NDYRA Platform vNext
Social + Member + Gym + Business OS

Soup-to-Nuts Engineering Blueprint (Aelric Edition)
Version v7.3 — Anti-Drift Locked + Scale-Ready + System-of-Record (Mindbody Replacement) + Flawless QC

Date: 2026-02-19
Owner: William Davis Moore
Audience: Aelric Architect (Lead Dev)
Tagline: Where Discipline Becomes Identity.

Change Log (v7.3.1)

- Integrated Blueprint v7.1 Scaling & Reliability Patch (expands Section 13 with capacity targets, index manifest, storage/media performance defaults, abuse controls, observability, and load testing gates).
- Added 'NDYRA Fully Replaces Mindbody' system-of-record requirements (tenant cutover flag, deterministic membership status enforcement, waiver + walk-in automation, migration toolkit).
- Extended booking/check-in with a Smart Booking Fork (tokens vs renew membership), canonical transactional booking RPC, and auditable staff overrides.
- Added new data model tables (waivers, migration batches, check-in overrides) with RLS-first policies, required indexes, and anti-drift gates.
- Updated API Blueprint with new server-side functions for migration import, waiver template management, and check-in overrides, plus RPC sign_current_waiver for secure waiver signing (no new patterns; service-role only for high-impact writes).
- Updated Testing + Gates to include E2E coverage for Quick Join, booking fork, waiver enforcement, migration idempotency, and scale/readiness load tests.
- Roadmap wording updated so CP31 social + E2E harness stays intact, while gym-onboarding-at-scale is gated on Migration Toolkit MVP + Quick Join + Waivers + Deterministic Check-In + Smart Booking Fork.

Flawless QC hardening: fixed RPC signature docs; locked waiver_signatures against client inserts (signing via RPC that verifies Storage object existence); enforced get_following_feed as SECURITY INVOKER (RLS preserved); hardened book_class_with_tokens with system_of_record gating + token eligibility + idempotency.

- v7.3.1 corrections: removed a duplicated Section 6 block, tightened waiver signature key format (Appendix B + sign_current_waiver), updated is_tenant_member eligibility (active OR comp), added tenant kill-switch requirements, and fixed residual v7.2 references.

How to Use This Blueprint

This document is the single source of truth for NDYRA. It is written to prevent build ambiguity and UI hiccups. If a feature is not in this doc, it is not in scope; if it is in this doc, it must be implemented exactly as specified.

Operating rules:

- Database Row Level Security (RLS) is the source of truth. Frontend hiding buttons is not security.
- No page-specific hacks. If a UI element is needed, it becomes a component or token.
- Client may read/write directly to Supabase only where RLS guarantees correctness.
- Anything money/ledger/biometrics/tokens are server-side only (service role) and fully auditable.
- All new routes must be declared in the Route Map section and must match the JS module naming convention.
- Follow existing architecture: ES modules + Route Map + UI Build Spec updates first. No new frontend framework; no new root folders.

0A. Anti-Drift Protocol (Mandatory)

ADP-0 Canonical Sources (Single Source of Truth)

- This blueprint is the canonical spec for product surface + architecture + data model.
- CP27 migrations SQL is the canonical schema implementation for Social Core.
- CP27 RLS tests are the canonical regression harness. If tests are not updated, the change is not real.
- CP27 Build Order Manifest is the canonical execution sequence. Do not reorder unless the blueprint is updated first.
- If it is not written, it does not exist. If it exists, it must be written.

ADP-1 Structure Freeze (No New Patterns)

- No new frontend framework (React/Vue/etc.). Extend via ES modules only.
- No new root-level folders. Allowed roots: /site, /netlify/functions, /supabase (if using CLI), /docs (for ADRs).
- Every new route MUST be added to Route Map + UI Build Spec before code is written.
- One pattern for pages: <body data-page="..."> + page module loaded via <script type="module">. No page-specific inline JS.
- One pattern for shared UI: tokens + components. If a UI rule repeats 2+ times, it becomes a component.

ADP-2 Database + RLS Freeze (RLS is Law)

- Default deny. RLS enabled on every public table (no exceptions without explicit written justification).
- No policy may use using(true) or with check(true) unless the table is explicitly approved as harmless-public AND recorded in the allowlist in the Anti-Drift Audit script.
- Post-adjacent tables MUST route through can_view_post(post_id) (posts, media, reactions, comments, stats, notifications visibility).
- Blocks MUST be enforced at the database layer (via is_blocked_between() used in can_view_post()).
- Money/ledger tables: client cannot insert/update/delete. Only server (service role / RPC) may write, with audit logging.
- Booking enforcement must be transactional and server-side for token spends and capacity enforcement.
- Waiver enforcement must be server-side (RLS or RPC), not UI-only.

ADP-3 Storage + Media Freeze (No Private Media Leaks)

- All uploads go direct-to-Supabase Storage (never through Netlify Functions).
- Bucket policies must be visibility-aware: public content readable; non-public content not publicly readable.
- No permanent public URLs for private media. Use signed URLs with short TTL and enforce by policy.
- Set cacheControl on uploads per Section 13.8.

ADP-4 Feed/Query Discipline (Scale-Safe by Default)

- Seek pagination only (created_at/id cursor). Never OFFSET pagination on feeds.

- No COUNT(*) per request in hot paths. Use cached post_stats / incremental triggers.
- Every feed query must have an index plan documented in the blueprint.
- No N+1 patterns in feed rendering: all required fields fetched in one query (joins/views ok).
- If follow graph grows large (500+), switch Following feed to server-side join-based RPC (Section 13.6).

ADP-5 Review Gates (Merge Blockers)

- No PR merges unless ALL gates pass: Anti-Drift Audit + RLS tests + E2E (Playwright) + QA report. If any gate fails, fix first—no exceptions.
- Any PR touching DB must include: migration diff, Anti-Drift Audit output, and updated RLS tests (if policies changed).
- Any PR adding a route must include: Route Map update + UI Build Spec update + layout reference.
- Any PR adding a table must include: RLS enabled, policies, indexes, and anti-drift allowlist updates where appropriate.
- Any PR touching tokens/ledger or booking capacity must include: RPC transaction proof, audit_log entry, and idempotency proof.
- Any PR touching waivers or migration import must include: negative tests for data leaks and idempotency tests.

ADP-6 Anti-Drift Checkpoints (Hard Stops)

- Checkpoint AD-1: Run migrations on staging, run Anti-Drift Audit SQL, snapshot RLS fingerprint, commit fingerprint to repo.
- Checkpoint AD-2: PostCard + FeedQuery finalized. No new UI variants without updating Component Spec.
- Checkpoint AD-3: RLS tests pass; audit returns zero failures; storage policies validated against private media leak attempts.
- Checkpoint AD-4: Cross-platform QA pass (iOS Safari + Android Chrome + Desktop PWA) and abuse smoke test.
- Checkpoint AD-Scale: Load tests (Section 13.12) pass on staging with a realistic dataset before onboarding 100 gyms / 20k users.

ADP-7 Drift Detection (Automated)

- Run the Anti-Drift Audit SQL in CI (or at minimum before each merge). It must fail the build on violations.
- Store and track an RLS fingerprint per checkpoint. Any fingerprint change requires an explicit approval note in /docs/adr/.
- Maintain a short ADR log. Every structural change adds an ADR.

0. NDYRA Product Thesis + Non-Negotiables

0.1 The Big Idea

NDYRA is a fitness operating system: a social fitness network + a gym business platform in one product. It replaces scattered tools and makes training identity actionable (bookings, memberships, tokens, biometrics, programming, events).

0.2 The Differentiator (Must Not Be Compromised)

- A profile is not bio + photos. A profile is a living performance card: training identity, biometrics snapshot, workout history, streaks, weekly minutes.
- Gym memberships + token wallets are actionable access primitives.
- Achievements (global + gym-defined).
- Privacy-controlled sharing (public vs followers vs clubs vs gyms vs staff).

0.3 Platform Vibe: Gym-Floor Social (No Negativity)

NDYRA's social layer must feel like a gym floor, not a comment war zone. We enforce this through product design (encouragement-forward reactions, comment friction, slow-mode), clear guidelines, and a strong moderation pipeline (gym-level + platform-level).

0.4 Brand Locks (Decisions Final)

Lock	Decision
Platform Name	NDYRA
Tagline	Where Discipline Becomes Identity.
Core UI Theme	Black canvas + white text + red accent. Cinematic but airy.
Feed Orientation	Portrait-first vertical feed. Default media ratio 4:5.
Reactions	Encouragement-only: 🔥 💪 ⚡ 👏 🧠 (no downvote, no laugh).
Comments	Max depth = 2. First comment requires reminder checkbox. Repeat offenders enter slow-mode.

0.5 Definition: NDYRA Fully Replaces Mindbody (Locked)

When a tenant sets NDYRA as their system of record, gym operations become authoritative in NDYRA. NDYRA must be strictly easier than current gym ops, and must fix known failures in legacy platforms (e.g., 'member can check in even when payment failed'). NDYRA must eliminate paper waivers and manual data entry for walk-ins.

- Member roster (CRM-lite).
- Membership plans + membership status enforcement.
- Scheduling (class_types + class_sessions).

- Booking + waitlist + cancellation rules.
- Check-in enforcement (membership + waiver + tokens).
- Payments for gym memberships + token packs (Stripe-backed).
- Waivers (digital signature + versioning).
- Walk-in onboarding automation (QR/kiosk).
- Exports (attendance + payroll CSV).

ClassPass is not migrated (it is a marketplace). NDYRA remains system-of-record for membership and check-in regardless.

0.6 No Drift / No Duplicate Patterns Amendment (Locked)

Aelric — implement this Blueprint v7.3.1 update as locked requirements. This is a 'no drift / no duplicate patterns' amendment that extends existing sections (Routes, Scheduling/Booking/Check-in, Payments/Ledger, Data Model, API Blueprint, Testing). Do not introduce a new framework, new root folders, or page-specific hacks. ADR note: v7.3.1 merges the scaling patch + the system-of-record amendment; no new patterns/frameworks.

