ChatGPT

Designing a Birdwatching Tour Platform’s User Profile System

Profile Data Architecture

Extended Profile Fields: In addition to the basic fields (email, name, etc.), the profile should include rich personal details to enhance user engagement. Consider adding a **bio** (short self-description), **location** (home city/region for context), **birding interests** (favorite bird families or habitats), **gear** (equipment used), **experience level** (e.g. *Beginner*, *Intermediate*, *Expert*), **social links** (Twitter, Instagram, etc.), and a **profile photo** (stored via Supabase Storage with a URL reference). These fields make profiles more informative and personable. For example, a profile could show “**Bio:** Passionate birder and photographer from Melbourne. **Experience:** 5+ years” to help tour operators and fellow birders know the user’s background at a glance. All these fields can be nullable and user-editable, with sensible defaults (e.g. empty bio) for new users.

User Roles and Profile Types: The platform will have different user roles beyond just regular customers. It’s best to introduce a **role identifier** rather than overloading the `is_admin` flag. For instance, add a `role` field (an enum or text) with values like `'user'`, `'operator'`, `'admin'`. This clarifies permissions and avoids confusion (currently, operators were being flagged with `is_admin=true`, which should be avoided). A suggested role hierarchy is: **User** – regular birder who books tours; **Operator** – tour provider (boat captain, lodge owner, etc.) who can create/manage tours and see bookings for their tours; **Admin** – platform owner with full access. Below is a role breakdown:

Role	Description	Access & Permissions
User	Regular birder using the platform	Browse and book tours; manage own profile
Operator	Tour operator (e.g. guide, provider)	Create/edit tours; view bookings for their tours; use operator dashboard
Admin	Platform administrator (owner)	Full access to all data; manage users, tours, and operators

Implementing this might involve a new column in `profiles` or a separate `user_roles` table linked to `auth.users`. Supabase supports custom JWT claims for roles, which can be set via an Auth hook so that each user’s role is embedded in their token for RLS checks [1]. (Guides who are not platform users can be handled via a separate `guides` table related to tours, since a guide might not log into the system even if they have a profile.)

Achievements, Badges, and Tour History: Rather than bloating the profiles table with every dynamic stat or award, use separate tables for these related records. For example, an `achievements` or `badges` table can list all possible badges (e.g. *First Tour Completed*, *Referral Star*, *500 Species Club*), and a join table `user_badges(user_id, badge_id, earned_at)` can record which badges a user has earned. Similarly, **tour history** can be derived from the existing `reservations` table – there's no need to duplicate this in profile. You can query `reservations` for a user's past tours or create a view for "tours_completed" details. Keeping achievements and history in their own tables follows normalization best-practices, which tend to be better for performance and querying flexibility [2]. Using JSONB to store lists of badges or tours inside the profile row is possible, but would make it harder to query across users and can lead to slower queries on large data [2]. In general, **structured relational tables** are preferable for data like badges or species lists that may be filtered or aggregated later.

Birding Statistics (eBird vs. Platform): We need to model a user's birding stats that come partly from eBird (an external system) and partly from our platform. It's wise to separate these concerns: - **Platform stats:** Fields like `tours_completed`, `referrals_count`, etc., can live in the profile or a related `profile_stats` table. Since these are specific to our app, we control their updates. For instance, `profiles.tours_completed` (or a count in `profile_stats`) can be incremented via a trigger each time a reservation is marked "completed". - **eBird stats:** We might store a user's **life list count** (total species observed per eBird) and perhaps a few other key metrics from eBird (number of checklists, last active date, etc.). We could add columns like `ebird_life_list_count` (int) and `ebird_last_checklist_date` to profiles, or keep a JSONB `ebird_stats` blob for flexibility. However, a simple numeric field for life list is most useful as a credibility indicator. The existing `life_list_range` (like "`100-300`") could be replaced or supplemented by an exact count if available.

Schema Sketch: Below is an updated schema snippet incorporating the above points (using separate tables for some data):

```
-- Profiles table (extends auth.users)
CREATE TABLE profiles (
  id             UUID PRIMARY KEY REFERENCES auth.users(id),
  email          TEXT,        -- could be kept in sync or removed if redundant
  name           TEXT,
  role           TEXT CHECK (role IN ('user','operator','admin')) DEFAULT 'user',
  bio            TEXT,
  location       TEXT,
  experience_level TEXT,
  social_links   JSONB,       -- e.g. {"twitter": "...", "instagram": "..."}
  profile_photo_url TEXT,
  ebird_username   TEXT,
  ebird_life_list_count INTEGER,
  fieldcraft_quiz_passed BOOLEAN DEFAULT FALSE,
  tier           INTEGER DEFAULT 0,      -- 0=New, 1=Verified, 2=Trusted
  trust_score    INTEGER DEFAULT 0,
  tours_completed INTEGER DEFAULT 0,
  referrals_count INTEGER DEFAULT 0,
  -- Internal fields (consider moving to separate table for security):
  strike_count   INTEGER DEFAULT 0,
  is_flagged     BOOLEAN DEFAULT FALSE,
  is_admin       BOOLEAN DEFAULT FALSE   -- (Deprecated by role, could remove
eventually)
);

-- Separate table for sensitive/internal info (if splitting for RLS):
CREATE TABLE profile_private (
  user_id     UUID PRIMARY KEY REFERENCES profiles(id),
  email       TEXT,       -- if we decide to store email separately for privacy
  phone       TEXT,
  strike_count INTEGER,
  is_flagged   BOOLEAN,
  notes        TEXT         -- e.g. admin notes about user
);

-- Badges/Achievements tables
CREATE TABLE badges (
  id SERIAL PRIMARY KEY,
  code TEXT UNIQUE,        -- e.g. "first_tour", "referral_pro"
  name TEXT,
  description TEXT
);
CREATE TABLE user_badges (
  user_id UUID REFERENCES profiles(id),
  badge_id INT REFERENCES badges(id),
  earned_at TIMESTAMPTZ DEFAULT now(),
  PRIMARY KEY(user_id, badge_id)
);

-- (reservations and referrals tables as given, possibly with indexes for
performance)
```

