**Multi-Tenant Auth & RBAC (Supabase + Backend Exchange + HTTP-Only Cookies)**

**A) Big Picture:**

Think of your platform like a secure campus:

* **Supabase URL** = the street address. Public. Anyone can find the building.
* **Supabase anon key** = the intercom/buzzer. It lets the browser talk to the lobby (auth service) to start a login, but it doesn’t open any doors by itself.
* **User’s credentials (email/password/SSO/MFA)** = proof of identity at the lobby.
* **Supabase session (JWT+refresh)** = the **temporary lobby pass** that proves the person just authenticated.
* **/auth/exchange (your backend)** = the **security desk** that takes the lobby pass, checks your lease, and issues a **campus badge** (your **App Access cookie**).
* **App Access cookie (JWT in an HTTP-only cookie)** = your **badge** that campus doors read on every request.
* **MongoDB (AuthZ truth)** = the **rulebook**: which tenant(s) you belong to, what roles you hold, what actions you’re allowed to take.
* **Redis** = the **clipboard at the security desk**: quick snapshots of permissions + a “version number” so badge changes take effect instantly.
* **Backend guards** = the **door locks** inside: they read the badge, check your tenant + permissions, and only then open doors (APIs/records).
* **Frontend** = the **signage & map**: built from the rulebook. It shows only the destinations (pages/buttons) your badge allows.

If you were using Supabase Postgres + RLS, the “door locks” would be table-level RLS policies. In our design, data lives in **MongoDB**, so **our backend enforces** the equivalent of RLS (tenant+permission checks) on every query.

**B) Trust Model: What we do and do not trust**

* We **trust** a **verified JWT** (signature, expiry) that **we** issue as an **HTTP-only cookie** after the exchange.
* We **do not trust**:
  + Any tenantId provided by clients (we always take tenantId from the token).
  + Any token stored in JavaScript memory/localStorage (we don’t store tokens in JS).
  + Stale permissions: we block them via **entitlementsVersion** (“ev”) checks in Redis.
* We assume identities can change (role updated, user revoked) and **design for instant effect** without logging users out.

**C) Components & Responsibilities (who does what)**

**Supabase (Identity)**

* Handles sign-in/sign-up, SSO, MFA, password resets.
* Returns a **short-lived session** (access + refresh) to the browser **just for the exchange** step.

**Backend (FastAPI) — “BFF / Security Desk”**

* **Owns user session after exchange** via **HTTP-only cookies**.
* Verifies tokens, enforces **tenant isolation** and **RBAC**, handles **refresh**, maintains **revocation** (JTI blocklist), and emits **audit logs**.
* Provides the endpoints:
  + POST /auth/exchange
  + POST /auth/refresh
  + POST /auth/logout
  + GET /me/context (menu/actions model for UI)

**MongoDB (Authorization Source of Truth)**

* **memberships** (user↔tenant), **roles** (permissions), **ui\_resources** (optional page/action map).
* Changes here **bump ev** so existing badges are considered out-of-date on the next request.

**Redis (Speed + Control)**

* Caches flattened permission sets; stores ev:{tenantId}:{userId} and jti:block:{jti}.
* The **ev check** gives **instant effect** for role changes.

**Browser (Web App)**

* Starts sign-in with Supabase, then **immediately** calls /auth/exchange.
* Never stores tokens; just sends **same-origin requests** so cookies auto-attach.
* Renders UI from **/me/context** (no hardcoded RBAC).

**D) Data Model Overview (AuthZ)**

* **users**: \_id, supabaseUserId, email, status, audit
* **memberships** (unique {tenantId, userId}):
  + roles:[roleIds/names], rooms:[…], status, entitlementsVersion:int, updatedAt
* **roles** (unique {tenantId, name}):
  + permissions:[string] e.g., students.read, messages.create
* **ui\_resources** (optional but recommended):
  + pages: { key:'page.students', required:['students.read'] }
  + actions: { key:'action.student.edit', required:['students.update'] }
* **refresh\_sessions**:
  + { userId, tenantId, hashedRefresh, device, userAgent, ip, createdAt, lastUsedAt, revoked }

**Indexing (tenant isolation):** Compound unique indexes include tenantId. All queries **inject** tenantId from the token.