Follow existing rules without exception:

- ES modules + Route Map + UI Build Spec updates first (before code).
- RLS is law (default deny, helper functions, post-adjacent tables gated by can_view_post).
- Money/ledger and other high-impact flows are server-side only.
- Transactional booking rules (FOR UPDATE locks; no oversells).
- Audit logging for overrides, ledger, and administrative actions.

0.7 CP31 Compatibility Constraint (Locked)

This update must not conflict with CP31 progress (post detail + create composer + E2E harness polish). Keep CP31 scope intact.

1. Roles, Relationships, Permissions

1.1 Global Roles (Platform-Wide)

Role	Meaning
Guest	No account. Can browse limited public content (public gym profiles + limited public posts).
User	Account created. Can follow gyms/users, post, react, join clubs, and hold tokens.
Member	User with active NDYRA subscription unlocking premium timer/builder and other premium features.
Platform Admin	Master admin. Full tenant visibility, moderation queue, support tools, feature flags.

1.2 Tenant Roles (Per Gym / Tenant)

Tenant roles are stored in tenant_users.role and are enforced by RLS.

Tenant Role	Capabilities
Gym Admin	Owns tenant settings, staff roles, memberships, pricing, scheduling, posts, templates, payroll exports, ads, events.
Gym Staff	Coaches/trainers. Can view check-ins, run class timers, manage attendance, create posts in gym context, moderate gym space.
Trainer	Staff subtype with public trainer profile + availability blocks + private session offerings.

1.3 Relationship Status (User <-> Gym)

Relationships drive content gating and commerce. These are not tenant roles.

Relationship	Unlocks
Follower	Free. Can view gym public posts/announcements; can receive updates; can buy tokens if enabled.
Gym Member	Paid/comp tier. Can book classes/training per plan; can see members-only posts + templates.
Token Holder	Has token wallet for that gym. Can book token-cost

sessions even without membership.

1.4 Permission Enforcement Rules

- Every table in public schema must have RLS enabled.
- RLS policies must reference helper functions (is_platform_admin, is_tenant_staff, is_tenant_member, etc.) to keep policies consistent.
- Money/ledger writes are service-role only (Netlify function or RPC). Clients never directly update token balances or create token_transactions.
- Device connection tokens (OAuth refresh tokens) are service-role only and must be encrypted at rest.
- Storage policies must align with visibility and waiver privacy (no public reads of waiver signatures).

2. Route Map (Public, Member, Business, Admin)

2.1 Public (Marketing + Discovery + Walk-In Join)

Route	Purpose
/	Landing page.
/pricing	NDYRA member subscription (timer/builder premium).
/for-gyms	Gym owner sales page.
/gyms	Gym directory preview (public).
/gym/{slug}	Gym public profile (about, preview posts, schedule preview, follow/join CTAs).
/gym/{slug}/join	Tenant-branded Quick Join (QR/kiosk walk-in onboarding: account + waiver + pay + book/check-in).
/login	Auth.
/join	Create NDYRA account + onboarding.

2.2 Member App (Core)

Route	Purpose
/app/fyp	For You feed (ranked blend).
/app/following	Following feed (chronological).
/app/profile	My profile (performance card + wall).
/app/gyms	Discover/manage gyms, memberships, tokens.
/app/clubs	Clubs list.
/app/clubs/{club_id}	Club main page (poster + feed + goals).
/app/clubs/{club_id}/members	Club members grid (faces + performance stats).
/app/wallet	Token wallet (tenant-aware).
/app/notifications	Notifications.
/app/timer	Timer system (existing HIIT56 tech).
/app/book/class/{class_session_id}	Book class flow (tokens/membership logic + Smart Booking Fork).

/app/post/{post_id}	Post detail + comments.
/app/create	Create post (composer).

2.3 Business Portal (Gym Tenant)

Route	Purpose
/biz/dashboard	KPIs + today schedule + alerts.
/biz/check-in	Scan/search member; Member Readiness Card; attendance mark; deterministic clearance + override.
/biz/classes	Class types, sessions, rosters, waitlist.
/biz/trainers	Trainer profiles, availability, private sessions.
/biz/members	CRM-lite members list, tags, notes.
/biz/posts	Gym announcements + posts + moderation tools.
/biz/timer-templates	Gym timer programming; public/private templates.
/biz/events	Host events; affiliate with Group56.
/biz/payroll	Timesheets + export CSV.
/biz/ads	Boost posts (local targeting).
/biz/settings	Tenant settings, tiers, tokens, staff, branding.
/biz/migrate	Migration toolkit home (status, instructions, upload).
/biz/migrate/members	CSV import: members + memberships + token starting balances.
/biz/migrate/schedule	Optional CSV import: class types + upcoming sessions.
/biz/migrate/verify	Dry-run results + mapping errors + counts.
/biz/migrate/commit	Commit import (idempotent batch).
/biz/migrate/cutover	Set system_of_record='ndyra' and enforce NDYRA as system-of-record.

2.4 Platform Admin

Route	Purpose

/admin/tenants	Create/suspend gyms; support.
/admin/users	Support tools; moderation; account actions.
/admin/moderation	Reports queue; takedowns; bans.
/admin/billing	Stripe event log + subscription states.
/admin/feature-flags	Rollout control.
/admin/audit	Audit log browsing.

3. UI System (Tokens, Grid, Typography, Components)

3.1 Visual Intent

NDYRA UI is cinematic + airy: dark canvas, bright clarity, and red only as intention. The feed feels like a performance gallery, not a chaotic scroll pit.

3.2 Layout Grid + Breakpoints (Locked)

- 8px spacing system only (8/16/24/32/40/48). No random margins.
- Mobile-first. Portrait-first feed. One-handed navigation.
- Max content widths: mobile full width minus 24px padding; desktop centered column 720px for feed.
- Breakpoints: Mobile 375–430; Tablet 768; Desktop 1280; Wide 1440.

3.3 Color Tokens (CSS Variables)

```
:root {
 --ndyra-bg-900: #07070A;
 --ndyra-surface-800: #0F1016;
 --ndyra-surface-700: #141522;
 --ndyra-stroke-700: #242636;
 --ndyra-text-100: #FFFFFF;
 --ndyra-text-300: #B9BBC7;
 --ndyra-text-500: #7C7F92;
 --ndyra-red-500: #E10600;
 --ndyra-red-600: #B90400;
}
```

3.4 Typography

Token	**Size/Line**	**Weight**
H1	32/38	700
H2	24/30	700
H3	18/24	700
Body	16/24	400
Caption	13/18	400
Micro	11/14	400

3.5 Components (Build Once; Pages Are Assemblies)

- AppShell (TopBar + BottomNav mobile; LeftRail desktop later)

- Button (primary red, secondary outline, ghost, destructive, loading)
- Pill/Chip (filters, tags, visibility, membership badges)
- Card (standard, feed card, metric card, gym card)
- Input (text, search, textarea, select, toggles, segmented controls)
- Modal/Drawer (bottom sheet mobile; side drawer desktop)
- MediaTile (FYP wall tiles, profile grid)
- PostCard (header + discipline strip + media + caption + reactions)
- ReactionBar (+ counts) 🔥 💪 ⚡ 👏 🧠
- CommentList + CommentComposer (depth 2, slow-mode)
- Avatar (sizes: 28/36/48/64/96)
- MetricStrip (biometrics + streak + weekly minutes)
- GymCTA (Follow / Join / Buy Tokens)
- Toast system
- NEW: DecisionSheet (bottom-sheet decision UX for Smart Booking Fork; also reusable for other multi-choice blocking decisions).
- NEW: WaiverSignaturePad (signature capture canvas component + submit controls; reusable across tenants).

3.6 Media Rules (Portrait-First)

- Default image ratio: 4:5 (portrait).
- Allowed: 1:1 square; 9:16 vertical video.
- Avoid landscape in main feed; allow landscape only in post detail view.
- All images stored in Supabase Storage with signed/public URLs based on visibility.
- Uploads must enforce payload constraints and set cacheControl per Section 13.8.

3.7 Accessibility + Usability (Non-Negotiable)

- Contrast must pass WCAG AA on dark background.
- Tap targets minimum 44px height.
- Keyboard navigation: focus states visible for desktop.
- Reduced motion: honor prefers-reduced-motion for animations.
- No red-only meaning: pair red with icons/labels for status.

4. UI Build Spec (Dev-Executable)

4.0 Scope Notes

This section defines screen-level contracts and data flows. Any route listed in the Route Map that is in active build scope must have a UI Build Spec entry here.

CP31 baseline (already shipped): /app/post/:id (comments + sticky composer + reactions) and /app/create (composer with media upload up to 10 files), plus E2E harness polish.

4.1 Shared UI Contracts

4.1.1 PostCard Contract

All post-rendering screens must use the same PostCard shape. Do not fork select shapes per page.

```
PostCardProps {
  postId: string
  author: { type: 'user'|'tenant', id: string, name: string, handleOrSlug: string,
avatarUrl?: string }
  visibility: 'public'|'followers'|'members'|'club'|'private'|'staff'
  createdAt: string
  contentText?: string
  discipline: { workoutType?: string, durationMin?: number, streakDays?: number,
weeklyMinutes?: number }
  media: Array<{ id: string, type: 'image'|'video', url: string, width?: number,
height?: number }>
  reactions: { fire: number, strong: number, bolt: number, clap: number, mind: number }
  viewerReaction?: 'fire'|'strong'|'bolt'|'clap'|'mind'|null
  commentsPreview: Array<{ id: string, user: {id, handle, avatarUrl}, body: string }>
}
```

4.1.2 Query + Pagination Rules

- Seek pagination (created_at + id cursor). Never OFFSET.
- Default feed page size: 25 posts. Comments page size: 50.
- Counts come from post_stats, not COUNT().
- No N+1 fetch patterns: PostCard must be renderable from a single select shape.