In the above, notice `profile_private` holds `strike_count` and other sensitive fields. This is one approach to handle field-level privacy by **table splitting**: the main `profiles` table can be broadly accessible for public info, while `profile_private` has a stricter RLS policy (only the user themselves and admins/operators see it). Supabase maintainers recommend this pattern for securing certain columns: *"split your table into two and add policies to the table that has the secured columns."* [3] . This way, even if a regular user's query inadvertently tries to select a hidden field, it won't be accessible unless the policy allows it.

eBird Integration Strategy

Integration Level – Options Analysis: eBird provides a public API (v2); however, it's somewhat limited regarding user-specific data (there's no OAuth or full profile data endpoint). We have a few approaches: - **(a) Minimal – store username & link out:** Easiest route is to simply save the user's `ebird_username` and perhaps display a link to their eBird profile. This requires no complex sync and respects privacy – if a user doesn't want to share details, they don't have to. We'd essentially treat eBird as an external profile: show a button "View on eBird" which opens their public profile (if it's public). - **(b) Fetch & cache key stats:** A richer approach is to use the eBird API to pull a few key stats, like the user's life list count (total species) or recent sightings count. Since eBird's API doesn't allow querying a user's entire life list directly (for privacy reasons, user-specific checklist data isn't openly accessible), we might rely on **specific endpoints or workarounds**. One known method is using eBird's **species totals by region**: for example, eBird API can return how many species a user has in a particular region (including "worldwide"). We could periodically fetch the worldwide species count for the user. This would give us an authoritative life list number to display (if the user's profile is public). We would store this in `ebird_life_list_count` and maybe update it weekly or when the user triggers a "Sync". Caching this avoids hitting the API on every profile load. If using the API, each user should use their own API key or we must abide by eBird's terms (they discourage sharing one API key for multiple users' data [4]). - **(c) Deep sync of sightings/badges:** Fully syncing a user's recent eBird activity (individual checklists, locations, etc.) is **not recommended** for our use-case. It would require fetching potentially large data and also intrudes on eBird's domain. Given the platform's purpose (tour booking, not a full birding journal), we likely don't need every checklist or eBird badge. Instead, we focus on the high-level stats that establish credibility and interest.

Recommended Approach: Start with option (a) and a bit of (b). In practice, that means: - **Store the eBird username** in the profile, and maybe a flag that they have "linked" their eBird. - **If the user opts in** (and their eBird data is public), use their username to fetch a couple of stats via API. For example, eBird has an endpoint for total species observed by a user in a region (the "world life list" count). We can call this endpoint when a user links their account or periodically (e.g., a CRON job or on-demand) to update `ebird_life_list_count`. - Display the life list count on their profile (e.g., "**Species seen:** 245"). This acts as a credibility signal that "this person has observed 245 species on eBird," indicating their birding experience. We'll **avoid deep integration** like importing all their sightings or lists, since it's beyond scope and could violate privacy expectations.

Account Verification: Because eBird lacks OAuth, there's no built-in way to prove "user X on our platform is truly user Y on eBird." We have a couple of options: - **Trust but verify lightly:** On linking, we could simply fetch the public profile data for that eBird username. If it exists and is public, we accept it. (If the profile is private or username not found, we inform the user that verification failed or limited data is available.) This isn't 100% foolproof (someone could claim to be someone else), but the incentive to do so is low in this context. - **Manual or code-based verification (optional):** For higher assurance, we could ask the user to perform a specific action on eBird as proof. For example, instruct them to add a code word to their eBird profile's "About Me" section or submit a checklist with a specific note, then our system could check for that. This is probably overkill for MVP. Given that eBird profiles can be private, many serious birders may not want to make them public just to verify. So, **our default approach will be not to force strict verification**. We can include a disclaimer that linking eBird is optional and based on trust. - We might revisit verification if we use eBird data for anything security-sensitive. As of now, it's more for enrichment.

Privacy Considerations with eBird Data: We must respect that some users keep their eBird data private. According to eBird, users can choose to make their profile public or not [5]. If a user's eBird profile or lists are private, our integration should: - Allow them to still link the account (perhaps just to have the username on file), but not pull any stats without permission. - Clearly inform the user that if they want their eBird stats shown on our platform, they need to set their eBird profile to public (or at least the relevant data). Otherwise, we'll just store the username and not display stats. - Provide a toggle in our profile settings like "Show my eBird statistics on my public profile" so the user has control. Default can be conservative (off) to respect privacy, requiring opt-in to display stats.

Use of eBird Data – Utility vs. Gamification: The goal of integrating eBird is primarily **utility and credibility**, not gamification. We're not trying to replicate eBird's leaderboard or give users gaming incentives. Instead: - **Credibility Signal:** A high life list count or certain notable sightings can signal that a user is an experienced birder (e.g. someone with 500+ species is likely very serious, whereas a user with <100 might be a novice). This could feed into our **trust_score or tier logic** (e.g., to become "Trusted" tier, perhaps linking eBird with a substantial life list could be one criterion). - **Personalization & Recommendations:** Knowing a user's birding profile could help suggest relevant tours (for instance, if the user has never seen seabirds, promote pelagic tours; if they've already seen a species that is the star of a tour, maybe highlight a different aspect). However, for MVP, this can be minimal – perhaps manual insights by admin rather than automated. - **Light Gamification:** We may include badges like "500 Club" for seeing 500+ species, but these should be subtle. The aim is to **acknowledge achievements without turning the platform into a competitive scoreboard**. As the product owner notes, *"eBird already exists"* for hardcore tracking; our platform should *"display stats tastefully, not as leaderboards or XP bars."* We might, for example, show a small badge or icon next to the profile name if they're above a certain life list threshold, purely as social proof.

