ChatGPT

Operator Public Profile Data Architecture for webird.ai

Operator Entity Design

Separate Operator Table vs Extending Profiles: Given that operator accounts have many fields and behaviors distinct from regular users, it's wise to use a separate `operators` table rather than bloating the `profiles` table. This aligns with a **hybrid design**: keep common user info in `profiles`, and put operator-specific data in its own table [1] [2]. A separate table makes it easier to manage role-specific fields (business name, credentials, etc.) and apply different security rules, without cluttering all user records [3]. It also improves clarity and security by segregating operator data (e.g. tour info, business details) from normal user data [3]. We'll remove the current `is_admin` flag approach and instead implement a proper role system.

Relationship Between Person and Business: Each operator is essentially a business entity. In the MVP, we expect a 1:1 relationship (each operator owned by one profile), but we will design for future flexibility where a company could have multiple team members. The recommended approach is a **many-to-many relationship** between users and operators using a junction table (often called a membership or group-user table) [4]. This way, one user can own or belong to multiple operator accounts and an operator can have multiple associated users (guides, co-owners, staff) [4]. For example, we might have:

- `operators` **table:** Each record represents a tour operator business (whether a solo guide or a company). Key fields might include an `id` (PK), `owner_profile_id` (initial owner's user ID), `name` (business name), and various profile fields (see next section). This table is linked to `profiles` via the owner or via memberships.
- `operator_members` **(or** `operator_users` **) table:** junction of `profile_id` and `operator_id` (both PKs), plus a `role` column. The role could be `'owner'`, `'admin'`, `'guide'`, etc., to distinguish permissions. This design allows multiple users per operator and multiple operators per user [4] [5]. In MVP, we'll likely have one entry per operator (the owner), but the schema will accommodate more. The membership table can also store role-specific info if needed (e.g. an operator-specific display name or title for the user) [5].

All tours will reference the `operators` table. The `tours.operator_id` FK should point to the operator entity (not directly to a user profile as currently). This cleanly separates tour ownership from individual user accounts.

Solo vs Company Operators: For a solo operator, the person's profile will be linked as the owner in `operator_members`. In effect, the individual and business are the same, but we still create an operator entry so that the data model is consistent. They might choose a business name or just use their own name for the operator profile. For a company with multiple guides, the operator entry represents the company, and multiple `operator_members` (owner, guides) link the team's profiles. This approach covers both scenarios seamlessly. It's generally advisable to plan for multi-user accounts even if initially not used, to avoid costly refactors later [6]. (DoneDone's team found that even if you think users won't need multiple accounts, it's safer to include a linking table from the start [4] [7].)

Role-Based Access: With this structure, we can enforce permissions via Supabase Row-Level Security (RLS). For example, an operator's members (by checking `operator_members.profile_id = auth.uid`) can be allowed to update that operator's data and its tours. Admins can have a separate role that bypasses these checks. Using a separate table for operators makes it straightforward to restrict normal users from accessing operator-only data unless they are members. We'll implement RLS policies so that: - Only operator members or admins can modify an `operators` record or its related content (tours, documents, etc.). - Public can read certain operator fields (the public profile info), but not sensitive fields (covered in Privacy & Visibility).

Proper indexing: we'll index foreign keys like `operator_members.profile_id` and `operator_members.operator_id`, and `tours.operator_id` for efficient joins. Each table will have PKs and FKs with cascades as appropriate. This ensures even with more complex joins, the queries remain fast.

Operator Profile Fields

To create rich public profiles, we need to store a variety of information about each operator. Below are the key fields and their proposed types, grouped by category:

- **Business Identity:**
- **Operator Name** (string/text) – The public name of the business or guide. This might be the individual's name for a solo guide or a company name for larger operators.
- **Logo or Profile Image** (URL or storage reference) – An image representing the operator (e.g. company logo or guide photo). This will be stored in Supabase Storage and referenced by URL/path.
- **Tagline** (string) – A short slogan or one-liner that captures the operator's brand or ethos (e.g. "Expert birding tours in the Outback").