4.1.3 Base Select Shape (Supabase)

```
const postSelect = `
 id, created_at, visibility, content_text, workout_ref,
 author_user_id, author_tenant_id, tenant_context_id,
 author_user:profiles!posts_author_user_id_fkey ( id, handle, display_name,
avatar_url ),
 author_tenant:tenants!posts_author_tenant_id_fkey ( id, name, slug, logo_url ),
 media:post_media ( id, media_type, storage_path, width, height, sort_order ),
 stats:post_stats ( fire_count, strong_count, bolt_count, clap_count, mind_count,
comments_count, score_48h )
`;
```

4.2 Screen Spec — For You Feed (FYP)

Route: /app/fyp

Permissions: Auth required (User). Guests redirected to /login. RLS filters content by visibility + relationships + blocks.

Core behavior: render a high-signal feed. Use the canonical select shape and seek pagination.

Data Sources

- profiles (viewer profile, location).
- follows_users, follows_tenants (viewer follow graph).
- gym_memberships (viewer memberships for members-only gating).
- tenant_locations (local tenant ids).
- posts + post_media + post_stats (feed content).

Supabase Queries (Exact)

```
// Required: viewer id
const { data: { user } } = await supabase.auth.getUser();
const uid = user.id;

// Viewer profile (location)
const { data: me } = await supabase
  .from('profiles')
  .select('id, handle, display_name, avatar_url, location_city, location_region')
  .eq('id', uid)
  .single();

// Follow graph
const [{ data: fu }, { data: ft }] = await Promise.all([
  supabase.from('follows_users').select('followee_id').eq('follower_id', uid),
  supabase.from('follows_tenants').select('tenant_id').eq('follower_id', uid),
]);
const followedUserIds = (fu ?? []).map(r => r.followee_id);
const followedTenantIds = (ft ?? []).map(r => r.tenant_id);

// Memberships (members-only gating)
const { data: gms } = await supabase
```

```
  .from('gym_memberships')
  .select('tenant_id')
  .eq('user_id', uid)
  .in('status', ['active','comp']);
const memberTenantIds = (gms ?? []).map(r => r.tenant_id);

// Local gyms (city match)
const { data: locs } = await supabase
  .from('tenant_locations')
  .select('tenant_id')
  .eq('city', me.location_city);
const localTenantIds = [...new Set((locs ?? []).map(r => r.tenant_id))];

// World slice (following + memberships)
const worldOr = [
  followedUserIds.length ? `author_user_id.in.(${followedUserIds.join(',')})` : null,
  followedTenantIds.length ? `author_tenant_id.in.(${followedTenantIds.join(',')})` :
null,
  memberTenantIds.length ? `tenant_context_id.in.(${memberTenantIds.join(',')})` :
null,
].filter(Boolean).join(',');

const { data: worldPosts } = await supabase
  .from('posts')
  .select(postSelect)
  .or(worldOr)
  .order('created_at', { ascending: false })
  .limit(25);

// Local slice (public discovery)
const { data: localPosts } = await supabase
  .from('posts')
  .select(postSelect)
  .in('author_tenant_id', localTenantIds)
  .eq('visibility', 'public')
  .order('created_at', { ascending: false })
  .limit(15);

// Trending slice (via post_stats)
const { data: trending } = await supabase
  .from('post_stats')
  .select(`post_id, score_48h, post:posts (${postSelect})`)
  .gt('score_48h', 0)
  .order('score_48h', { ascending: false })
  .limit(15);
```

4.3 Screen Spec — Following Feed

Route: /app/following

If followedUserIds + followedTenantIds exceeds 500 total, switch Following feed to server-side join-based RPC get_following_feed (Section 13.6).

4.4 Screen Spec — Post Detail

Route: /app/post/{post_id}

CP31 baseline implemented: shows one PostCard + full comments list (depth max 2) + sticky composer + reactions.

Supabase Queries (Exact)

```
const { data: post } = await supabase
  .from('posts')
  .select(postSelect)
  .eq('id', postId)
  .single();

const { data: comments } = await supabase
  .from('post_comments')
  .select(`
    id, post_id, user_id, parent_id, body, created_at,
    user:profiles ( id, handle, display_name, avatar_url )
  `)
  .eq('post_id', postId)
  .is('deleted_at', null)
  .order('created_at', { ascending: true })
  .limit(50);
```

4.5 Screen Spec — Create Post

Route: /app/create (CP31 baseline implemented)

Rules: at least one of text or media; max 10 files; uploads go direct-to-Supabase Storage bucket post-media; insert post_media rows; navigate to /app/post/{id}.

4.6 Screen Spec — Quick Join (Walk-In Automation)

Route: /gym/{slug}/join (public; QR/kiosk)

Goal: eliminate paper waivers and staff manual data entry. This is a primary gym-ops weapon.

Flow Steps (Locked)

- Resolve tenant by slug; render tenant branding + purpose selector: drop-in / trial / membership.
- Create account: name, email, phone (required). Optional photo capture.
- Sign waiver: show active waiver template; capture signature; upload to waiver-signatures bucket; call RPC sign_current_waiver(tenant_id, signature_storage_path) to finalize and write waiver_signatures (captures IP/user-agent; no client insert).
- Pay if required: drop-in or membership checkout via serverless Stripe checkout creation.
- Auto-book next class OR generate check-in QR/pass (tenant-configurable).
- Confirm screen: 'You're good to go.' Provide explicit staff-friendly confirmation token/QR.

Hard Rules

- No staff typing. This route must support self-serve completion.
- No waiver = no eligible check-in. Enforcement is server-side (RLS/RPC).
- All money flows are server-side only. Client never writes ledger.

4.7 Screen Spec — Book Class + Smart Booking Fork

Route: /app/book/class/{class_session_id}

Trigger Condition (Locked)

- Membership exists but is not eligible (past_due/paused/canceled/expired).
- Token wallet balance is sufficient for token_cost.
- Tenant rules allow booking via membership OR tokens (common case).

Required UX (Locked)

- Show a blocking bottom-sheet DecisionSheet:
- Option A: Use Tokens (X tokens) — books now (calls canonical RPC book_class_with_tokens).
- Option B: Renew Membership — goes to membership checkout.
- Option C: Update Payment Method — only if status is past_due (Stripe Customer Portal or update card flow).
- Option D: Cancel.

If tokens are insufficient, disable token option and offer renew/update only.

4.8 Screen Spec — Business Check-In (Deterministic Clearance)

Route: /biz/check-in

Staff sees a Member Readiness Card with deterministic 'CLEARED' only when rules pass.

Member Readiness Card must include (Locked)

- Waiver status (signed? active version ok?).

- Membership status (active/comp/past_due/paused/etc.).
- Token balance.
- Today's booking / class session context (if applicable).
- Injury/modification flags (only if user consented).
- Staff notes (tenant_member_notes).

CLEARED Rules (Locked)

- Waiver signed AND
- (membership eligible OR sufficient tokens for booked class OR staff override).
- Staff override requires reason capture and must write audit_log entry.

4.9 Screen Spec — Migration Toolkit (Business Portal)

Routes: /biz/migrate/*

Goal: gyms will not switch systems if migration is painful. Provide a first-class migration workflow.

Minimum UI Steps (Locked)

- Upload CSV (Mindbody export or generic).
- Map columns to canonical fields (members, plans, memberships, token starting balances).
- Preview errors + dry-run summary (counts, conflicts, missing required fields).
- Commit import (server-side; idempotent).
- Send invites/magic links.
- Cutover: set tenants.system_of_record='ndyra'.

5. Social System (Posts, Feeds, Anti-Negativity, Moderation)

5.1 Post Types (MVP -> Later)

- MVP: Text post, Photo post (1-10 images), Workout share post (timer/class/workout history).
- Later: Video posts, Stories, Live (very later).

5.2 Visibility + Context Model

- Visibility enum values: public, followers, members, club, private, staff.
- author_user_id XOR author_tenant_id (exactly one set).
- tenant_context_id optional; if set, the post is 'inside' that gym space and can be gated members/staff.
- club_id optional; if set, visibility must be club (enforced by DB constraint).

5.3 Reactions System (Encouragement Only)

- Reactions (locked): fire, strong, bolt, clap, mind.
- post_reactions has primary key (post_id, user_id): at most one reaction per user per post.
- Changing reaction is UPDATE of reaction_type.

5.4 Comments + Friction

- Max thread depth = 2 (parent + one level replies).
- First comment ever requires reminder checkbox; store acknowledged timestamp.
- Slow-mode enforcement via user_moderation_state.slow_mode_until (RLS helper can_comment_now).
- Gym option: members-only comments on gym posts.

5.5 Moderation Pipeline (MVP Required)

- Report post/comment/profile/tenant with reason codes.
- Mute user (client-side + DB optional).
- Block user (DB-enforced mutual invisibility).
- Gym-level moderation: tenant staff can hide posts inside their tenant_context_id.
- Platform moderation: platform admins can remove any content; all actions write audit_log.

6. Gyms (Profiles, Memberships, Tokens, Ads, Events)

6.1 Gym Profile (Public + Gated)

- Public view (guest allowed): About, photos, locations, contact, website; public posts preview; schedule preview; CTAs (Follow, Join, Buy Tokens).
- Member view (unlocked): Full schedule with booking; members-only posts; members-only timer templates.
- Staff/Admin view: Member roster + CRM-lite; check-in dashboard; class rosters + attendance + waitlist; gym moderation tools.

6.2 Tenant System-of-Record Flag (Cutover Control)

Add to tenants:

```
system_of_record text not null default 'external'  -- values: external|ndyra
```

When system_of_record='ndyra':

- Booking/check-in enforcement relies solely on NDYRA membership + tokens + waiver state.
- External systems are not authoritative.

When system_of_record='external':

- NDYRA may show public schedule preview and social surfaces.
- Business check-in and booking surfaces must be disabled or clearly marked as non-authoritative (no partial enforcement).
- Cutover to 'ndyra' happens only through /biz/migrate/cutover and must be auditable.

6.2.1 Tenant Kill Switch Flags (Launch-Day Safety)

Add simple tenant-level kill switches so you can instantly pause high-risk operations without redeploying.

- DB: add boolean flags on tenants (default FALSE): kill_switch_disable_booking, kill_switch_disable_checkin, kill_switch_disable_migration_commit.
- Server-side enforcement only: every booking RPC / check-in action / migration commit step MUST reject when the relevant kill switch is true (return deterministic error codes).
- Logging: every kill-switch denial writes an audit_log entry (tenant_id, user_id, action = 'kill_switch_block', reason, request_id).
- UI: show an explicit “Temporarily disabled” state (never silent failure) and provide a deterministic next step (contact, retry later, or staff override if allowed).

