**Part 2 — Admin & RBAC Modeling, UI Mapping, Threat Model, and Operations**

**A) Admin Role Modeling (how tenants control access without code)**

**1) Core objects**

* **Tenant**: a school/daycare (isolated data boundary).
* **User**: identified by supabase\_user\_id.
* **Membership**: *(tenant, user)* binding with roles and attributes (e.g., assigned rooms).
* **Role**: a named bundle of permissions (per tenant, customizable).
* **Permission**: a verb-like string (e.g., students.read, attendance.mark).
* **Entitlements Version (ev)**: integer incremented when any membership/role affecting the user’s effective permissions changes.

**2) Role templates and custom roles**

* Ship **starter templates** (editable per tenant):
  + owner, admin, lead\_teacher, teacher, assistant, parent, billing\_manager, support\_viewer (optional).
* Tenants can **clone and customize** a template or **create a role from scratch** by toggling permissions.
* **Naming rules** (recommended):
  + Lowercase, snake\_case: lead\_teacher, billing\_manager.
  + Unique per tenant ({tenantId, role.name}).

**3) Role editor UX (no code changes)**

* **Left pane**: Permission catalog grouped by domain (Students, Staff, Attendance, Messaging, Rooms, Billing, Reports, Admin).
* **Right pane**: Role under edit, with sections:
  + **Included permissions** (chips)
  + **Constraints** (optional ABAC filters, e.g., “limited to rooms: Foxes, Bears”)
  + **Preview**: Pages & actions this role unlocks (from ui\_resources)
* **Save** → backend writes roles.permissions and bumps **ev** for all memberships using this role.

**4) Membership manager**

* For each user in a tenant:
  + Assign one or multiple roles.
  + Optional attributes for ABAC (e.g., rooms=["Foxes","Bears"]).
  + Status: active | revoked | pending.
  + Saving membership changes **bumps ev** for that (tenant, user).

**5) Principles**

* **Least privilege**: start narrow; add only the permissions needed.
* **Composability**: prefer small, clear permissions over mega-roles.
* **Clarity**: permission names reflect the action and resource (resource.action).
* **Auditability**: every role or membership change is logged (who, what, when).

**B) ACL vs ACF (resources vs actions)**

**ACL (Access Control List)** — the set of **resources** the user can see/use.  
**ACF (Action Control Functions)** — the **operations** the user can perform on those resources.

**1) Permission naming convention**

* resource.action in lowercase, dot-delimited.
* Examples:
  + Students: students.read, students.create, students.update, students.delete, students.export
  + Staff: staff.read, staff.create, staff.update, staff.delete
  + Attendance: attendance.view, attendance.mark, attendance.export
  + Messages: messages.read, messages.create, messages.moderate, messages.delete
  + Rooms: rooms.read, rooms.assign, rooms.update
  + Billing (optional phase): billing.read, billing.charge, billing.export
  + Admin: admin.roles.read, admin.roles.update, admin.memberships.update, admin.audit.read

**2) Page/action mapping (UI resources)**

* **Pages (ACL)** require at least one **read** permission for that domain.
* **Actions (ACF)** require the corresponding **action** permission.
* Example (server-maintained ui\_resources):

{

"pages": [

{ "key": "page.students", "label": "Students", "required": ["students.read"] },

{ "key": "page.staff", "label": "Staff", "required": ["staff.read"] },

{ "key": "page.attendance", "label": "Attendance", "required": ["attendance.view"] },

{ "key": "page.messages", "label": "Messages", "required": ["messages.read"] },

{ "key": "page.admin", "label": "Admin", "required": ["admin.roles.read"] }

],

"actions": [

{ "key": "action.student.create", "required": ["students.create"] },

{ "key": "action.student.edit", "required": ["students.update"] },

{ "key": "action.student.delete", "required": ["students.delete"] },

{ "key": "action.attendance.mark","required": ["attendance.mark"] },

{ "key": "action.message.send", "required": ["messages.create"] },

{ "key": "action.message.moderate","required": ["messages.moderate"] }

]

}

* The **frontend** renders pages and controls only when required ⊆ userPermissions.

**C) ABAC (Attribute-Based Access Control) patterns**

RBAC answers “*what role are you?*”. ABAC adds “*and do your attributes permit this object/action right now?*”.