- **Description** (text) – A detailed overview of the operator: background, story, what makes their tours special, etc. This can be a longer Markdown or text field for rich content.

- **Experience & Expertise:**

- **Years of Operation** (integer) – How many years the operator has been in business (could also be derived from a start year field). This helps convey experience at a glance.
- **Regions Covered** (relation or array) – The geographic areas where the operator runs tours. This could link to a `regions` table (many-to-many via an `operator_regions` table) or be stored as an array of region identifiers. Listing service areas is useful if operators span multiple locations.
- **Specialties (Bird Species or Habitats)** (text or related table) – A brief summary of what the operator specializes in. For example, certain bird families (raptors, seabirds) or habitats (rainforest, wetlands). SafariBookings, for instance, uses a "Tour Types" field in profiles to show what an operator specializes in [8]. Similarly, we should allow operators to indicate their niche (this could be free-text tags or a predefined list of categories).

- **Languages Spoken** (text or small table) – Languages that the operator or guides can speak (important for international customers). This could be a text array (e.g. ["English","Spanish"]) or a separate table if we want to normalize and filter by language.

- **Credentials & Trust Indicators:**

- Rather than a single field, credentials will be handled via a related table (see **Verification & Trust Flags** below). However, on the profile we may display **verified badges** or summary info (e.g. “Licensed Guide – Verified”, “Insured – Yes”). We’ll have a mechanism to show which credentials the operator has provided and which are verified (e.g. an icon or label for each). The details (like license numbers or document scans) will not be directly shown publicly, just the fact of verification.

- **Equipment & Facilities:**

- For operators offering specialized equipment (e.g. birding by boat), we’ll include fields to highlight these:

    - **Equipment/Facilities Description** (text) – A general description of equipment provided (e.g. “All tours include high-quality binoculars and a spotting scope”).
    - **Vessel Details** (text or sub-table) – For boat-based operators, details like boat name, type (boat, 4x4, etc.), capacity, safety gear. If an operator has multiple vessels or vehicles, we might use a separate table (`operator_assets`) to list each with its specs. For MVP, a simple text field or a JSON object could suffice to list key equipment.
    - These fields help set expectations (capacity, comfort, safety) for customers, especially for niche tours. We can expand this as needed (e.g. a structured table if many operators need to list multiple assets).

- **Media (Photos & Videos):**

- **Photo Gallery** – Multiple images showcasing the tours, wildlife, or past tour groups. We will allow operators to upload a gallery of photos. Likely this is handled by a separate table (e.g. `operator_media`) with an `operator_id` and a storage URL for each image, plus maybe a caption or order. Having numerous high-quality photos is crucial for engagement – TripAdvisor notes that pages with 20+ photos get 150% more engagement from travelers [9]. We should encourage operators to add plenty of photos (and display them prominently).
- **Video Links** (optional) – If operators have promotional videos (hosted on YouTube/Vimeo or uploaded), we can allow a few video links. Video can further boost engagement (profiles with videos see ~34% more engagement) [10]. In the database, this could be another media table or simply a field for a YouTube embed link.

- We will store at least 3-6 images minimum (we could even enforce a minimum number of images to ensure profile richness, similar to some platforms requiring a certain number of photos [11]). The images and videos are public-facing and give visual proof of the operator’s experience.

- **Contact & Social:**

- **Location/Base** (string) – The operator's base location (city/region). This gives context on where they operate from (e.g. "Based in Darwin, NT"). We will not list a precise address publicly, just a general location for context.
- **Contact Info** – We will collect email, phone, or other contact details, but **these will not be shown publicly** (see Privacy section). Instead, the profile page will have a "Contact Operator" button that uses our platform's messaging system. Internally, in the database we might store:
    - Email, phone number, or other preferred contact (for admin use or post-booking).
    - Perhaps an emergency contact or business address for admin records.
- **Website & Social Links** (URLs) – If the operator has a website, Facebook, Instagram, etc., we *could* allow them to list these. However, linking out might encourage off-platform communication. Many marketplaces choose not to show external contact links to avoid leakage. If we do allow it (for credibility), we should weigh the risk. At minimum, this can be stored (in case we let premium operators list their site). If shown, it might only be visible to logged-in users or on request.
- **Messaging** – We won't store messaging here (that will be in a separate messages table), but from a data perspective, the public profile will integrate with a messaging feature that uses the operator's user ID(s). The key is that **the platform mediates initial contact**. We explicitly will **not show** personal emails or phone numbers on the profile to protect privacy and keep the booking on-platform [12].