6.3 Membership Plans + Membership Status Enforcement (Fix legacy failures)

Blueprint previously referenced gym_memberships with status active/paused/canceled. This is expanded to support deterministic operational enforcement.

6.3.1 Membership Status Enum (Required)

- active
- past_due
- paused
- canceled
- comp
- expired (optional if derived; enforcement must behave as if expired exists)

Enforcement rule (Locked):

If status is not active or comp, the member is not eligible for membership-based booking/check-in. No UI-only enforcement.

6.3.2 Staff Override (Allowed but Auditable)

- Staff override is allowed at check-in only, and must capture: reason, staff_user_id, timestamp.
- Override must write an audit_log entry.
- Override must be scoped to tenant and (optionally) class session/day; do not create permanent silent bypasses.

6.4 Tokens (ClassPass-like but Tenant-Aware)

- Tokens stored per tenant in token_wallets (tenant_id, user_id, balance).
- All token balance changes are derived from token_transactions ledger (immutable).
- Bookings that require tokens call server-side RPC spend_tokens (transactional).
- Refunds/cancellations create offsetting token_transactions rows; never delete ledger rows.

6.5 Ads / Boosted Posts (Later, Approved)

Gyms can boost posts to local audiences; boosted content is labeled Sponsored. (Not part of this amendment.)

6.6 Events Network (Group56)

Gyms can host local events and optionally affiliate them with Group56. (Not part of this amendment.)

7. Scheduling + Booking + Check-In + Payroll Export

7.1 Class Scheduling (Tenant)

- class_types: defines class name, default token cost, description.
- class_sessions: defines date/time, coach, capacity, visibility (public/members), token_cost override.
- class_bookings: user booking state (booked/canceled/attended/no_show/waitlist).

7.2 Booking Rules (Hard)

- Capacity must be enforced transactionally (no oversells).
- Waitlist auto-promotion when a spot opens.
- Cancellation cutoff per tenant (default 2h).
- Token spend/refund rules must be deterministic and auditable (RPC + single transaction).
- Waiver enforcement is required before first eligible booking/check-in.

7.3 Waiver Enforcement (Required)

Before a user is eligible for their first check-in, the user must have a valid waiver signature for the tenant's active waiver version. No 'let them in and fix later'.

Recommended implementation (locked pattern):

- Expose tenant waiver template management to tenant staff via /biz/settings or /biz/migrate (server-side function waiver-template-update).
- Store waiver templates append-only with versioning; store waiver signatures append-only with waiver_version + signature_storage_path. Waiver signatures are created via RPC sign_current_waiver (no client inserts).
- Enforce waiver requirement server-side in both booking and check-in paths (RLS helper has_signed_current_waiver(tid, uid) OR inside transactional RPCs).

7.4 Booking Eligibility (Membership vs Tokens) (Locked)

Eligibility is determined server-side using membership status + waiver status + token wallet balance. UI must reflect server truth; it must not be the enforcement layer.

Precondition: booking and check-in enforcement is only authoritative when tenants.system_of_record = 'ndyra'. If system_of_record = 'external', booking/check-in endpoints must reject (and UI must disable/label as non-authoritative).

Membership-based eligibility

Eligible if gym_memberships.status in ('active','comp') AND waiver requirement satisfied.

Token-based eligibility

Eligible if token_wallets.balance >= class_sessions.token_cost (>0) AND waiver requirement satisfied AND class_sessions.visibility = 'public' (tokens never allowed for members-only sessions).

Ineligible statuses

If membership status is past_due/paused/canceled/expired, membership-based booking/check-in is blocked until renewed/reactivated (unless staff override at check-in).

7.5 Smart Booking Fork (Tokens vs Renew Membership)

This extends the existing /app/book/class/{class_session_id} flow without introducing new patterns.

Trigger

- membership exists but not eligible (past_due/paused/canceled/expired)
- AND tokens sufficient for token_cost
- AND tenant rules allow membership OR tokens

Required Backend Support (Locked)

Implement/standardize the transactional wrapper RPC as the canonical path:

```
book_class_with_tokens(p_class_session_id uuid)
returns (booking_id uuid, remaining_balance int)
```

Behavior (single DB transaction):

- FOR UPDATE lock the class_session row to prevent oversells.
- Validate capacity and booking rules.
- Create booking row (class_bookings).
- Call spend_tokens() inside same transaction (ledger write).
- Return booking_id + remaining_balance.

Renew membership path

- Use existing Stripe checkout flows (server-side, idempotent webhooks).
- After webhook confirms subscription/payment, set gym_memberships.status='active'.
- Optionally auto-return to booking and reattempt booking.

Update payment method path

- Provide tenant-specific Stripe Customer Portal link OR NDYRA-managed 'update card' flow.
- Only show when membership status is past_due.

7.6 Business Check-In (Deterministic)

Check-in must be deterministic: staff should never wonder if someone is allowed in. This fixes the legacy 'payment failed but allowed' bug.

Staff UI must show

- Waiver status (signed? correct version?).

- Membership status (active/comp/past_due/paused/canceled/expired).
- Token balance.
- CLEARED indicator only when deterministic rules pass.

CLEARED rules

- waiver signed AND
- (membership eligible OR sufficient tokens for booked class OR staff override)

Staff override

- Requires reason capture.
- Writes audit_log entry and (recommended) checkin_overrides row.
- Override should be scoped to tenant and to the check-in context.

7.7 Payroll Export (MVP)

- Track classes coached + attendance count.
- Track private sessions completed.
- Apply pay rules (flat rate, per head, per session).
- Export CSV per pay period.

8. Clubs (Groups + Goals + Accountability)

Clubs are opt-in groups with a shared training identity and goals. Clubs can be global or gym-affiliated. They provide a safe accountability loop without turning NDYRA into an argument platform.

8.1 Club Core

- club_poster (main image), club_name, description.
- club_goals (weekly minutes, sessions, streak).
- club_membership (roles: owner/mod/member).
- club feed: posts with visibility=club and club_id set.
- Club leaderboard (optional later).

9. Biometrics (Optional, Privacy-First)

Biometrics are an optional layer: weight, body fat %, resting heart rate, VO2 max estimates, mobility snapshots. Biometric data is private by default and can be shared selectively.

9.1 Rules

- Biometrics are never required for core app usage.
- Default privacy: only the user can read biometric history.
- Any sharing is explicit and revocable.
- Device integrations (Apple Health / Google Fit) are future scope.

10. Payments + Ledger (Stripe-Backed, Server-Side Only)

10.1 Payment Principles (Locked)

- All payment initiation and webhook handling is server-side (Netlify Functions).
- All ledger writes are server-side (service role). The client never writes token_transactions or membership payments.
- All webhook handlers must be idempotent (use event.id dedupe table).
- All money-impacting actions write audit_log entries.
- Never mark membership active until Stripe confirms payment (webhook).

10.2 Membership Checkout

- Membership plans map to Stripe Products/Prices.
- Checkout created via /netlify/functions/stripe-create-checkout (existing blueprint function).
- On webhook confirmation: create/update gym_memberships row, set status active, store stripe subscription ids.
- On payment failure: set status past_due; block membership-based booking/check-in (server-side enforcement).

10.3 Token Pack Checkout

- Token packs map to Stripe Products/Prices.
- On webhook confirmation: insert token_transactions credit entry and update token_wallets balance transactionally.
- Refunds produce offsetting debit entry; never delete ledger.

10.4 Past Due Recovery (Update Payment Method)

- When membership status is past_due, surface 'Update Payment Method' option (Smart Booking Fork and Business Portal).
- Use Stripe Customer Portal when possible (preferred).
- Alternative: NDYRA-managed update-card flow using SetupIntent (server-side).

10.5 Audit Logging (Required)

- Membership status changes (active <-> past_due <-> canceled) are logged.
- Token ledger mutations are logged with actor (user or staff) and server request id.
- Staff check-in overrides are logged (reason required).
- Migration batches are logged (import_batch_id).

11. Data Model (Public Schema, RLS Everywhere)

11.1 General Rules

- Every table has: id, created_at, updated_at (as applicable), and tenant_id where multi-tenant.
- RLS enabled on every table. Default deny. Policies written explicitly.
- Indexes required for every hot path; avoid excessive indexes; validate with EXPLAIN.
- All new tables must be added to Anti-Drift Audit allowlist where appropriate.

11.2 Core Entities (High Level)

Entity	**Purpose**
profiles	User profile + identity + location.
tenants	Gyms / studios. Includes system_of_record flag.
tenant_users	Staff/admin role assignments.
gym_membership_plans	Membership plan catalog per tenant.
gym_memberships	User <-> tenant membership status (enforced).
token_wallets	Token balance per tenant/user.
token_transactions	Immutable token ledger per tenant/user.
class_types	Class definitions per tenant.
class_sessions	Scheduled class occurrences.
class_bookings	Bookings/waitlist per class session.
attendance	Check-in/attendance marks.
waiver_templates	Versioned waiver templates per tenant.
waiver_signatures	User waiver signatures per tenant/template/version.
migration_batches	Import batches for idempotent migration.
checkin_overrides	Explicit override record for eligibility exceptions.

11.3 Tenant System-of-Record Field

tenants.system_of_record values: external | ndyra. Default external.

11.4 Membership Status Model

gym_memberships.status must support: active, past_due, paused, canceled, comp, expired.

Eligibility helper (used in RLS and business logic): membership is eligible only when status in ('active','comp').

11.5 Waiver Tables (New, Required)

waiver_templates

- id uuid pk
- tenant_id uuid not null
- title text not null
- body_html text not null
- version int not null
- is_active boolean not null default false
- created_at timestamptz not null default now()

waiver_signatures

- id uuid pk
- tenant_id uuid not null
- user_id uuid not null
- waiver_template_id uuid not null
- waiver_version int not null
- signed_at timestamptz not null default now()
- signature_storage_path text not null
- ip_address text
- user_agent text
- created_at timestamptz not null default now()

Storage bucket:

- Bucket: waiver-signatures
- Policies: user can read own signature; tenant staff can read signatures for their tenant; no public reads.