Technical Implementation: We will likely implement a **Supabase Edge Function** or Next.js API route to handle eBird API calls securely (so we don't expose our API key or hit CORS issues on the client). This function can be called when a user links their account or periodically. For instance, when the user enters their eBird username and clicks "Verify", our backend function could: 1. Call eBird API endpoint (with our server-side API key) for the user's species total. 2. Save the returned count in `profiles.ebird_life_list_count`. 3. Possibly also store a timestamp `ebird_last_sync` to track staleness.

Supabase Edge Functions are well-suited for this kind of third-party integration and can run globally with low latency [6]. They can be invoked on demand, and we could even schedule them (Supabase now supports scheduled invocations or we can use an external cron) to refresh eBird stats for all linked users maybe once a day or week.

Computed Fields and Derived Data

Our profile has several fields that are **derived or computed** from user activity. We need to decide which of these to store (denormalize for quick access) and which to compute on the fly or via background jobs. Key examples include: - **tours_completed:** number of tours the user has completed. - **referrals_count / referral_success_rate:** how many referrals the user made and how many converted (success rate = successes/total). - **tier_progress_percentage:** e.g. if tier is based on points, how far along the user is to the next tier. - **trust_score:** an aggregate reputation score.

Storing vs Computing: As a rule of thumb, if a value is **expensive to compute on each request** or frequently needed for sorting/filtering, it should be stored and updated asynchronously. If it's cheap to compute (like a simple count with an index) and not needed in large lists, on-the-fly is fine. Let's consider each: - **Tours completed:** This can be computed by counting rows in `reservations` table where `user_id = X` and `status = 'completed'`. With a proper index on `(user_id, status)`, this query is fast even on the fly. However, if we plan to display this often (e.g. on profiles and maybe also use it in trust_score), we might store it in the profile for convenience. In the schema above, we included `tours_completed` in the profiles table and would maintain it via triggers. - **Trigger approach:** Create a trigger on `reservations` after update/insert that if a reservation's status changes to "completed", increment that user's `tours_completed`. Likewise, if a reservation is canceled or a completed one is deleted, decrement it. This keeps the count in sync. This denormalization trades a bit of write complexity for very fast reads. - Alternatively, we could create a **view** or use a Supabase computed column for tours_count using a subquery. But since Supabase's real-time and RLS features work best with actual table columns, a trigger-maintained column is straightforward. - **Referrals and referral success:** We have a `referrals` table (referrer_id, referred_id, etc.). We might want to show how many people a user referred, and how many of those actually went on to complete a tour (successes). Similar to tours_completed: - We can maintain `referrals_count` (total referred) and `referrals_success` in the profile. Each time a referral is created (i.e., someone signs up with their code), increment `referrals_count`. If that referred user ends up completing a booking, increment the referrer's `referrals_success` (this could be triggered by the referred user's reservation completion). - **Referral success rate** can be computed on the fly as `referrals_success / referrals_count` (with proper handling if count is 0). No need to store the percentage itself, just compute it in the API response or client. - Because referrals might not be super high volume, this can also be computed with a join when needed, but having counts readily available could simplify things like awarding a "Referral Pro" badge when a threshold is hit. - **Tier progress percentage:** This is likely a purely virtual value. For example, if Tier 1 (Verified) requires 100 points and Tier 2 (Trusted) requires 500 points, and the user has 350, then their progress to Tier 2 is 50%. This can be calculated in real-time in the client or API by using the trust_score and tier thresholds. There's no need to store it in the DB, since it's just a presentation detail. We will, however, need a clear definition of what it takes to move between tiers (points, quizzes, admin approval, etc.) so we can compute this consistently. - **Trust Score calculation:** *What inputs should feed it?* The **trust_score** is a central derived metric representing user reliability/trustworthiness. We should define it clearly: - Start with a base of 0 for new users. - Increase trust_score for positive actions: completing tours (without issues), referring new users who complete tours, possibly leaving reviews or other community contributions, and passing the fieldcraft quiz (that could give a one-time boost or be required for a certain tier). - Decrease trust_score for negative signals: no-shows or last-minute cancellations of tours, receiving operator complaints, accumulating strikes (each strike could subtract a chunk of points). - Example scheme: +10 points for each tour completed, +5 for each successful referral, +20 for passing the quiz, +X for certain badges (maybe not, to avoid gaming), and -30 for a strike, -10 for a cancellation without good reason, etc. This exact scheme can be fine-tuned, but it provides a quantitative backing for the tier system. - The trust_score should probably be capped or normalized to some scale (e.g. 0 to 100 or similar) for simplicity. Tiers might correspond to ranges of trust_score (e.g. 0-49 = New, 50-79 = Verified, 80+ = Trusted) in addition to other criteria. - **Where to compute trust_score?** We could compute it on the fly by summing various factors via queries, but that would involve multiple joins (reservations, referrals, strikes, etc.) every time we need it. Better is to maintain it incrementally: - Use triggers or logic in the backend when key events occur. For example, when a reservation is marked completed, add 10 to user's trust_score. When `fieldcraft_quiz_passed` flips true, add the quiz points. When `strike_count` increases, subtract the penalty. - Alternatively, have a background job recalc trust_score for all users periodically (if the formula is complex). But real-time update is more gratifying (a user sees their score go up immediately after finishing a tour). - We must be careful to not double-count (triggers should ensure one-time addition per event) and consider reversals (if a tour completion is revoked or a strike is removed by admin, adjust accordingly).

Stored vs. Real-Time Summary: In general, **store what you need for fast filters and frequent display**, and **compute ad-hoc what is infrequent or dynamic**. For example, if we show a leaderboard of most tours completed (maybe for admins), having `tours_completed` stored lets us `ORDER BY` it easily. If we rarely need something (like referral success rate might not be shown publicly at all, could be only in admin view), computing that on the fly is fine.