Schema Example: In the `operators` table, we could have columns like: `id (PK)`, `name`, `tagline`, `description`, `logo_url`, `base_location`, `years_experience` (or `established_year` to compute years), etc. Fields like specialties or languages could be a text array (e.g. `specialties text[]`, `languages text[]`) or handled via join tables if we prefer normalization (e.g. a separate `operator_specialties` table linking to a `species` reference table). For simplicity, MVP can use arrays or JSON for things like species or equipment lists, and we can normalize later if needed for filtering.

All these fields (except contact info) would be **publicly visible** to showcase the operator. We will ensure the profile page is comprehensive – studies show operators with fully filled-out profiles (all fields, lots of photos, etc.) appear more active and trustworthy [13]. In our UI, we'll encourage operators to fill every section ("fill all the boxes — email, website, phone number, address. Add as many photos as you can" [14]) to avoid sparse profiles.

Guides vs Operators (Modeling Multi-Guide Teams)

Multiple Guides per Operator: Our design supports an operator having multiple guides or staff members. Using the `operator_members` junction, we can associate many user profiles to one operator. Each guide will have their own entry in the `profiles` table (with personal info like name, avatar, etc.), and a membership linking them to the operator with a role (e.g. guide, manager). This approach treats guides as full platform users, which is beneficial if later we allow guides to log in, manage tours, or communicate with guests. Even if in MVP only the owner actively manages the profile, having guides as user profiles keeps data consistent and allows expansion.

Guide Mini-Profiles on Operator Page: Customers are often interested in *who* will be leading the tour. Thus, on the operator's public profile, we can display a "Meet the Guides" section listing each guide's name, photo, and a short bio or credentials. To support this: - We can add a **bio field** to the `profiles` table (for any user, which guides can use to describe themselves professionally). If the profile already has some "about me" (birding experience) fields from the user profile research, those can be leveraged. - The `operator_members` table could also hold a **title/position** (e.g. “Lead Guide” or “Boat Captain”) for each member, which the owner can set. This way, on the profile we can show *“John Doe – Lead Guide, 10 years experience”*. - The guide’s photo comes from their user profile avatar. We can encourage guides to upload a professional photo.

What Guide Info Matters: Key details for each guide include their **experience (years birding or guiding)**, **areas of expertise** (e.g. specific birding skills or local knowledge), **certifications** (like if they are a certified ornithologist or first-aid trained), and **languages** they speak. Essentially, anything that might build trust with customers should be shown. We might list a couple of highlight bullets per guide (e.g. “Expert in raptor identification; Fluent in Spanish”). All this information can be stored either in the guide’s profile (as structured fields or a bio) or in an extended guide profile table. Since guides are users, it’s simplest to reuse the `profiles` fields (e.g. a guide can have a “birding level” or certification field in their user profile from our previous research).

For instance, if our `profiles` table (from user research) already includes things like *years birding*, *favorite bird*, etc., those could be repurposed or extended for guides. If not, we could create a `guide_profiles` table linking to profile ID with specific guide-only info. MVP-wise, using the existing profile/bio fields is fine.

Data Model for Guides: To illustrate, suppose **Operator X** has two guides: Alice and Bob. In `operator_members`, we will have: - (operator_id = X, profile_id = Alice, role = 'guide') - (operator_id = X, profile_id = Bob, role = 'guide')

Alice and Bob will each have a row in `profiles` with their personal details and a pointer to their auth user (or we can create them as users without auth if needed just for display, though ideally they sign up). The membership table can optionally hold **status** (active/inactive) in case a guide leaves the company, we can deactivate rather than delete for records.