11.6 Migration Tables (Recommended)

migration_batches

- id uuid pk (import_batch_id)
- tenant_id uuid not null
- source text not null (e.g., 'mindbody', 'generic_csv')
- mode text not null ('dry_run'|'commit')
- status text not null ('running'|'completed'|'failed')
- stats_json jsonb not null default '{}'::jsonb
- created_by uuid (staff user id)
- created_at timestamptz not null default now()

Idempotency rule: the same CSV can be run twice with the same import_batch_id without duplicates.

11.7 Check-In Overrides (Recommended)

checkin_overrides

- id uuid pk
- tenant_id uuid not null
- user_id uuid not null (member being overridden)
- staff_user_id uuid not null
- class_session_id uuid (optional)
- reason text not null
- created_at timestamptz not null default now()

11.8 Required Helper Functions (RLS-Friendly)

- is_platform_admin()
- is_tenant_staff(tid)
- is_tenant_member(tid) — must treat status active OR comp as eligible.
- has_signed_current_waiver(tid, uid) — true only when signature exists for tenant active waiver version.
- can_book_class(session_id, uid) — optional helper; can be implemented via RPC for complex rules.
- can_comment_now(uid) — slow-mode + shadow-ban check (Section 13.9).

11.9 Index Requirements (Hot Paths)

Social indexes are in Section 13.7. Additional hot path indexes for gym ops:

- gym_memberships (tenant_id, user_id) unique; and (user_id, status) index for quick eligibility checks.
- token_wallets (tenant_id, user_id) unique; index on (user_id) for wallet UI.
- waiver_templates (tenant_id, is_active) partial unique where is_active=true (only one active per tenant).
- waiver_signatures unique (tenant_id, user_id, waiver_version) for idempotent signing + readiness checks.
- class_sessions (tenant_id, starts_at) index for schedule views.
- class_bookings index (class_session_id, status) for roster views; plus partial unique (class_session_id, user_id) where status='booked' to prevent duplicate active bookings.
- token_transactions partial unique (tenant_id, user_id, ref_id) where ref_type='class_booking' and kind='spend' (prevents accidental double-spend).
- attendance (tenant_id, class_session_id) index for payroll exports.

12. API Blueprint (Netlify Functions + RPC, No New Patterns)

12.1 Rules (Locked)

- All money/ledger writes are service-role only.
- All CSV migration imports are service-role only.
- All waiver template updates are tenant-staff only and must bump version and activate via server-side function.
- All waiver signature writes are server-side only via RPC sign_current_waiver(). Clients must not insert directly into waiver_signatures.
- All check-in overrides are staff only and must be auditable.
- Use ES modules for serverless functions. No new framework.
- All serverless functions must validate auth token, tenant role, and inputs. All must be idempotent where needed.

12.2 Required Netlify Functions (v7.3)

Function	**Purpose**
stripe-create-checkout	Create Stripe checkout session for membership or token packs (existing).
stripe-webhook	Handle Stripe events; idempotent; writes membership statuses and ledger (existing).
tenant-migration-import	CSV import with mapping/dry-run/commit; idempotent with import_batch_id (NEW).
waiver-template-update	Tenant staff only; bumps waiver version; sets active; writes audit_log (NEW).
checkin-override	Staff-only; creates override record + audit_log; returns updated readiness state (NEW).

12.3 RPCs (Database)

- spend_tokens(tenant_id, user_id, amount, ref_type, ref_id) — existing canonical ledger spend (server-side).
- refund_tokens(...) — existing refund path (server-side).
- sign_current_waiver(tenant_id, signature_storage_path) — NEW canonical waiver signing finalizer (verifies signature object exists; writes waiver_signatures; idempotent per user+version).
- book_class_with_tokens(class_session_id) — NEW canonical wrapper: locks capacity + creates booking + spends tokens in one transaction.
- get_following_feed(limit, cursor_created_at, cursor_id) — recommended scale-safe feed RPC (Section 13.6).
- can_comment_now(user_id) — helper for slow-mode/shadow-ban (Section 13.9).

12.4 tenant-migration-import Contract (Locked)

Path: /.netlify/functions/tenant-migration-import

Inputs

- tenant_id (must match staff user's tenant)
- import_batch_id (uuid) — used for idempotency and audit
- mode: dry_run | commit
- entity: members | schedule (members required; schedule optional)
- file: CSV content (multipart or uploaded to temp storage then referenced)

Requirements

- Idempotent: running the same CSV twice with the same import_batch_id must not create duplicates.
- Matches/creates users by normalized email (canonical key).
- Upserts memberships + wallets (wallet via ledger-style import credit).
- Writes audit_log entry for the batch and stores summary in migration_batches.stats_json.
- No client direct writes.

Entities Imported (MVP Required)

- Members (email canonical key).
- Membership plans.
- Memberships (plan + status + dates).
- Token starting balances (if any).
- Optional: class schedule (class types + upcoming sessions).
- Optional: staff list.

12.5 waiver-template-update Contract (Locked)

Path: /.netlify/functions/waiver-template-update

- Tenant staff/admin only.
- Creates a new waiver_templates row with version=prev+1 and is_active=true; sets previous active to false.
- Writes audit_log entry with old/new version numbers.
- Never edits historical waiver templates (immutability).

12.6 checkin-override Contract (Locked)

Path: /.netlify/functions/checkin-override

- Tenant staff/admin only.
- Requires: tenant_id, user_id, reason, optional class_session_id.
- Creates checkin_overrides row and audit_log entry in the same request.
- Returns updated readiness evaluation for the Member Readiness Card.

13. Scaling + Performance + Abuse Protection (Scale-Ready)

13.1 What Scales Easily vs Pain Points

- Scales easily: Static pages via Netlify CDN; Supabase Storage for images (with proper caching).
- Hard parts: feed ranking queries at high QPS; counts on hot posts; abuse/spam waves; notification fan-out; one-to-many announcements.

13.2 MVP Scaling Strategy (Base)

- Use post_stats to avoid COUNT() on every render.
- Index posts for seek pagination and author slices (created_at desc, author_user_id + created_at desc, author_tenant_id + created_at desc, tenant_context_id + created_at desc).
- Index post_reactions and maintain counts in post_stats via triggers.
- Seek pagination only; never OFFSET.

13.3 Abuse Protection (Base)

- Edge rate limiting for all /api/* endpoints (Netlify).
- DB constraints: one reaction per user per post; unique follow rows; prevent spam duplicates.
- Account age limits: new accounts limited to X posts/day until verified (store in user_moderation_state).
- Shadow bans: hide content from everyone except the author while allowing them to keep using the app (reduces escalation).

13.4 Capacity Targets + SLOs (Locked)

The platform is considered Scale-Ready when:

- 100 active tenants can be onboarded without query-plan changes.
- 20,000 registered users can be supported without introducing COUNT(*) hot paths or OFFSET pagination.
- Peak concurrency of ~500 sessions does not exhaust DB connections or cause feed latency regressions.

SLOs (service level objectives):

- Feed read: p95 < 400 ms; p99 < 1,000 ms (DB + API layer).
- Post detail (post + first 50 comments): p95 < 400 ms.
- Create post: 99% of successful uploads produce a fully viewable post (no broken media references).
- Reaction/comment writes: p95 < 300 ms under nominal load.
- No private media leaks: storage + RLS validated with negative tests at every release gate.
- Read path error rate < 1%; write path error rate < 0.5%.

13.5 Data Growth Model + Guardrails

- Seek pagination everywhere (created_at + id cursor). Never OFFSET in feeds.

- Hot-path counts must come from post_stats (or other cached stats tables), not COUNT(*).
- Post/media payload control: enforce max images per post; enforce max image dimensions/bytes at upload.
- Avoid large IN(...) lists on the client when follow graph grows; prefer join-based server-side queries (RPC) once thresholds are exceeded (see 13.6).

13.6 Feed Query Plan Hardening

Hard rules:

- All feed queries must be explainable with index-backed plans (EXPLAIN ANALYZE on staging).
- No N+1 fetch patterns for PostCard rendering; use the canonical select shape.
- Do not pull follow graphs into the client if it creates unbounded IN lists.

Scale-safe approach (recommended):

- Keep existing client-side merge for early pilot, but define a hard threshold: if followedUserIds or followedTenantIds exceed 500 total, switch Following feed to server-side feed RPCs.
- Add RPCs that return feed posts using join-based filtering (follows_users / follows_tenants / gym_memberships) to avoid IN-list bloat.

RPC scaffold (copy/paste, refine as needed):

```
create or replace function public.get_following_feed(
  p_limit int default 25,
  p_cursor_created_at timestamptz default null,
  p_cursor_id uuid default null
)
returns setof public.posts
language sql stable
-- SECURITY INVOKER (do NOT set SECURITY DEFINER; posts RLS must apply).
set search_path = public
as $$
with me as (select auth.uid() as uid),
base as (
  select p.*
  from public.posts p
  join me on true
  left join public.follows_users fu
    on fu.follower_id = me.uid and fu.followee_id = p.author_user_id
  left join public.follows_tenants ft
    on ft.follower_id = me.uid and ft.tenant_id = p.author_tenant_id
  where (fu.follower_id is not null or ft.follower_id is not null)
```

```
    and (p_cursor_created_at is null
      or (p.created_at, p.id) < (p_cursor_created_at, p_cursor_id))
  order by p.created_at desc, p.id desc
  limit least(p_limit, 50)
)
select * from base;
$$;
```

Locked: get_following_feed MUST be SECURITY INVOKER (no SECURITY DEFINER). Posts RLS (using can_view_post) must remain the enforcement layer. Do not ship a SECURITY DEFINER feed RPC, or you risk bypassing visibility and leaking private posts.