We will utilize **Postgres triggers and functions** on Supabase to handle these updates. Supabase supports writing such triggers in SQL or PL/pgSQL that run on data changes. For instance:

```
CREATE OR REPLACE FUNCTION handle_reservation_update()
RETURNS trigger AS $$
BEGIN
  IF NEW.status = 'completed' AND OLD.status IS DISTINCT FROM 'completed' THEN
    UPDATE profiles SET
      tours_completed = tours_completed + 1,
      trust_score = trust_score + 10
    WHERE id = NEW.user_id;
  END IF;
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_reservation_complete
AFTER UPDATE ON reservations
FOR EACH ROW
EXECUTE PROCEDURE handle_reservation_update();
```

Similarly, we'd have triggers for insert on referrals (increment referrer's count) and on update of referred user's tour status to completed (increment referrer's success count and maybe trust score).

Some computations might be done in **Next.js API routes or Edge Functions** instead, especially if they involve external services or complex logic. For example, calculating tier_progress_percentage and deciding if a user is eligible to upgrade could be done on the server side when the user visits their profile or completes an action (since it may involve checking multiple conditions).

Background Jobs: If some data becomes expensive to update synchronously (for example, if we ever pulled a user's *entire* eBird life list species and integrated it), we'd push that to a background task. Supabase Edge Functions now support background tasks for long-running jobs [7]. For MVP, most of our derived fields are straightforward enough to handle in real-time triggers or immediate API calls.

Profile Privacy and Visibility Layers

Designing a privacy model is crucial so that users feel in control of their data and sensitive information is protected. We need to support multiple **visibility levels** for profile information: - **Public (anyone)** – e.g. a public profile page that non-logged-in users or any user can see. Likely this would include only very basic info: first name (or chosen display name), maybe location and bio if the user opts to share them, and perhaps high-level birding stats (if public). By default, we might keep most of this minimal unless the user explicitly makes their profile public. - **Authenticated Users (community)** – other logged-in birders who are not necessarily connected. They might see a bit more than the general public by default (for example, perhaps they can see the user's life list count or tours completed count to assess experience). But contact info (email/phone) would still be hidden. - **Tour Participants (contextual friends)** – users who have been on a tour together (or are about to) should be allowed to see more details about each other. For instance, if

I'm booked on a tour with you, I might get to see your full name, a way to contact you (if you consent, maybe via platform messaging or email proxy), and your past tour history or birding interests, since we're effectively teammates for that event. This can help group cohesion and trust (people know who they'll be touring with). - **Operators (for their customers)** – tour operators should have access to certain profile info for users who book their tours. This includes contact info (email, phone) to coordinate the trip, and possibly relevant medical/dietary notes if those were part of booking (not in profile per se). They might also benefit from seeing a user's tier or any flag status, so they know if someone is high-trust or potentially problematic (e.g., an operator might appreciate knowing if a user has multiple strikes for no-shows – although if a user is flagged, ideally the platform admins are already handling that). We'll expose only what is necessary for operations. - **Admins** – have full visibility into all profile data, including internal flags, strike counts, etc. They need this for moderation and support purposes. - **Self** – of course, each user can see all their own info (including things like their strike_count or trust_score if we decide to show those to them – perhaps a user can't see their strike_count to avoid conflict, but they should see trust_score and how they can improve it).

User-Configurable Settings: We will allow users to configure some privacy settings, within sensible bounds: - The user can decide if their profile is "public" or only visible to logged-in users. For example, a toggle **"Make my profile public to others"** – off by default (meaning only registered users can see it, not the whole internet). If off, we might still show a very limited public view (like just a display name and "This profile is private" message). - They can decide whether to display their eBird stats and perhaps tour history to others. For instance, **"Show my life list count on my profile"** (default off, since some consider that private). - They might have a setting **"Show my past tours to other participants"** – meaning if enabled, people who were on a tour with them can see which other tours they've done (which can be a conversation starter). If disabled, that info remains hidden. - Contact information like email/phone will **never** be public; it's only shared with operators for tours you book, and possibly with co-travelers only through an in-app messaging system (to protect emails). - Internal fields (strikes, flags) are **system-enforced private** – users have no ability to make those public (and likely they can't even see their own strike count, to avoid gaming the system).

Supabase RLS Implementation: Supabase's Row-Level Security will enforce much of this on the database side to prevent unauthorized reads/writes: - **Row-level filtering:** We can ensure that no one can select another user's profile row unless appropriate. For example, a base policy could be *"allow selecting from profiles if the profile is public OR the viewer is logged-in (for basic fields) OR the viewer shares a tour with the profile owner OR the viewer's role is operator for a tour of the owner OR the viewer is admin."* This can be a fairly complex policy using joins (e.g. checking the reservations table for a common tour). It's possible to include such logic in RLS (joining to external tables in policies is allowed, though one must watch performance [8]). For instance:

```
create policy "allow_profile_view" on profiles
for select using (
  -- admins can see all
  auth.jwt() ->> 'user_role' = 'admin'
  -- owners can see their own
  OR auth.uid() = id
  -- if profile is marked public
  OR profiles.is_public = TRUE
  -- if both users have a reservation for the same upcoming tour
  OR EXISTS (
       SELECT 1 FROM reservations r1
       JOIN reservations r2
         ON r1.tour_id = r2.tour_id
        AND r1.user_id = profiles.id
        AND r2.user_id = auth.uid()
       WHERE r1.status NOT IN ('cancelled') AND r2.status NOT IN ('cancelled')
     )

-- (Additional clause for operators: if auth.uid is an operator and has a tour
with this user)
  OR EXISTS (
       SELECT 1 FROM tours t
       JOIN reservations r
         ON t.id = r.tour_id
       WHERE r.user_id = profiles.id
         AND t.operator_id = auth.uid()
     )
);
```

The above is an illustrative policy logic (combining multiple OR conditions). In practice, we might split into multiple simpler policies for clarity (Supabase allows multiple policies per table). For example, one policy for "owners can view self", one for "public profiles visible to all authenticated", one for "tour participants can view each other", etc. Combining them is powerful but must be done carefully to not leave a hole. We also mark certain columns as **restricted** via separate tables or by not including them in a "public" view (discussed next).