**E) Tokens & Cookies (what’s inside, how long they live)**

**App Access Token (JWT in HTTP-only cookie)**

* **Claims** (minimal):
  + sub = supabase\_user\_id
  + tid = active tenant
  + rids = roles (ids or names)
  + ev = entitlementsVersion snapshot
  + ph = optional permissions hash
  + jti = unique token id (for blocklist)
  + exp = short expiry (15–30 min)
* **Signature**: RS256 or EdDSA (asymmetric) with key rotation.
* **Where stored**: **HTTP-only cookie** access (Secure, SameSite=Strict/Lax).

**App Refresh Token**

* **Where stored**: **HTTP-only cookie** refresh (Secure, SameSite=Strict/Lax).
* **Server storage**: store **only a hash** in DB/session table.
* **Rotation**: rotate on every use; invalidate old one.
* **TTL**: 7–30 days.

**CSRF Token (for unsafe methods)**

* **Non-HTTP-only** csrf cookie + matching request header (double-submit).
* Keep **GET** idempotent. Require CSRF header for POST/PUT/PATCH/DELETE.

**F) Canonical Flows (step-by-step with diagrams)**

**Phase 1 — Login & Initial Exchange**

sequenceDiagram

actor User

participant FE as Browser (Frontend)

participant SA as Supabase (IdP)

participant API as Backend (/auth/exchange)

participant MG as MongoDB (AuthZ)

participant RD as Redis (Cache/ev)

User->>FE: Enters credentials, clicks "Login"

FE->>SA: Authenticate (password/SSO/MFA) using anon key

SA-->>FE: Supabase session (access & refresh)

Note over FE: Session is ephemeral—used only once

FE->>API: POST /auth/exchange (send Supabase session + tenant choice)

API->>SA: Verify session server-side (sig, exp, project)

SA-->>API: Valid (uid), details

API->>MG: Lookup membership (uid + tenantId)

API->>API: Resolve roles → flatten permissions; compute ev

API->>RD: Set ev:{tid}:{uid} = ev; cache permset:{tid}:{uid}

API-->>FE: Set HttpOnly cookies {access(15–30m), refresh(7–30d)}; return safe user context

Note over FE,API: Browser discards Supabase session; uses cookies going forward

**What changes where (state changes):**

* **MongoDB**: unchanged during login (unless you create user records on first login).
* **Redis**: writes ev:{tid}:{uid} and permset:{tid}:{uid} (TTL 5–15 min for permset).
* **Cookies (browser)**:
  + access (JWT with tid, rids, ev, jti, exp) — **HttpOnly**
  + refresh (opaque/random) — **HttpOnly**, rotation on use
  + optional csrf (non-HttpOnly) for double-submit

**Why this is safe:**

* **Passwords** never hit your servers (Supabase handles identity).
* **Tokens** never land in JS (HttpOnly cookies).
* You still get **full RBAC control** via exchange, with **instant revocation** via ev.

**Phase 2 — Authorized API Call (what checks run on every request)**

flowchart LR

A[Request from Browser] --> B[Read access cookie]

B --> C[Verify JWT sig/exp]

C --> D[Check JTI blocklist in Redis]

D --> E[Compare token.ev vs Redis ev:{tid}:{uid}]

E -->|Mismatch| F[401 EV\_OUTDATED → client calls /auth/refresh]

E -->|Match| G[Load permset from Redis (or rebuild from Mongo)]

G --> H[Check route permission]

H --> I[Inject tenantId from token into DB queries]

I --> J[Return tenant-scoped data]

**Important details:**

* **No Authorization header** from JS; cookies are attached automatically.
* **Tenant isolation**: We never accept tenantId from the client body/query; we **inject** tenantId = token.tid.
* **Permission check**: route declares the required permission (e.g., students.read), which we test against permset.

**Example (conceptual)**:

* Route: GET /students requires students.read.
* Pipeline:
  + Verify token → tid = TENANT\_A, sub = 123, ev=12
  + If ev != Redis ev:{TENANT\_A}:{123} → 401 EV\_OUTDATED
  + Else load permset → confirm it contains students.read
  + Query: find({ tenantId: "TENANT\_A" }) (server-injected)
  + Return data.