13.7 Index Manifest (Must Exist Before 20k Users)

All indexes below are required OR must be explicitly replaced by a documented alternative. Validate with EXPLAIN (ANALYZE, BUFFERS) on staging.

```
-- Posts (feeds + profiles)
create index if not exists posts_created_id_idx
  on public.posts (created_at desc, id desc);
create index if not exists posts_author_user_created_id_idx
  on public.posts (author_user_id, created_at desc, id desc);
create index if not exists posts_author_tenant_created_id_idx
  on public.posts (author_tenant_id, created_at desc, id desc);
create index if not exists posts_tenant_ctx_created_id_idx
  on public.posts (tenant_context_id, created_at desc, id desc);
-- Optional: helps public-only discovery slices
create index if not exists posts_visibility_created_id_idx
  on public.posts (visibility, created_at desc, id desc);

-- post_stats (trending)
create index if not exists post_stats_score_48h_idx
  on public.post_stats (score_48h desc);
create index if not exists post_stats_last_engaged_idx
  on public.post_stats (last_engaged_at desc);

-- post_comments (already required in blueprint; completeness)
create index if not exists post_comments_post_created_idx
  on public.post_comments (post_id, created_at asc);

-- post_reactions
create index if not exists post_reactions_post_created_idx
  on public.post_reactions (post_id, created_at desc);
```

```
-- tenant_locations (local discovery)
create index if not exists tenant_locations_city_idx
  on public.tenant_locations (city, tenant_id);

-- notifications
create index if not exists notifications_user_created_idx
  on public.notifications (user_id, created_at desc);

-- If post_media has sort_order (CP31):
-- create index if not exists post_media_post_sort_idx
--   on public.post_media (post_id, sort_order asc);
```

13.8 Storage + Media Performance Defaults

- Always upload direct-to-Supabase Storage (still required).
- Set cacheControl on uploads. Public content can be cached aggressively; private content must use short-lived signed URLs.
- Enforce upload constraints: max dimensions and max bytes per image (client-side), and store width/height metadata for layout stability.
- Lazy-load media in feeds and prefer a single visible media tile; load additional carousel items on demand.

Recommended Storage upload options (supabase-js):

```
await supabase.storage.from('post-media').upload(path, file, {
  upsert: false,
  cacheControl: '31536000', // 1 year (public posts only)
  contentType: file.type
});
```

For non-public posts, do not rely on a permanently public URL. Use signed URLs with short TTL and ensure bucket/object policies enforce visibility.

13.9 Abuse Protection (Scale Requirement)

At 20k users, abuse becomes a throughput problem. Minimum required controls:

- Rate limit high-impact serverless endpoints (/api/*) at the edge (Netlify).
- DB-level uniqueness constraints already present (one reaction per user per post; unique follow edges).
- Slow-mode enforcement via user_moderation_state.slow_mode_until for comments (RLS check).
- Shadow bans must hide content from everyone except the author (enforced in can_view_post).

RLS helper function for comments (copy/paste, refine as needed):

```
create or replace function public.can_comment_now(p_user_id uuid)
returns boolean
language sql stable
security definer
set search_path = public
as $$
select coalesce((
  select (ums.shadow_banned_at is null)
     and (ums.slow_mode_until is null or ums.slow_mode_until < now())
  from public.user_moderation_state ums
  where ums.user_id = p_user_id
), true);
$$;
```

13.10 Notification Fan-out Strategy (Don’t Melt the DB)

- One-to-one notifications (reaction/comment to post author) are safe at this scale.
- One-to-many fan-out (gym announcements to thousands of members) must be asynchronous.
- Do not insert 10k+ notifications in a single request/trigger.
- Use an outbox table (notification_jobs) and a scheduled worker to fan-out in batches, OR a broadcast model where clients materialize announcements without per-user rows.
- Web Push sends must be queued/batched; never done inline with the post creation request.

13.11 Observability for Scale

- Client telemetry events (feed_open, post_impression, reaction_set, comment_create) with sampling.
- Error tracking (JS + serverless) capturing route, release, and anonymized user id.
- DB: slow query logging enabled; dashboards for CPU, connections, cache hit rate.
- Release tagging: every deploy has a build version surfaced in /site/assets/build.json and sent with telemetry.

13.12 Load Testing + Release Gates

Scaling work is not real until it passes repeatable load tests. Add a load test harness and run it in CI against staging.

Minimum scenarios

- Feed read: 25 posts page, 25-50 RPS for 10 minutes, with realistic auth sessions.
- Hot post: reaction writes at 5-10 RPS with concurrent feed reads.
- Post detail: load post + first 50 comments at 10-20 RPS.

Gate thresholds (initial, adjust with real telemetry)

- No error rate > 1% on read scenarios.
- Feed p95 < 400 ms; post detail p95 < 400 ms.
- DB CPU and connection count remain within safe headroom (no sustained saturation).

13.13 Scale Implementation Checklist (Add to Build Order Manifest)

Execute in this order to avoid drift:

Step	What	Done When
SG-1	Scaling migration (indexes + helper functions)	Indexes in 13.7 exist in staging/prod; Anti-Drift Audit + RLS tests updated and passing.
SG-2	Storage caching defaults	Uploads set cacheControl; private media uses signed URLs; negative leak tests pass.
SG-3	Telemetry + error tracking	Events emit with release version; errors captured with route + build id.
SG-4	Load test harness + CI gate	Load tests run against staging on every merge to main; thresholds enforced.
SG-5	Scale rehearsal	Synthetic dataset approximating 100 tenants; load tests + query plan review recorded in /docs/adr/.

13.14 Scale-Ready Acceptance Checklist

- RLS audit passes (no allow-all policies; post-adjacent tables gated by can_view_post).
- All indexes in 13.7 exist in production and are verified by EXPLAIN plans on staging.
- Load tests pass the thresholds in 13.12 on staging with a realistic dataset.
- Media uploads set cache headers; private media is not publicly readable by URL.
- No N+1 regressions: PostCard renders with one data fetch per page of posts.
- Incident runbook exists (how to triage slow feed, connection exhaustion, spike in errors).

13.15 “No Updates Until 20k Users” Reality Check

Even with a scale-safe architecture, you still ship updates for security and platform churn. To minimize surprise work, adopt an ops cadence:

- Monthly dependency and security audit (especially supabase-js and serverless dependencies).

- Quarterly iOS/Android/Safari/Chrome compatibility sweep against the OS support matrix.
- Continuous monitoring of DB performance and function error rates; fix regressions before they become outages.

14. Observability + Analytics + Audit

14.1 Audit Log (Required)

- audit_log captures administrative and ledger actions: who, what, target, before/after JSON, timestamp, ip/user_agent.
- Every moderation takedown, ban, token adjustment, membership status override, and system-of-record cutover must create an audit_log row.
- Migration imports write a batch audit entry with import_batch_id and summary stats.

14.2 Telemetry Events (Client)

- feed_open, post_impression, reaction_set, comment_create
- booking_create, token_purchase, checkin_success, waiver_signed, migration_invite_sent
- Events are POSTed to a telemetry endpoint with sampling to control cost.
- Every event includes: build version (from /site/assets/build.json), route, and anonymized user id.

14.3 Monitoring

- Supabase: enable slow query logs; dashboards for CPU, connections, cache hit rate.
- Netlify: monitor function error rate and cold starts; track rate-limit events.
- Stripe: monitor webhook retries and failures (idempotency issues show here).
- Error tracking: Sentry (or equivalent) for JS + serverless; capture release version, user_id (hashed), route.
- Trace correlation: emit trace IDs from frontend requests and propagate through Netlify Functions.
- Uptime: external synthetic monitoring for /api/health + key pages (login, feed). Alert on latency spikes and auth failures.
- Security monitoring: alert on unusual auth attempts, spikes in reports/spam flags, repeated migration import failures.

15. Dev Workflow (Environments, Migrations, Testing, Release)

15.1 Environments (Required)

- Local dev (optional): Supabase local via CLI.
- Staging: separate Supabase project + Stripe test mode.
- Production: locked Supabase project + Stripe live mode.

15.2 Secrets + Env Vars (Netlify)

- SUPABASE_URL
- SUPABASE_ANON_KEY
- SUPABASE_SERVICE_ROLE_KEY (functions only)
- STRIPE_SECRET_KEY
- STRIPE_WEBHOOK_SECRET
- TELEMETRY_INGEST_KEY

15.3 Database Migrations

- All schema changes must be migrations committed to repo (no manual prod edits).
- Migrations must include: DDL + RLS policies + indexes.
- Every migration must have a rollback plan (at least documented).
- After every migration: run Anti-Drift Audit + RLS tests.

15.4 Testing (Merge Blockers)

RLS tests

- Verify forbidden access using anon key and random users.
- Validate post-adjacent tables require can_view_post.
- Validate waiver tables: users read own signature only; staff read tenant signatures only; no public reads.
- Validate membership enforcement: status past_due/paused/canceled/expired blocks membership-based booking/check-in server-side.

Unit tests

- Scoring functions and helper utilities.
- Ledger integrity: spend/refund invariants.

E2E tests (Playwright)

- Onboarding + create post + react + comment + follow.
- Quick Join: /gym/{slug}/join → account → waiver → payment (mock) → confirmation.
- Booking fork: membership inactive + tokens sufficient → choose tokens → booking succeeds.
- Booking fork: past_due → update payment option present.

- Check-in: waiver missing blocks.
- Check-in: past_due blocks unless override.
- CSV import: 50 members, mixed statuses → verify counts.
- CSV import idempotency: rerun → no duplicates.

Load tests (Scale Gate)

- Run Section 13.12 load scenarios against staging on every merge to main.
- Fail the build if thresholds regress.

15.5 2026 Engineering Standards (Tooling + Security + Cross-Platform)

- Type safety: migrate new modules to TypeScript (or JS + JSDoc type checking). Enforce strict mode for all new code.
- Bundling: Vite recommended for module bundling, code-splitting, asset hashing; keep static HTML routes but compile JS modules.
- Code quality: ESLint + Prettier + EditorConfig; pre-commit hooks to block broken formatting/tests.
- CI/CD: lint, unit tests, RLS smoke tests (staging), Playwright E2E; Netlify preview deploy per PR.
- Security headers: CSP, HSTS, X-Content-Type-Options, Referrer-Policy, Permissions-Policy at CDN edge; lock allowed media origins.
- Dependency hygiene: monthly dependency audit; pin major versions; SBOM generation later.
- Privacy compliance ops: data export + deletion workflows; maintain store privacy disclosures as features ship.

