

Revision #3
Created 2026-06-30 19:45:24 UTC by John
Updated 2026-08-02 09:14:50 UTC by John