**1) Common daycare attributes**

* **User attributes** (from membership):
  + rooms: list of rooms/classrooms the staff belongs to (e.g., ["Foxes","Bears"]).
  + relationship: for Parents, link to studentIds they are guardians of.
  + employment\_status: active, suspended.
* **Object attributes**:
  + Student record has roomId (or assigned homeroom).
  + Message has senderId, targetRoomId, tenantId.
  + Attendance record has roomId, date.
* **Context attributes**:
  + Time-of-day constraints (e.g., attendance marking windows).
  + Feature flags per tenant.

**2) Examples (how ABAC narrows allowed ops)**

* **Teacher may edit only students in their rooms**
  + RBAC: students.update required.
  + ABAC: student.roomId ∈ membership.rooms.
* **Parent may view only their child**
  + RBAC: students.read (or students.read\_mine if you prefer).
  + ABAC: student.\_id ∈ user.guardianOf.
* **Attendance marking window**
  + RBAC: attendance.mark.
  + ABAC: currentTime ∈ tenant.attendanceWindow (e.g., 6am–6pm local).
* **Messaging moderation**
  + RBAC: messages.moderate.
  + ABAC: message.tenantId == token.tid (always) and maybe message.roomId ∈ membership.rooms if moderating room-specific channels.

**3) Enforcement pipeline with ABAC**

flowchart LR

A[Token verified] --> B[ev gate vs Redis]

B --> C[Load permset (Redis/DB)]

C --> D[Check route permission e.g., students.update]

D --> E[Build ABAC filter from membership attrs (e.g., rooms)]

E --> F[Inject tenantId + ABAC filter into DB query]

F --> G[Perform operation]

ABAC is not a separate store; it’s **derived** from membership attributes and injected into queries/updates.

**4) Modeling “scoped permissions”**

Two styles work:

* **Single permission + ABAC**: students.update plus ABAC filters for room scoping.
* **Namespaced permissions**: students.update.my\_rooms vs students.update.all.  
  Use sparingly; ABAC often suffices without exploding permission count.

**D) UI Mapping (how the UI stays dynamic without hardcoding)**

**1) /me/context response (server → client)**

* Returns:
  + tenantId, roleNames, permissions (flattened list)
  + menuModel (pages/actions and whether they’re enabled)
  + Optional featureFlags and ABAC hints (e.g., allowed rooms list)

**2) Frontend behaviors**

* **Navigation build**: include pages where required ⊆ permissions.
* **Control gating**: enable buttons where required ⊆ permissions.
* **ABAC hints**: show only room filters the user is allowed to see (nice-to-have).
* **On 401 EV\_OUTDATED**: silently refresh and re-render menu/actions.

**3) Multi-tenant UX**

* On tenant switch:
  + Call /auth/switch or /auth/exchange with the new tenant.
  + Recompute ev, set fresh access cookie, fetch new /me/context.
  + Rebuild nav and controls accordingly.

**E) Monitoring & Alerting (what to watch and when to page)**

**1) Metrics (per tenant + global)**

* **Auth Exchange**: success rate, latency, failures by reason.
* **Refresh**: success/failure counts, rotation errors.
* **401 EV\_OUTDATED**: rate (should spike right after role edits; otherwise low).
* **403 Permission Denied**: rate per endpoint (anomalies can indicate config mistakes or abuse).
* **Redis**:
  + permset cache hit ratio (aim >90% on hot paths)
  + ev read latency (should be low & stable)
* **Latency**: p50/p95 for guarded endpoints (e.g., GET /students).
* **Login failures**: by IP, by account (detect brute force).
* **Anomalies**: refresh storms, sudden IP geolocation changes for a user.

**2) Alerts**

* **Auth exchange failures** > X% over Y minutes.
* **Refresh failures** spikes (could indicate cookie/CSRF misconfig).
* **Redis miss ratio** > threshold → degraded perf.
* **403 rate** surge on particular action → RBAC regression.
* **Admin role changes** volume spike → possible mass misconfig.
* **Suspicious activity**: multiple tenants accessed by same sub with far-apart geos within short window.