- **Column-level restrictions:** As discussed, PostgreSQL doesn't have built-in per-column RLS, so we use other strategies:
- **Separate tables for sensitive fields:** We moved `strike_count`, `is_flagged`, etc., to `profile_private`. We will write an RLS policy on `profile_private` such that only the corresponding user (probably not even them for strike_count, maybe only admins) and admins can select it. For example, on `profile_private`: *"allow select if auth.uid() = user_id (perhaps excluding strike_count column via column privileges) or if auth.role = admin."* We might actually not even grant `authenticated` any select on `profile_private` at all, and only let a secure function or admin role access it. The main profile query doesn't include those fields, so they're effectively hidden.
- **Views for public data:** Another approach is to create a Postgres **VIEW** that selects only the non-sensitive columns from profiles (and maybe masks some based on privacy settings), and expose that view to the client instead of the base table. For instance, a view `public_profiles` that shows id, name, bio, etc., but not email or internal flags. We could then apply a simpler RLS on that view (though by default, SELECT on a view can bypass base table RLS unless configured carefully with `security_invoker` or by moving base table to a secure schema [9] [10]). A simpler method is what we've done: ensure the profile table itself only contains fields that are safe to potentially expose, and offload truly private fields elsewhere.
- **Column update restrictions:** We also enforce that certain fields can't be modified by users or only by admins. For example, `tier` or `trust_score` should likely be not directly settable by the user. We can use RLS `WITH CHECK` policies or triggers to prevent unauthorized updates. E.g., *"Users can*

update their own profile but cannot directly change tier, trust_score, strike_count, is_admin, etc." This can be done with a trigger that raises an exception if those fields are altered (as suggested in a Supabase discussion for column security) [11] [3] .
- We will leverage Postgres privileges as well: ensure the `authenticated` role (which all logged-in users use) does not have update rights on columns like `is_admin` or `strike_count` at all.

Default Visibility: We will adopt a **privacy-conservative default**. For any new user: - Their profile is not publicly visible to non-users by default. - Other authenticated users can see only a limited subset (perhaps just name and maybe a generic "Birder since 2024" or so). We might initially treat profiles as semi-private, only truly opened up when the user decides. - When a user books a tour, we can prompt them that their profile (name, bio, maybe contact preference) will be visible to their tour mates, encouraging them to fill in details. But if they haven't, that's fine – it might just show a placeholder. - Over time, as users become more comfortable, they might open up more. We will make it clear what info is shared and with whom in the UI (transparency is key to trust). - For example, **Strike Count and flags** are never visible to the user or others (only admins). **Trust score** might be partially visible to the user (e.g. "Your trust score is 50. Reach 60 to upgrade to Trusted tier!") but not directly shown to others except maybe an indirect indicator (like a badge or tier label). We likely won't show "User A has trust_score 72" to other users – we'll just show that User A is "Trusted tier" or not.

In summary, we implement privacy by combining **user settings, RLS rules, and data separation**. Supabase's flexibility allows us to enforce these rules at the database level for security. We should thoroughly test the policies to ensure, for instance, that a random authenticated user cannot select someone else's email or hidden fields (the query should simply return null or no row). By structuring queries through an API (Next.js or Supabase RPCs), we can also ensure the backend only queries what the user is allowed to get.

API Design for Profile Operations

We will expose a set of API endpoints (or use Next.js route handlers) to handle profile-related functionality. Using Next.js 14 (with the App Router) allows us to create **Route Handlers** under the `/app/api` directory for server-side API endpoints, which is ideal for integrating with Supabase. We'll outline key endpoints and their behavior:

- **GET `/api/profile/[id]` – Public Profile View:**
- **Purpose:** Retrieve the public-facing profile info for user with id or username `[id]`. This is what another user (or the public) sees.
- **Data:** This should return only fields allowed by the privacy settings. For example: name, bio, location (if not hidden), experience level, life_list_count (if user allowed it), number of tours completed, and maybe badges earned. It will not include email, strike_count, etc.
- **Auth:** Could be open to public (if we allow public profiles) or require auth if profiles are only visible to authenticated users by default. We'll likely allow public access if the user's profile is marked public.

- **Implementation:** We can use Supabase client on the server to select from `profiles` (and maybe join `user_badges`) with the RLS taking care of filtering. Or create a Supabase Cloud Function for it. But a Next route is straightforward here. E.g., `supabase.from('profiles').select('name,bio,location,experience_level,ebird_life_list_count,to` where the query will only return allowed fields due to RLS. Alternatively, call a Postgres function that applies all the privacy logic and returns a sanitized profile.

- **GET `/api/profile/me` – Own Profile Detail:**

- **Purpose:** Get the full profile of the logged-in user (for editing or viewing in their account section).
- **Data:** All profile fields that the user is allowed to see of themselves. This includes everything except perhaps internal flags. (We might decide whether to show them their strike_count or not; possibly not, to avoid antagonism – an admin can inform them if there's an issue. For trust_score, we *do* want them to see it or at least see their tier progress.)
- **Auth:** Must be authenticated. We get user ID from the JWT, and fetch their profile where `id = auth.uid()`.