Permissions: Only the operator’s owner/admin should be able to add or remove guides (initially likely done by an admin or via a request, since MVP doesn’t have team management UI). With RLS, we’d ensure only an operator’s owner (or platform admin) can insert/delete their `operator_members` records.

Showing Guides on Profile: The public profile page will query `operator_members` for that operator (joined with `profiles`) to get the list of guides. We might show each guide’s name, photo, and a short description. This helps customers feel comfortable (they see the faces and qualifications of who might be leading them). It’s a selling point, especially in a premium segment.

If Guides are not Users: In rare cases, an operator might want to list a guide who doesn’t have a separate login (maybe a subcontractor). We could allow a pseudo-profile creation, but it’s simpler to require each guide to have a profile (the owner can invite them to join, or create a profile for them). DoneDone’s membership design indicates supporting multiple users per account is best done with a join table and distinct user records [4]. We’ll follow that approach, which inherently covers guide listings.

Reviews & Ratings Architecture

A robust review system is critical for trust. We need a schema to store customer reviews of operators (and their tours) and to display aggregate ratings.

Review Entity: We'll create a new `reviews` table. Each review should capture: - **id** (PK) - **operator_id** (FK to operators) - **tour_id** (FK to tours, nullable if we ever allow general operator review; but ideally tie to a specific tour experience) - **user_id** (FK to profiles of the reviewer) - **rating** (integer, e.g. 1–5 stars) - **comment** (text for the review content) - **created_at** timestamp - **…** possibly fields like `booking_id` (FK to a booking/reservation if we want to double-ensure veracity), or a boolean `verified` (though we can infer verified if linked to booking).

Tour-Specific vs Operator-Level Reviews: We will likely tie reviews to **tours (bookings)**, meaning when a user leaves a review, they are reviewing the particular tour they went on. This is similar to TripAdvisor/Viator where each tour listing has its own reviews. However, since one operator may run many tours, the operator's profile should show an **aggregate rating** and potentially all reviews across their tours. By storing `operator_id` alongside `tour_id`, we have flexibility: - We can query reviews by `operator_id` to get all feedback for that operator. - We can query by `tour_id` to show reviews on a specific tour page. Alternatively, we might *not* store operator_id in the review if we can join through tour, but storing it denormalizes for convenience (one less join to get operator's reviews, at the cost of redundancy). Given an operator deletion would cascade to tours anyway, either approach works. For simplicity and performance, we'll store `operator_id` on the reviews (and enforce via trigger or check that it matches the tour's operator).

Verified Purchase Only: We will enforce that only users who actually attended (booked) a tour can leave a review. This can be implemented by linking the review to a `booking_id`. For example, if we have a `bookings` table (with fields `tour_id`, `user_id`, `status`, etc.), we add `booking_id` to reviews. We ensure via RLS or a database constraint that: - `reviews.user_id = bookings.user_id` and `reviews.tour_id = bookings.tour_id` for the given booking, and that booking has status "completed". Essentially, the review must correspond to a completed booking by that user. - Each booking can only have one review (we can enforce a unique constraint on booking_id in reviews to prevent duplicates).

If a user somehow tries to insert a review for an operator without a completed tour, the policy will reject it. This is similar to "Verified Purchase" badges on Amazon – only legitimate participants can review [12]. (We might mark reviews on the UI as "Verified" implicitly because all reviews will be from real bookings in our system.)

Review Content and Responses: The `reviews` table will hold the customer's rating and comment. We should allow operators to **respond to reviews** (publicly visible responses). For that, we can either: - Add a column `response_text` (text, nullable) and `response_date` to the reviews table. When an operator writes a reply, we fill these in. - Or have a separate table `review_responses` with `review_id`, `operator_id`, `response_text`, `response_date`. (One-to-one relationship since typically one official response per review.)