**3) Dashboards**

* **Funnel**: Login → Exchange → First API call.
* **RBAC changes** vs **EV\_OUTDATED** occurrences.
* **Tenant isolation** checks (e.g., denied cross-tenant attempts).
* **Top 10 slowest guarded endpoints**.

**F) Incident Runbooks (step-by-step responses)**

**1) Account compromise suspected**

1. **Blocklist jti** for active tokens (jti:block:{jti} in Redis).
2. **Revoke refresh sessions** for the user (delete/mark revoked by hash).
3. Force **password reset** and **MFA enrollment** (IdP).
4. Review **audit logs** (recent role changes, access attempts).
5. Notify tenant admin if needed.

**2) Stolen refresh token suspected**

1. Invalidate **all refresh sessions** for the (tenant, user).
2. **Rotate signing keys** only if broader breach suspected (plan overlap carefully).
3. Notify user; require re-auth.
4. Add anomaly detection rules for future (device/UA/IP binding).

**3) RBAC misconfiguration (wide unintented access)**

1. Identify offending role change from audit logs.
2. **Revert role** to last good config.
3. **Bump ev** for impacted memberships.
4. Add a regression test for that permission edge case.

**4) Redis outage or high error rate**

1. Fail open vs fail closed **policy decision**:
   * For read-only endpoints, you may **fail closed** (prefer secure default).
   * For mission-critical read paths, temporarily **recompute from DB** (slow but correct).
2. Increase DB capacity if needed; page infra team.
3. Add backpressure & protect the DB from thundering herds.

**5) Supabase outage (login path impacted)**

1. Existing sessions continue to work until access expires.
2. **Refresh** may fail; communicate degraded sign-in/refresh to users.
3. Consider extending **grace window** in backend (short-term) if acceptable.
4. Coordinate with Supabase status, communicate on status page.

**6) CSRF/XSS incident**

* **CSRF**: confirm SameSite + CSRF header checks; tighten CORS.
* **XSS**: deploy CSP with no inline scripts; sanitize suspect content; rotate secrets/tokens if needed; conduct code review on recent changes.

**G) Threat Model & Mitigations**

| **Threat** | **Vector** | **Impact** | **Mitigations** |
| --- | --- | --- | --- |
| **XSS** | Malicious inline/script injection steals tokens | Account takeover | **HttpOnly cookies** (tokens not in JS), strong **CSP**, sanitize untrusted content, dependency hygiene |
| **CSRF** | Cross-site form/requests using cookies | Unwanted state changes | **SameSite=Strict/Lax**, **CSRF token** on unsafe methods, check **Origin/Referer** |
| **Token replay** | Stolen access/refresh used elsewhere | Unauthorized access | Short access TTL; **refresh rotation**; **JTI blocklist**; device/UA/IP binding; anomaly alerts |
| **Password brute force** | Credential stuffing | Account compromise | IdP rate-limits, bot checks, MFA, breach password checks, generic errors |
| **Session fixation** | Reuse of known session identifiers | Hijacked session | Rotate tokens on login/refresh; deny old refresh after rotation |
| **Tenant escape (IDOR)** | Supplying another tenant’s ID | Data leakage | **Ignore client tenant**; **inject token.tid** server-side; compound unique indexes with tenantId |
| **Privilege escalation** | Gaining admin powers via misconfig | Broad access | Permission catalog discipline; admin endpoints guarded; **audit** role edits; **ev** version gate |
| **Injection (DB/NoSQL)** | Unvalidated inputs | Data corruption/leak | Strict input validation; typed models; reject content-type mismatch; parameterized queries |
| **SSRF/Open redirect** | Misused callback URLs | Token theft | Strict allowlists for redirect URLs; validate upstream calls |
| **Supply chain** | Vulnerable dependencies | RCE/data exfil | Dependency scanning (pip-audit), CSP, image scanning, key rotation policies |

**H) Day-in-the-Life Scenarios (user journeys end-to-end)**

**1) Parent (single tenant)**

* **Login → Exchange**: Gets access/refresh cookies.
* **Home page**: /me/context returns pages: ["Messages","My Child"], actions: ["message.send"].
* **View child profile**: API injects tenantId and ABAC (studentId ∈ user.guardianOf).
* **Send a message**: Must have messages.create; if removed by admin, next click returns 401 EV\_OUTDATED → refresh → button disappears.
* **Cross-tenant URL**: Any attempt fails; token tid enforces isolation.