16. Checkpoint Roadmap + Acceptance Criteria

16.1 Principle

Checkpoints are the execution gates. A checkpoint is not 'done' until acceptance criteria and anti-drift gates pass.

16.2 CP31 (Done) — Post Detail + Create Composer + E2E Harness Polish

- Post detail (/app/post/:id): comments thread + sticky composer + reply pill + cancel.
- Post detail reactions work (encouragement-only).
- Create (/app/create): text (300 char max) + optional media upload (up to 10).
- Uploads to Supabase Storage bucket post-media; writes post_media rows (storage_path, media_type, sort_order).
- Local E2E harness: mjs MIME + Netlify-style /app/post/:id fallback routing; Playwright HTML report.

Acceptance gates:

- Playwright smoke: open post detail; add comment (demo mode ok); reply; react; create text post; create media post.
- Storage policies verified (no private leak).
- Anti-Drift Audit passes.

16.3 CP32 (Next) — Profile + Notifications + Moderation UX + E2E

- Profile page (own + other): posts grid, follow/unfollow, followers/following lists.
- Notifications page (likes/comments/follows): list + unread/mark seen.
- Moderation UX scaffolding: hide post, report post, block user (no new tables unless blueprint says).
- E2E: add smoke tests for /app/post/:id and /app/create and include in gates.

16.4 Gym Ops System-of-Record Milestone (Pre-Scale Gate)

Before onboarding gyms at scale, NDYRA must be 'easier than Mindbody' at the front desk. The following must be complete and gated by tests:

- Migration Toolkit MVP (CSV import + mapping UI + idempotent server-side import + invites).
- Quick Join (/gym/{slug}/join) + Digital Waivers (versioned templates + signatures + enforceable policies).
- Deterministic Business Check-In (readiness card + waiver + membership status + tokens + auditable overrides).
- Smart Booking Fork (tokens vs renew membership; canonical RPC).
- Membership status enforcement (active/comp only; past_due blocks; no UI-only enforcement).

16.5 Proposed Sequencing (Draft, Adjust as Needed)

Checkpoint	Theme	Delivers

CP33	Gym foundations	tenants.system_of_record; membership status enum; membership plan plumbing; token wallets; readiness evaluation helper; initial business check-in shell.
CP34	Migration + Waivers + Quick Join	Migration toolkit routes + tenant-migration-import; waiver templates/signatures + policies; /gym/{slug}/join flow; waiver enforcement in check-in.
CP35	Booking fork + Determinism + Exports + Scale gate	book_class_with_tokens RPC; Smart Booking Fork UI; deterministic clearance + override function; payroll export CSV; scale rehearsal + load test gate.

16.6 Acceptance Criteria (System-of-Record)

- A walk-in can self-onboard via QR: account + waiver + payment (mock) + confirmation in under 90 seconds on a phone.
- A member with past_due status cannot check in unless staff override is recorded with reason (auditable).
- A member with inactive membership but sufficient tokens can book via token path with one tap (Smart Booking Fork).
- Migration import can run twice without duplicates (idempotent).
- RLS tests cover waiver and membership enforcement; no private signature leaks.
- Load tests pass thresholds (Section 13.12) on staging with synthetic dataset approximating 100 tenants.

Appendix A — SQL Patches (v7.3.1 Additions)