The separate table might be cleaner (and could allow multi-step back-and-forth, but generally one response is enough). MVP: a single response stored in the reviews row is fine.

Only users who are members of the operator (or admin) should be able to create or edit a response. RLS can enforce that on the `review_responses` or the `reviews.response_text` field (through a policy that allows update of that field only to operator members).

Ratings Aggregation: We need to compute the operator's overall star rating and review count to display on their profile (and possibly in listings). We have two main options: 1. **Compute on the fly:** Query the `reviews` table for each operator page load (e.g. `SELECT AVG(rating), COUNT(*) FROM reviews WHERE operator_id=X`). This ensures real-time accuracy but could be slow if there are many reviews or if loading many operator cards simultaneously. 2. **Store precomputed values:** Maintain in the `operators` table fields like `avg_rating` and `review_count`, updated whenever new reviews are added or changed. This is denormalization but vastly speeds up reads [15] [16].

Given we anticipate potentially many reviews and want to sort or filter by rating, storing the aggregate is beneficial. We will likely add columns `rating_avg` (float) and `rating_count` (int) in `operators`. Initially they can be NULL or default 0.

Updating Aggregates: We will use either triggers or Supabase edge functions to update these aggregates when a review is inserted/updated. For example, using a Postgres trigger on `reviews` AFTER INSERT: - Increment the operator's review count and adjust the average using the running average formula [17]. We can do:

```
new_avg = (old_avg * old_count + NEW.rating) / (old_count + 1)
new_count = old_count + 1
```

Then update `operators` set `rating_avg = new_avg, rating_count = new_count` where `id = NEW.operator_id` [18]. This constant-time update avoids a full aggregate scan on each insert [17].

Similarly, for a review update (e.g. user edited from 4 stars to 5 stars), or delete, we'd adjust accordingly (or simpler: recalc from scratch in those less frequent cases). Alternatively, we could recompute average by subquery in trigger (SELECT AVG from reviews where operator_id = X) [19], but the incremental method is more efficient.

This trigger-based approach ensures the operator's `rating_avg` is always up-to-date after each review [20]. In a low-volume scenario, we could recalc on the fly, but as reviews grow, it's more efficient to maintain the computed values [21].

Schema summary for reviews: - Table `reviews`: PK id, FKs `operator_id`, `tour_id`, `user_id`, maybe `booking_id`. Fields: rating (smallint), comment (text), created_at, etc. - Table `review_responses`: PK id or just reuse review_id as PK, FK to `review_id`, `operator_id`, fields: response_text, responded_at.

Indexes: We will index `reviews.operator_id` (for quickly fetching all reviews of an operator and computing aggregates if needed) and `reviews.tour_id` (to show tour-specific feedback). Possibly an index on `(operator_id, user_id)` to ensure a user doesn't review the same operator multiple times outside of separate tours (but since linked to bookings, that's naturally limited).

Verified tags: In the UI, all reviews coming from our system are inherently verified, but if we import any external reviews or allow manual addition, we might mark those accordingly. For now, assume all are verified since they come via completed bookings.

Review RLS: - Inserting a review: require `auth.uid = booking.user_id` for a completed booking matching the review's tour/operator (enforce via SELECT EXISTS subquery in policy). - Reading reviews: generally public (anyone can read reviews on an operator's page). - Editing/deleting: only the author can edit their review (and only within certain time?), and perhaps we allow admin to remove inappropriate ones. We'll add policies accordingly.

Aggregate Display: On the operator profile, we'll show something like "★ 4.8 (32 reviews)" which comes directly from the `operators.rating_avg` and `rating_count` fields (no slow query needed at runtime). We might also show a breakdown (how many 5-star vs 4-star, etc.), which if needed can be computed on the fly or precomputed if performance demands.