- **Implementation:** This can be done with a Supabase client call from the frontend (`supabase.from('profiles').select(...).single()`) or via a Next API route that calls the same. Since the user can technically fetch their own profile directly via Supabase JS (with RLS allowing only their row), an extra API route isn't strictly necessary. But if we need to attach computed fields (like tier_progress_percentage or assemble data from multiple tables), doing it in a Next.js server function might be convenient. For example, Next route could fetch profile, then also compute `tier_progress = trust_score / tier_threshold[tier+1]` and add that to the response.

- **PUT `/api/profile/me` – Edit Profile:**

- **Purpose:** Allow the user to update profile fields like name, bio, location, social links, etc.
- **Data:** The request body would contain the fields to update. We will validate and strip out any disallowed fields (e.g., if someone tries to set `is_admin: true` – the RLS/DB will ignore or reject it anyway).
- **Auth:** Auth required. The RLS policy will ensure they can only update their own row. We also set up a policy: user can update all columns **except** restricted ones. We might implement that via a stored procedure or separate table as described. For instance, we could have to use an RPC (`supabase.rpc('update_profile', {...})`) that only updates allowed fields. Another simpler method is to have multiple update endpoints: one for general info, one for sensitive fields that only admins call. But likely a single endpoint is fine if RLS prevents the disallowed parts.

- **Implementation:** Use Supabase JS from the client or a Next API route to perform the update. After update, triggers might fire (e.g., if they changed `ebird_username`, we could trigger an eBird sync in background, or if they updated something that affects trust_score – though typically profile edits won't directly change trust_score except the quiz).

- **POST `/api/profile/verify-ebird` (or `/api/ebird-sync`) – Verify or Sync eBird:**

- **Purpose:** User initiates linking their eBird account. They provide their eBird username (if not already stored) and request to fetch data.
- **Process:** This endpoint (server-side) will call the eBird API as discussed. It might require the user to pass their own eBird API key if eBird's terms demand individual keys. (We learned that sharing one API key for multiple users might violate terms [4]. We could, as a stopgap, use one key just to fetch

public stats, which might be okay since it's not user-sensitive data; but to be safe, we can also register our app and get a key).

- The function will attempt to fetch something like `GET https://api.ebird.org/v2/product/lifelist/<username>?detail=full` (if such existed) or more realistically use a combination of region endpoints. If successful, update `ebird_life_list_count` in DB. It could also return the fetched stat directly.
- **Auth:** Must be the user themselves (so require JWT). Possibly require that they have already set their `ebird_username` in profile (or include it in this request).
- **Implementation:** Likely a Next.js API route or Supabase Edge Function that has our API key and does the HTTP request to eBird. The response would indicate success or failure (e.g. "eBird username not found or private").

- Optionally, if we implement a verification code method, this endpoint might also accept a code and then check eBird profile page for that code – but as decided, we will not do that for MVP.

- **POST** `/api/profile/fieldcraft-quiz` **– Quiz submission:**

- **Purpose:** Handle the user's fieldcraft quiz answers and update their profile if they pass.
- **Process:** The user takes a quiz (maybe a multiple-choice questionnaire about birding ethics or field safety). The front-end can evaluate the score, or send answers here for evaluation. To avoid exposing correct answers on client, it's better to evaluate server-side.
- If the user passed, we set `fieldcraft_quiz_passed = TRUE` and possibly increase their tier from 0 (New) to 1 (Verified), if other conditions are met (maybe passing the quiz is the primary requirement for Verified tier). We'd then also increment trust_score (say +20) as a reward.
- This endpoint may also trigger other actions: e.g., send them a "congrats on verification" email or issue a badge.
- **Auth:** Must be the user (or an admin manually could trigger for them, but usually user).

- **Implementation:** Could be a Next route that receives the answers, checks them against a correct key (could store correct answers in DB or code), computes score. If fail, responds with a message to retry; if pass, updates profile. This will also be subject to RLS (user can only update themselves). Alternatively, we store quiz answers in a table and have a function calculate, but likely simpler in code.

- **GET** `/api/profile/tier-status` **– Tier Eligibility:**

- (This could be merged with GET profile/me as additional info, but potentially a separate endpoint or an attribute in the profile response.)
- **Purpose:** Provide information on what the user needs to do for next tier. For example, "You are 10 points away from Trusted tier. Complete 1 more tour or get 1 referral to reach it."
- **Implementation:** This is mostly derived from trust_score and known thresholds, plus checking if quiz is passed (since maybe Trusted tier might also require the quiz, but Verified definitely requires it). This can be done on the fly in the client as well, but having the server summarize it might be nice.

- Possibly unnecessary to have a dedicated endpoint; the logic can live in the front-end or in the profile GET.

- **Admin/Operator Endpoints:** We should also consider endpoints for admin and operator actions:

- **GET** `/api/admin/profiles` – list of profiles with certain filters (like all flagged users or a search by email). Admin dashboard use.
- **POST** `/api/admin/flag-user` or **PUT** `/api/profile/[id]` – to allow admins to update someone's `is_flagged`, `strike_count`, or role. This would bypass normal user RLS; we'd authorize by checking the JWT role is admin (Supabase JWT custom claim or separate service role). Possibly easiest is to use Supabase's **Service Role** (admin secret key) for admin actions from a secure environment.
- **GET** `/api/operator/tour-participants/[tourId]` – gives an operator the profiles of users on their tour, with contact info. This ensures an operator only sees profiles for their tours. Alternatively, operators use the general profile endpoint but we supply a query parameter that includes tour context and RLS allows it via the reservations join policy.

Next.js vs Supabase Direct: With Supabase's client library, many of these operations (view profile, edit profile) can be done directly from the frontend by calling the database if RLS policies are in place. For example, editing profile: `supabase.from('profiles').update({bio: ..., location: ...}).eq('id', user.id)`. The RLS will ensure they can only update their row, and our policies/triggers ensure no forbidden fields are changed. This is a very "thin" API approach and leverages Supabase's auto-generated APIs. For more complex flows (quiz grading, eBird fetch), a Next.js API route or Supabase Function is needed.