**Phase 3 — Silent Refresh (keeping users logged in, with fresh RBAC)**

**When it happens:**

* **Timer** (e.g., at T-2 minutes before access expiry), **or**
* On **401 EV\_OUTDATED/EXPIRED** from the API.

**What happens:**

1. Browser calls POST /auth/refresh (cookies auto-attach).
2. Backend:
   * Validates **refresh** (hash lookup; not revoked; normal device/UA/IP).
   * (Optionally) Re-computes permissions if ev changed since last time.
   * **Rotates** refresh (writes new hash, invalidates old).
   * Mints a new **access** cookie (fresh exp, fresh claims).
3. API retry succeeds; UI continues seamlessly.

**User experience:** No popups, no re-login—completely silent.

**Phase 4 — Role Change (instant effect without logout)**

**Scenario:** Admin promotes a teacher to “Lead Teacher” (adds students.update).

**State changes:**

* **MongoDB**: role or membership updated.
* **Backend job** (or the admin API itself) **bumps entitlementsVersion** for affected membership(s).
* **Redis**: ev:{tid}:{uid} updated (e.g., 12 → 13).

**What users see:**

* Their **next request** still carries a token with ev=12.  
  The API compares to Redis (ev=13) → returns **401 EV\_OUTDATED**.  
  The browser immediately calls /auth/refresh, gets a new access cookie with ev=13, and re-renders the UI (now **Edit** buttons appear).

**Phase 5 — Tenant Switch (multi-tenant members)**

**Scenario:** A regional admin has Tenant A & Tenant B.

**Flow:**

1. User selects tenant in the UI.
2. Browser calls /auth/exchange **or** /auth/switch with the selected tenant.
3. Backend verifies membership for that tenant, recomputes ev, resets permset and sets a fresh **access** cookie with tid = NEW\_TENANT.
4. API calls now scope to the new tenant; UI updates from /me/context.

**G) Frontend Responsibilities (simple & safe)**

* Use Supabase client to sign in → **immediately** call /auth/exchange and **discard** the Supabase session.
* **Never** store tokens; rely on same-origin **cookies**.
* On app load (or tenant switch) call **GET /me/context** to get pages/actions model; render accordingly.
* On **401 EV\_OUTDATED/EXPIRED**, **silently call /auth/refresh** and retry once.
* Send a **CSRF header** (value == csrf cookie) for unsafe methods.

**H) Security Controls (cookie mode essentials)**

* **Cookies**: HttpOnly, Secure, SameSite=Strict (or Lax if needed); narrow domain/path.
* **CSRF**: Double-submit cookie or server-stored token; enforce on POST/PUT/PATCH/DELETE.
* **Headers**: HSTS, CSP (no inline scripts), X-Content-Type-Options, frame-ancestors (or X-Frame-Options), Referrer-Policy, Permissions-Policy.
* **Rate limits**: /auth/exchange, /auth/refresh, password resets, and admin endpoints.
* **Redis**: ev:{tid}:{uid}, permset:{tid}:{uid} (TTL 5–15 min), jti:block:{jti}.
* **Audit**: log logins, exchanges, refreshes, role/membership changes (never log secrets).

**I) What can go wrong (and what happens)**

* **Access expired** → API 401 EXPIRED → browser refreshes silently → OK.
* **RBAC changed** → API 401 EV\_OUTDATED → browser refreshes → UI updates.
* **Refresh revoked/compromised** → refresh fails → user is redirected to login.
* **Redis down** → backend recomputes from Mongo (slower) but correct.
* **CSRF attempt** → blocked by SameSite + CSRF header requirement.

**J) One-Screen Summary**

* **Login**: Browser → Supabase; **immediately** /auth/exchange → backend sets **HttpOnly cookies**.
* **Every call**: Backend reads access cookie → verifies → checks Redis ev → enforces permissions → injects tenantId.
* **RBAC change**: bump ev → next call refreshes → **instant effect**.
* **Security**: No tokens in JS; cookies + CSRF; short access TTL; refresh rotation; audit + rate limits.
* **UI**: Gets a **menu/actions model** from /me/context; no hardcoded RBAC.