Verification & Trust Flags

To build trust, we need to track various credentials and verifications for each operator. This includes licenses, permits, insurance, certifications, etc. We'll design a flexible structure for these:

Credentials Table: We introduce an `operator_credentials` table to store documents or certifications. Key fields: - **id** (PK) - **operator_id** (FK to operators) - **type** (text or enum) – The kind of credential. For example: `"guide_license"`, `"national_park_permit"`, `"insurance"`, `"first_aid_cert"`, etc. We can maintain a predefined list or enum of allowed types for consistency. - **document_url** (text) – URL or path to the uploaded document (in Supabase Storage) as proof. E.g., a PDF of their insurance, a scan of their permit. - **identifier** (text, optional) – Any identifying info like a license number or certificate ID. - **expiration_date** (date, nullable) – When this credential expires, if it is time-limited. - **verified** (boolean or status enum) – Whether an admin has verified this credential. - **verified_at** (timestamp) and **verified_by** (FK to admin profile) – record who and when verified. - **notes** (text, optional) – Admin notes or comments on verification (e.g. "Verified via document #1234, expires in June").

This structure allows an operator to have multiple credentials (rows). For example, one operator might have a guide license expiring 2025, a boat permit expiring 2024, and an insurance policy expiring 2023 – each would be a row.

By using a separate table, we handle one-to-many elegantly and can easily query or update specific credentials. It's better than adding many columns to `operators` like `has_license`, `license_expiry`, `insurance_expiry` etc., which would be sparse or insufficient as needs grow.

Linking Regions or Tours: If a credential is region-specific (e.g., a permit for a particular national park), we can include a `region_id` reference in `operator_credentials`. This way, if an operator works in multiple regions that require distinct permits, each permit can be tied to the correct region. (Alternatively, encode the region in the `type` or have a subtype, but an explicit column is clearer.)

Verification Workflow: - **Submission:** The operator (or an admin on their behalf) uploads a document or enters credential info. We create an `operator_credentials` record marked `verified = false` initially. - **Admin Review:** An admin user will check the document. If it's valid, they set `verified = true, verified_at = NOW(), verified_by = admin_id`. If not, they might reject or leave it unverified (we might add a status or reason if needed, e.g., `status: pending/approved/`

`rejected` instead of simple boolean). - Possibly, we notify the operator of verification status (out of scope for DB design but just noting).

Enforcing Authenticity: We may want operators to periodically update documents. The `expiration_date` helps with that. We won't store a separate "expired" flag because we can derive it by comparing the date to TODAY [22]. This avoids redundant data and auto-expires credentials without manual toggling. A simple view or query can find credentials where `expiration_date < now()` to flag them. We could schedule a daily function to mark credentials as expired or send notifications. But for display, we'll simply not show an expired credential as "verified" (or show it with an "expired" label).

For example, if insurance expired yesterday, `verified` might still be true (since it was valid at verification time), but our UI can show "Insurance: **Expired** (as of Date)" by checking the date. We might also proactively require re-verification by admin for such cases.

Public Display of Verifications: On the operator's public profile, we **do not expose the actual documents** or detailed info (for privacy and practicality). Instead, we surface **trust badges or lines** such as: - "Licensed Tour Operator (Verified by webird)", - "Liability Insurance on file (Expires Dec 2025, Verified)".

These would be derived from the `operator_credentials` data. For instance, any credential with `verified=true` of a certain type would enable a badge. We might have a mapping of credential types to user-facing labels and icons.

We should be careful to only display if verified and not expired. Unverified submissions might be visible to the operator themselves in their dashboard ("Status: pending verification") but not to customers.

Security & Privacy: The `operator_credentials` table contains sensitive documents. We will lock it down via RLS: - Only the operator's members (owner) and admins can select or download their documents. - Regular users or the public get **no access** to this table. They only see the outcome (badges). - We might even separate the document storage permissions: Supabase Storage can have a bucket where only the uploader and admins can read. That prevents someone from guessing a URL.

Data Lifecycle: If an operator's credential is renewed, we have two approaches: - Update the existing row's `expiration_date` and maybe attach new file (overwriting or new URL). Keep `verified` false until re-verified. - Insert a new row for the new validity period, perhaps marking the old one as superseded or leaving it as historical. If we want a history, we might keep old records. But likely we only care about current validity, so updating in place is fine for simplicity.