**2) Teacher (assigned to “Foxes” room)**

* **Permissions**: students.read, students.update, attendance.mark, messages.create.
* **ABAC**: Limited to rooms ["Foxes"].
* **Edit student**: Only students where roomId ∈ ["Foxes"].
* **Attendance**: Allowed during window (ABAC by time).
* **Promotion**: Admin grants lead\_teacher (+ students.update for all rooms). Admin saves → ev bumped → next request 401 EV\_OUTDATED → refresh → teacher now sees more students.

**3) Admin (multi-tenant regional)**

* **Tenant switch**: Uses /auth/switch to move between Tenant A and Tenant B; new access minted with new tid.
* **Role edit**: Updates teacher role to add messages.create. Save bumps ev for all members using that role; teachers’ UI updates on next refresh.
* **Audit**: Reviews role change log and 403 spikes on messages.create after a policy tweak.

**I) Acceptance Tests & UAT Checklist**

**1) Happy paths**

* Login → Exchange → /me/context → first API call success.
* Silent refresh before expiry; user stays signed in.
* Tenant switch re-computes menu/actions correctly.

**2) RBAC correctness**

* Parent cannot access another parent’s child (ABAC).
* Teacher can update only in assigned rooms (ABAC).
* Role change triggers **401 EV\_OUTDATED** and seamless refresh → UI updates.

**3) Tenant isolation**

* Forcibly passing tenantId from client is ignored; server injects token.tid.
* Cross-tenant attempts always denied.

**4) Security behaviors**

* Cookies are HttpOnly, Secure, SameSite=Strict/Lax.
* CSRF checks enforced on unsafe methods.
* Access TTL ~20m; refresh rotation on every use.
* JTI blocklist kills a token immediately.

**5) Resilience**

* Redis miss gracefully recomputes from DB (slower but correct).
* Supabase outage: existing sessions work until expiration; refresh failure leads to clear error and re-login.

**J) Example Permission Catalog (starter set)**

**Students**  
students.read, students.create, students.update, students.delete, students.export

**Staff**  
staff.read, staff.create, staff.update, staff.delete

**Attendance**  
attendance.view, attendance.mark, attendance.export

**Messages**  
messages.read, messages.create, messages.moderate, messages.delete, messages.export (optional)

**Rooms**  
rooms.read, rooms.update, rooms.assign

**Billing (optional phase)**  
billing.read, billing.charge, billing.export, billing.refund

**Reports**  
reports.read, reports.export

**Admin**  
admin.roles.read, admin.roles.update, admin.memberships.update, admin.audit.read

You can extend this catalog as features grow; keep names clear and stable.

**K) Quick Reference (cheat sheets)**

**Redis keys**

* ev:{tenantId}:{userId} → integer, bumped on any RBAC change
* permset:{tenantId}:{userId} → flattened list (TTL 5–15m)
* roles:{tenantId}:{roleName} → role definition (TTL 30–60m)
* jti:block:{jti} → revocation flag (TTL ≤ token lifetime)

**Cookie defaults**

* access: HttpOnly, Secure, SameSite=Strict/Lax, TTL ~20m
* refresh: HttpOnly, Secure, SameSite=Strict/Lax, TTL 7–30d, **rotated**
* csrf: non-HttpOnly, used with header on unsafe methods

**Guard order (every request)**

1. Verify token sig/exp
2. Check jti blocklist
3. Compare token ev vs Redis ev
4. Load permset; check route permission
5. Inject tenantId from token; apply ABAC filters

**L) Executive Recap (one screen)**

* **Identity at Supabase**, **authorization in Mongo**, **speed & revocation via Redis**.
* **Option 3**: Browser authenticates → immediately exchanges with backend; backend owns **HTTP-only cookies**.
* **UI is data-driven** from /me/context; no hardcoded RBAC.
* **Instant RBAC changes** via ev version gate; silent refresh updates UI without logout.
* **Security-by-default**: no tokens in JS, cookies + CSRF, short lifetimes, refresh rotation, audit trails, and strict tenant injection.