We might adopt a hybrid: - Use direct Supabase calls on the client for simple CRU operations (Create profile done automatically at sign-up via trigger, Read profile, Update basic fields). - Use Next.js API or Edge Functions for anything requiring secret keys or heavy logic (e.g., eBird integration, quiz evaluation, tier upgrades, sending emails). - Ensure webhooks/triggers in the DB handle invariant updates (like counting tours, trust score changes) without needing a dedicated API call for those events.

Webhooks/Triggers on Profile Changes: We will have certain actions upon profile changes: - If a user upgrades tier (e.g., from 0 to 1, or 1 to 2), we can trigger a welcome email or notification. This could be done via a DB trigger that calls an HTTP webhook (Supabase doesn't natively call external URLs, but we could use an Edge Function listening to database changes or a `NOTIFY`/Realtime subscription). Simpler: our Next.js backend can periodically check who met criteria and send emails. - If `fieldcraft_quiz_passed` becomes true, perhaps trigger granting a "Verified Birder" badge, and sending data to analytics. - If a user is flagged or gets a strike, perhaps notify admins or log it somewhere. - When a user links eBird (ebird_username set), trigger a one-time fetch of their life list (this can be done by a function called via a trigger or by the backend after the insert).

Supabase **Triggers** in SQL can't directly send external requests (unless using extensions), so often the approach is: - Use Supabase Realtime (which can forward DB changes to clients) or a dedicated function. For instance, we could have an Edge Function subscribed to the `profiles` changes (via Supabase's realtime or just triggered by calling the function from the frontend after update) to perform side effects like external API calls or email. - Supabase is working on or has introduced `pg_net` extension which can do webhooks from Postgres, but that might be overkill at this stage.

In summary, we will define a clear set of API endpoints aligning with typical profile operations: - Viewing profiles (self vs others with privacy), - Editing profile, - Special actions (link eBird, take quiz, etc.), - Admin/operator specific reads/updates.

All of these should follow RESTful or RPC semantics, and we'll ensure to secure them by checking roles (via RLS or in handler code).

For example, a **Next.js route handler** for editing profile might look like:

```
// pages/api/profile/me.js (or in app/api/profile/me/route.ts for Next 13+)
import { supabaseServerClient } from '@/lib/supabase'; // an instance with
service role or user JWT
export async function PUT(request) {
  const { user } = await getUserFromRequest(request); // our helper to get JWT
user
  if (!user) return new Response('Unauthorized', { status: 401 });
  const updates = await request.json();
  // Strip out disallowed fields just in case
  delete updates.trust_score; delete updates.tier; delete updates.is_admin;
  // ... (more deletes)
  const { error } = await supabaseServerClient
    .from('profiles')
    .update(updates)
    .eq('id', user.id);
  if (error) {
    return new Response(error.message, { status: 400 });
  }
  return new Response('OK');
}
```

This is just illustrative; with RLS and proper DB constraints, many of these protections are enforced at the DB level too.

Performance Considerations

Performance must be kept in mind, especially for profile page load times and any queries that aggregate stats.

Database Indexing: We should add indexes to support the queries we'll run frequently: - On **reservations**: an index on `user_id` (to quickly find all tours a user has, e.g., to compute tours_completed or list their tour history) and an index on `tour_id` (for operators listing participants). Potentially a composite index on `(user_id, status)` if we often query completed tours for a user. - On **referrals**: index on `referrer_id` (to count how many a user referred and find those referrals). - On **profiles**: if we allow searching profiles by name or email (for admin console or operators finding a user), an index on `name` and `email` would help. Also index `role` if we query by role frequently (e.g., list all operators). - On **badges**: maybe not needed beyond PKs, unless we query "which users have badge X" often (then index on badge_id in user_badges). - If using `ebird_username` to fetch data or ensure uniqueness, consider a unique index on `ebird_username` (if we want to ensure no two users claim the same eBird account). - For any JSONB fields that we query inside (e.g., if we let filtering by social_links content, which is unlikely), we'd add GIN indexes. But most JSON (like social_links) we won't filter by, it's just stored data.

Query Patterns: The profile page will likely need to display: - Basic profile info (one row from `profiles`), - Possibly the user's badges/achievements (join on user_badges->badges), - Perhaps a summary of upcoming or past tours (which could come from reservations join tours table), - Birding stats (life list etc).

We should avoid N+1 queries. Using a single SQL query with joins or a single Supabase `.select()` with foreign tables can fetch a lot in one go. Supabase allows queries like:

```
supabase.from('profiles').select(`
   id, name, bio, location, experience_level, ebird_life_list_count,
tours_completed,
   user_badges ( badge: badges(name, description) )
`).eq('id', someId);
```

This would retrieve the profile and their badges in one call. Under the hood it does multiple queries but the Supabase API optimizes it.

For listing **tour history** on profile: Instead of storing a big JSON list, we can query `reservations` (with join on tours to get tour info) on demand. If a user has, say, 50 completed tours, fetching them isn't too heavy. We can paginate if needed. We should ensure `reservations` table is indexed and perhaps has a composite primary key (likely (tour_id, user_id) or an internal id) for performance.

Caching Strategies: - We might use Next.js Incremental Static Regeneration or SSR caching for public profile pages, but since content can change with new tours or updated stats, SSR (Server-Side Rendering) on each request for a public profile might be better to always show fresh info (with data fetched from Supabase with appropriate caching headers). For our internal logged-in views, Next.js can use **React Server Components** to fetch data on the server side, reducing client load. - If some computations are expensive (e.g., if we ever show a "global leaderboard of top birders"), we might pre-compute those in a materialized view or cache in Redis. But at MVP scale (one region, modest users), simple queries should suffice. - Supabase's network is quite fast (managed Postgres with caching). But if we notice any slow queries, we'll add the necessary indexes or denormalize further.