These patches extend the existing schema and policies. They are written to follow RLS-first rules and avoid new patterns. Apply in staging first, run Anti-Drift Audit, then production.

```
-- ==========================================================
-- NDYRA v7.3 SQL Patches (Flawless QC)
-- Adds/locks: system_of_record, waiver templates/signatures (secure signing),
migration batches,
-- check-in overrides, membership status expansion, booking-with-tokens RPC, and helper
functions.
-- ==========================================================

-- 1) Tenant system_of_record flag (cutover control)
alter table public.tenants
  add column if not exists system_of_record text not null default 'external';

-- Optional hardening: constrain values
do $$
begin
  if not exists (
    select 1 from pg_constraint where conname = 'tenants_system_of_record_chk'
  ) then
    alter table public.tenants
      add constraint tenants_system_of_record_chk
      check (system_of_record in ('external','ndyra'));
  end if;
end $$;

create index if not exists tenants_system_of_record_idx
  on public.tenants (system_of_record);

-- 2) Membership status model (expand enforcement statuses)
do $$
declare
  v_label text;
begin
  -- Create enum if missing
  if not exists (select 1 from pg_type where typname = 'membership_status') then
    create type public.membership_status as enum
('active','past_due','paused','canceled','comp','expired');
  end if;

  -- Ensure required enum labels exist (safe add)
  foreach v_label in array['active','past_due','paused','canceled','comp','expired']
loop
    if not exists (
      select 1
      from pg_enum e join pg_type t on t.oid = e.enumtypid
      where t.typname = 'membership_status' and e.enumlabel = v_label
    ) then
      execute format('alter type public.membership_status add value %L', v_label);
    end if;
  end loop;

  -- Enforce allowed values on gym_memberships.status (works whether status is text or
enum)
  if to_regclass('public.gym_memberships') is not null
     and exists (
       select 1
       from information_schema.columns
       where table_schema='public' and table_name='gym_memberships' and
column_name='status'
```

```
      )
  then
    if not exists (select 1 from pg_constraint where conname =
'gym_memberships_status_chk') then
      alter table public.gym_memberships
        add constraint gym_memberships_status_chk
        check ((status::text) in
('active','past_due','paused','canceled','comp','expired'));
    end if;
  end if;
end $$;

-- Eligibility helper: treat ACTIVE or COMP as eligible membership
create or replace function public.is_tenant_member(tid uuid)
returns boolean
language sql
stable
security definer
set search_path = public
as $$
  select exists (
    select 1
    from public.gym_memberships gm
    where gm.tenant_id = tid
      and gm.user_id = auth.uid()
      and (gm.status::text) in ('active','comp')
  );
$$;

revoke all on function public.is_tenant_member(uuid) from public;
grant execute on function public.is_tenant_member(uuid) to authenticated;

-- 3) Waiver templates + signatures (RLS enabled, default deny)
create table if not exists public.waiver_templates (
  id uuid primary key default uuid_generate_v4(),
  tenant_id uuid not null references public.tenants(id) on delete cascade,
  title text not null,
  body_html text not null,
  version int not null,
  is_active boolean not null default false,
  created_at timestamptz not null default now()
);

create unique index if not exists waiver_templates_one_active_per_tenant
  on public.waiver_templates (tenant_id)
  where is_active = true;

create unique index if not exists waiver_templates_tenant_version_unique
  on public.waiver_templates (tenant_id, version);

alter table public.waiver_templates enable row level security;

-- Public can read active waiver template (needed for /gym/{slug}/join pre-signature
UX).
drop policy if exists waiver_templates_select_active on public.waiver_templates;
create policy waiver_templates_select_active
  on public.waiver_templates
  for select
  using (
    is_active = true
    or public.is_tenant_staff(tenant_id)
    or public.is_platform_admin()
  );
```

```
-- Only staff/admin can create new templates (writes should normally be via server-side
function).
drop policy if exists waiver_templates_insert_staff on public.waiver_templates;
create policy waiver_templates_insert_staff
  on public.waiver_templates
  for insert
  with check (
    public.is_tenant_staff(tenant_id)
    or public.is_platform_admin()
  );

-- Hard rule: waiver_templates are append-only (no update/delete policies).
drop policy if exists waiver_templates_update_staff on public.waiver_templates;
drop policy if exists waiver_templates_delete_staff on public.waiver_templates;

create table if not exists public.waiver_signatures (
  id uuid primary key default uuid_generate_v4(),
  tenant_id uuid not null references public.tenants(id) on delete cascade,
  user_id uuid not null references auth.users(id) on delete cascade,
  waiver_template_id uuid not null references public.waiver_templates(id) on delete
restrict,
  waiver_version int not null,
  signed_at timestamptz not null default now(),
  signature_storage_path text not null,
  ip_address text,
  user_agent text
);

-- Idempotency: a user signs once per tenant per waiver_version.
create unique index if not exists waiver_signatures_one_per_user_version
  on public.waiver_signatures (tenant_id, user_id, waiver_version);

create index if not exists waiver_signatures_tenant_user_idx
  on public.waiver_signatures (tenant_id, user_id);

alter table public.waiver_signatures enable row level security;

-- Users can read their own signature rows
drop policy if exists waiver_signatures_select_self on public.waiver_signatures;
create policy waiver_signatures_select_self
  on public.waiver_signatures
  for select
  using (auth.uid() = user_id);

-- Tenant staff can read signatures for their tenant
drop policy if exists waiver_signatures_select_staff on public.waiver_signatures;
create policy waiver_signatures_select_staff
  on public.waiver_signatures
  for select
  using (public.is_tenant_staff(tenant_id) or public.is_platform_admin());

-- IMPORTANT: No client insert policy. Signing is finalized via RPC
sign_current_waiver() below.
drop policy if exists waiver_signatures_insert_self on public.waiver_signatures;

-- No update/delete policies (append-only legal record).
drop policy if exists waiver_signatures_update_self on public.waiver_signatures;
drop policy if exists waiver_signatures_delete_self on public.waiver_signatures;

-- Helper: check if a user has signed the current active waiver version
create or replace function public.has_signed_current_waiver(p_tenant_id uuid, p_user_id
uuid)
```

```
returns boolean
language sql
stable
security definer
set search_path = public
as $$
  with active as (
    select id as waiver_template_id, version
    from public.waiver_templates
    where tenant_id = p_tenant_id and is_active = true
    order by version desc
    limit 1
  )
  select exists (
    select 1
    from public.waiver_signatures ws
    join active a on a.waiver_template_id = ws.waiver_template_id and a.version =
ws.waiver_version
    where ws.tenant_id = p_tenant_id
      and ws.user_id = p_user_id
  );
$$;

-- Canonical signing RPC (secure): verifies Storage object exists (prevents spoofing).
-- Expected Storage key format (required):
--   t/{tenant_id}/u/{user_id}/v{waiver_version}/{signature_uuid}.png
create or replace function public.sign_current_waiver(p_tenant_id uuid,
p_signature_storage_path text)
returns table (waiver_signature_id uuid, waiver_version int)
language plpgsql
security definer
set search_path = public, storage
as $$
declare
  v_template_id uuid;
  v_version int;
  v_user_agent text;
begin
  if auth.uid() is null then
    raise exception 'auth_required';
  end if;

  select wt.id, wt.version
    into v_template_id, v_version
  from public.waiver_templates wt
  where wt.tenant_id = p_tenant_id and wt.is_active = true
  order by wt.version desc
  limit 1;

  if v_template_id is null then
    raise exception 'no_active_waiver';
  end if;

  if p_signature_storage_path is null or length(trim(p_signature_storage_path)) = 0
then
    raise exception 'signature_path_required';
  end if;

    -- Required object key format:
  -- t/{tenant_id}/u/{user_id}/v{waiver_version}/{signature_uuid}.png
  if p_signature_storage_path !~ (
    '^t/'||p_tenant_id::text||'/u/'||auth.uid()::text||'/v'||v_version::text||'/[0-9a-
fA-F-]{36}\.png$'
```

```
  ) then
    raise exception 'invalid_signature_path';
  end if;

  -- Verify object exists in Storage (bucket: waiver-signatures)
  if not exists (
    select 1
    from storage.objects o
    where o.bucket_id = 'waiver-signatures'
      and o.name = p_signature_storage_path
  ) then
    raise exception 'signature_object_not_found';
  end if;

  begin
    v_user_agent := (current_setting('request.headers', true)::json->>'user-agent');
  exception when others then
    v_user_agent := null;
  end;

  insert into public.waiver_signatures (
    tenant_id,
    user_id,
    waiver_template_id,
    waiver_version,
    signed_at,
    signature_storage_path,
    ip_address,
    user_agent
  ) values (
    p_tenant_id,
    auth.uid(),
    v_template_id,
    v_version,
    now(),
    p_signature_storage_path,
    inet_client_addr()::text,
    v_user_agent
  )
  on conflict (tenant_id, user_id, waiver_version) do nothing
  returning id into waiver_signature_id;

  if waiver_signature_id is null then
    select ws.id
      into waiver_signature_id
    from public.waiver_signatures ws
    where ws.tenant_id = p_tenant_id
      and ws.user_id = auth.uid()
      and ws.waiver_version = v_version
    order by ws.signed_at desc
    limit 1;
  end if;

  waiver_version := v_version;
  return next;
end;
$$;

revoke all on function public.sign_current_waiver(uuid, text) from public;
grant execute on function public.sign_current_waiver(uuid, text) to authenticated;

-- 4) Migration batches (optional but recommended for idempotency + auditing)
create table if not exists public.migration_batches (
  id uuid primary key default uuid_generate_v4(),
  tenant_id uuid not null references public.tenants(id) on delete cascade,
  import_batch_id text not null,
  source_system text not null default 'mindbody',
  status text not null default 'created', -- created|dry_run|committed|failed
  summary jsonb not null default '{}'::jsonb,
  created_at timestamptz not null default now()
);

create unique index if not exists migration_batches_tenant_batch_unique
  on public.migration_batches (tenant_id, import_batch_id);

alter table public.migration_batches enable row level security;

drop policy if exists migration_batches_select_staff on public.migration_batches;
create policy migration_batches_select_staff
  on public.migration_batches
  for select
  using (public.is_tenant_staff(tenant_id) or public.is_platform_admin());

-- No client inserts/updates (server-side only).
drop policy if exists migration_batches_insert_staff on public.migration_batches;
drop policy if exists migration_batches_update_staff on public.migration_batches;

-- 5) Check-in overrides (auditable staff override)
create table if not exists public.checkin_overrides (
  id uuid primary key default uuid_generate_v4(),
  tenant_id uuid not null references public.tenants(id) on delete cascade,
  user_id uuid not null references auth.users(id) on delete cascade,
  staff_user_id uuid not null references auth.users(id) on delete restrict,
  reason text not null,
  created_at timestamptz not null default now()
);

create index if not exists checkin_overrides_tenant_user_created_idx
  on public.checkin_overrides (tenant_id, user_id, created_at desc);

alter table public.checkin_overrides enable row level security;

drop policy if exists checkin_overrides_select_staff on public.checkin_overrides;
create policy checkin_overrides_select_staff
  on public.checkin_overrides
  for select
  using (public.is_tenant_staff(tenant_id) or public.is_platform_admin());

-- No client inserts (server-side function only).
drop policy if exists checkin_overrides_insert_staff on public.checkin_overrides;

-- 6) Canonical booking RPC: book a class using tokens (transactional, deterministic)
create or replace function public.book_class_with_tokens(p_class_session_id uuid)
returns table (booking_id uuid, remaining_balance int)
language plpgsql
security definer
set search_path = public
as $$
declare
  v_tenant_id uuid;
  v_token_cost int;
  v_capacity int;
  v_visibility text;
  v_starts_at timestamptz;
  v_system_of_record text;
  v_kill_booking boolean;
  v_booked_count int;
begin
  if auth.uid() is null then
    raise exception 'auth_required';
  end if;

  -- Lock the class_session row to serialize capacity checks.
  select cs.tenant_id, cs.token_cost, cs.capacity, cs.visibility, cs.starts_at
    into v_tenant_id, v_token_cost, v_capacity, v_visibility, v_starts_at
  from public.class_sessions cs
  where cs.id = p_class_session_id
  for update;

  if v_tenant_id is null then
    raise exception 'class_session_not_found';
  end if;

  -- System-of-record gate: bookings/check-ins are authoritative only when NDYRA is
SOR.
  select t.system_of_record, t.kill_switch_disable_booking
    into v_system_of_record, v_kill_booking
  from public.tenants t
  where t.id = v_tenant_id;

  if v_system_of_record <> 'ndyra' then
    raise exception 'tenant_not_authoritative';
  end if;

  if v_kill_booking then
    raise exception 'booking_disabled';
  end if;

  -- Token eligibility: token_cost must be set and class must be public (never tokens
for members-only).
  if v_token_cost is null or v_token_cost <= 0 then
    raise exception 'tokens_not_available';
  end if;

  if v_visibility is not null and v_visibility <> 'public' then
    raise exception 'tokens_not_allowed';
  end if;

  if v_starts_at is not null and v_starts_at <= now() then
    raise exception 'class_already_started';
  end if;

  -- Idempotency: if already booked, return existing booking_id without spending again.
  select b.id
    into booking_id
  from public.class_bookings b
  where b.class_session_id = p_class_session_id
    and b.user_id = auth.uid()
    and b.status = 'booked'
  order by b.created_at desc
  limit 1;

  if booking_id is not null then
    select tw.balance
      into remaining_balance
    from public.token_wallets tw
    where tw.tenant_id = v_tenant_id and tw.user_id = auth.uid();

    return;
  end if;

  -- Waiver enforcement
  if not public.has_signed_current_waiver(v_tenant_id, auth.uid()) then
    raise exception 'waiver_required';
  end if;

  -- Capacity enforcement
  if v_capacity is not null then
    select count(*)
      into v_booked_count
    from public.class_bookings b
    where b.class_session_id = p_class_session_id
      and b.status = 'booked';

    if v_booked_count >= v_capacity then
      raise exception 'class_full';
    end if;
  end if;

  -- Create booking row
  insert into public.class_bookings (tenant_id, class_session_id, user_id, status,
created_at)
  values (v_tenant_id, p_class_session_id, auth.uid(), 'booked', now())
  returning id into booking_id;

  -- Spend tokens in the same DB transaction
  remaining_balance := public.spend_tokens(v_tenant_id, auth.uid(), v_token_cost,
'class_booking', booking_id);

  return;
end;
$$;

revoke all on function public.book_class_with_tokens(uuid) from public;
grant execute on function public.book_class_with_tokens(uuid) to authenticated;
```

Appendix B — Storage Policies Notes (Waiver Signatures)

Create storage bucket: waiver-signatures (private). Policies (conceptual):

- Users can upload to waiver-signatures only using the exact object key format: t/{tenant_id}/u/{user_id}/v{waiver_version}/{signature_uuid}.png (authenticated only).
- Users can read only their own signature objects (t/*/u/{user_id}/...); no cross-user reads.
- Tenant staff should view signatures via server-side signed URLs (preferred). Avoid broad storage read policies unless you can express strict tenant scoping.
- No public reads (bucket stays private).
- If signed URLs are used, keep them short-lived (e.g., minutes) and log issuance in audit_log (staff_id, tenant_id, signature_path, reason).

Appendix C — Verification Checklist (v7.3.1)

- Route Map updated for /gym/{slug}/join and /biz/migrate routes (/members, /schedule, /verify, /commit, /cutover).
- UI Build Spec updated for Quick Join, Booking Fork, Business Check-In readiness, Migration Toolkit.
- RLS enabled on all new tables (waiver_*, migration_batches, checkin_overrides).
- RLS policies reviewed: no allow-all policies; select policies scoped; waiver_signatures has no client insert; high-impact writes are server-side only.
- Storage bucket waiver-signatures exists and cannot be read publicly.
- sign_current_waiver RPC works end-to-end and rejects spoofed signatures (requires existing object in Storage).
- Membership eligibility enforced server-side: past_due blocks membership-based booking/check-in.
- Tenant kill switches verified: booking/check-in/migration commit reject when disabled, with deterministic error codes + audit_log entries.
- book_class_with_tokens RPC passes transactional oversell tests and enforces system_of_record + token eligibility + idempotency (no double-spend).
- Migration import is idempotent with import_batch_id and writes audit_log.
- E2E tests pass: Quick Join, Booking Fork, Check-In enforcement, CSV import idempotency.
- get_following_feed RPC is SECURITY INVOKER (no SECURITY DEFINER) and does not bypass posts visibility/RLS.
- Load tests pass thresholds (Section 13.12) on staging with 100-tenant synthetic dataset.