Supabase Edge Functions & Edge Caching: - For third-party API calls (eBird), we minimize impact by doing them asynchronously. For example, when a user loads their own profile page, if we need fresh eBird data, we might show the last cached number and concurrently call a refresh in the background (updating later). This way, the UI is snappy. Edge Functions could also be scheduled to update eBird stats at off-peak times. - We could also use an Edge Function to compute something heavy and cache it. However, at this stage, nothing seems too heavy. Trust score and tour counts are low-cost calculations given the volume.

Concurrency and Consistency: - Using triggers to update counts (tours_completed, etc.) means those numbers are always up-to-date by the time we query the profile. We should be mindful of race conditions (e.g., two reservations completing at the exact same time). PostgreSQL will serialize the updates on the same row, so it should be fine, but we must ensure our trigger functions use atomic operations (they do a single `SET tours_completed = tours_completed + 1` which is atomic). - If a user's trust_score is derived from many events, updating it in real-time could cause a slight performance hit on those event transactions (each completion triggers an update on profiles). This should be okay unless the events are extremely frequent. If it became an issue, we could offload trust_score recalculation to a background job that runs periodically. But real-time feedback is nice for user engagement.

Edge Functions vs Next.js Server: - We have two places to run server code: Next.js API routes (running on our Vercel/Node server environment) and Supabase Edge Functions (running on Deno at Supabase's edge). Both can achieve similar tasks. A best practice is to use Edge Functions for things tightly integrated with the database or needing to run close to the data (to reduce latency or use Supabase's secrets), and Next.js API for things more related to our frontend or that benefit from being in the same codebase/runtime as the site. - For example, verifying eBird could be a Supabase Edge Function if we wanted to call it from the database (imagine a stored procedure `verify_ebird(username)` that calls the function). But since it's user-initiated, a Next.js API route is equally fine and keeps all logic in one project. - Using Supabase Edge Functions might shine if we later implement things like a scheduled daily summary email or a webhook receiver (e.g., if we had webhooks from another service to update profile, we could catch them there). As noted in Supabase docs, Edge Functions are great for "integrating your Supabase project with third-parties" [6] , so we will use them where appropriate.

Load Testing and Scaling: - For MVP in one region, the user count is small. But thinking ahead, the design should scale. Splitting data into separate tables (profiles, profile_private, user_badges, etc.) helps with scaling because queries can target just what's needed. - If we expect growth, we might consider caching the profile page response. Next.js could statically generate public profile pages and revalidate them every X minutes. Since public info doesn't change second-by-second, that's an option to reduce DB load if we had thousands of random public profile hits. But for now, likely not needed. - We will ensure to use **pagination** or sensible limits on any lists (e.g., if showing user's tour history, don't pull 1000 records at once unnecessarily). - Also, by designing the schema not to be tied to a single region (e.g., not assuming all tours are in one place), we can later shard by region if needed or at least filter by region easily.

Multi-Region & Internationalization (Future): While not needed for MVP (which is focused on East Gippsland, Australia), it's wise to not paint ourselves into a corner: - All monetary values (if any in profiles or tours) should have an associated currency. For example, if there is a price or currency preference on profile, store it, or default currency on tours. Right now, everything is AUD, but we can add a `currency` column on tours or payments if those exist. - The `location` field on profile could later be used to filter content (e.g., show nearby tours). Perhaps instead of free-text, we might later want structured location (country, state). For now, free text is fine; just note that if expanding globally, we might add normalized location fields. - Role-wise, if we onboard operators from different regions, we may need an operator<->region mapping or even multi-currency pricing. Our schema can handle this by adding fields; nothing fundamental needs change with profiles. - We might also consider a future `organizations` table if we have multiple tour companies operating. Each operator profile could be linked to an organization. But at MVP, one operator might just be one organization. - Internationalization of text (like bio in different languages) is not in scope, but nothing in our design prevents extending that later.

In conclusion, this profile system architecture emphasizes **extensibility and security**: we normalized data into appropriate tables for badges and stats (avoiding performance pitfalls of overusing JSONB [2]), we integrated eBird at a sensible level for credibility (while noting the limitations of their API and respecting user privacy [5]), and we implemented a robust privacy model leveraging Supabase's RLS and design patterns (like table splitting for sensitive data [3]). The API layer in Next.js/Supabase is designed to cover all user and admin actions with best practices such as server-side verification for external integrations and incremental updates for derived fields. With proper indexing and judicious use of serverless functions, the system should scale well and provide a fast, secure experience for users managing and viewing profiles on the birdwatching tour platform.

---

[1] Custom Claims & Role-based Access Control (RBAC) | Supabase Docs
https://supabase.com/docs/guides/database/postgres/custom-claims-and-role-based-access-control-rbac

[2] database - PostgreSQL JSONB versus keeping a separate table - Stack Overflow
https://stackoverflow.com/questions/45772130/postgresql-jsonb-versus-keeping-a-separate-table

[3] [11] Policy that limits colums that can be updated · supabase · Discussion #656 · GitHub
https://github.com/orgs/supabase/discussions/656

[4] I made a Website that Allows for Easy Use of the eBird API! Find Nearby Hotspots, Recent Notable Observations in Your Region, and a lot more! : r/birding
https://www.reddit.com/r/birding/comments/152e20f/i_made_a_website_that_allows_for_easy_use_of_the/

[5] My eBird : Help Center
https://support.ebird.org/en/support/solutions/articles/48000794682-my-ebird

[6] [7] Edge Functions | Supabase Docs
https://supabase.com/docs/guides/functions

[8] Can you use a join in a row level security policy? #811 - GitHub
https://github.com/orgs/supabase/discussions/811

[9] [10] database - Supabase: Solutions for column-level security - Stack Overflow
https://stackoverflow.com/questions/72756376/supabase-solutions-for-column-level-security