Example Records: - Operator 5 uploads a guide license PDF, expiring 2024. We add: `{id: 101, operator_id: 5, type: "guide_license", expiration_date: 2024-12-31, verified: false}` initially. Admin verifies -> set `verified=true, verified_at=..., verified_by=...`. - Operator 5 also uploads an insurance document (no explicit expiration given? Could be yearly). We might require an expiry date input. Suppose it's valid through 2023-06-30. We store that. Our system can later flag that after June 30, 2023 it's expired and maybe send them a reminder to upload a new one.

Indexes: We will index `operator_credentials.operator_id` for quick lookup of all credentials for an operator (when an operator or admin views them). Possibly index `type` if we query by type often (like find all with type "insurance" about to expire). Also index `expiration_date` if we run scheduled jobs to find expiring soon.

Admin Interface: Admins will have a dashboard to view and verify documents. They might filter by type or pending status. The structured table makes this manageable.

In summary, this design centralizes all trust documents in one related table, with robust metadata. It's flexible to add new credential types without schema changes (just use a new type string). It also supports fine-grained verification (each item verified separately). This is far preferable to scattering verification fields across the operator profile.

By modeling expiration, we also ensure we **don't treat outdated credentials as valid** – we use the data itself to check validity, rather than a manual flag [22].

Statistics & Computed Fields

We want to display various performance stats on the operator profile (both for transparency to customers and as gamification for operators). The suggested stats include: **tours completed, total guests served, years on platform, response time, booking confirmation rate,** and **repeat customer rate**. We need to decide how to compute and store each.

1. Tours Completed: This likely means the number of individual tour instances the operator has successfully run. If each booking or tour date is an instance, we can count those. Assuming we have a `bookings` or `tour_occurrences` table that tracks each scheduled tour date and its completion: - We can increment a counter whenever a tour is marked "completed". For example, when a guide finishes a tour and marks it done (or when the date passes with bookings, etc.), we update an `operators.tours_completed` field. - Alternatively, compute on the fly: count all bookings with status = completed for that operator. However, doing that for every profile view could be expensive over time. Better to maintain a counter. - Implementation: A trigger on a booking status change (from pending/ongoing to completed) could do `UPDATE operators SET tours_completed = tours_completed + 1 WHERE id = NEW.operator_id` if NEW.status = 'completed'. Similarly, if a booking is canceled or the status is rolled back, adjust accordingly (or ignore cancellations if we only increment on completion). - If tours are separate from bookings (like a tour that has many bookings on the same date), we might count unique tour dates. But likely each booking is an instance (for private tours) or each tour date is a group booking. We'll clarify this in business logic. A safe approach: count distinct tour dates completed.

2. Total Guests Served: Sum of all participants the operator has guided. This can be derived from bookings as well (sum of `booking.guests_count` for all completed tours). - We can maintain this similarly via triggers: when a booking is completed, add its guest count to a running total. For example, `UPDATE operators SET guests_served = guests_served + NEW.guest_count`. - If a booking is canceled or modified, subtract accordingly. - Alternatively, compute by joining all bookings and summing. But again, denormalizing for quick reads is fine given these stats are frequently displayed and not too heavy to update. - This stat shows scale of experience (an operator might have done 50 tours with 200 total guests, etc.).

3. Years on Platform: This is straightforward – we have the date the operator account was created (we should have a `created_at` on `operators`). We can simply display "Member since 2023" or calculate years since that date. No need to store a separate number, it's derived on the fly (one calculation per profile view is trivial). If we want to be exact, "Years on platform" = floor(now - created_at in years). - We may still store the `created_at` timestamp in `operators` when the record is inserted (Supabase can auto-gen this). That suffices.

4. Response Time: This measures how quickly the operator responds to messages or inquiries. To compute this, we need message data: - Each inquiry message timestamp vs the operator's first response timestamp. We could log these differences and average them. - Possibly maintain a rolling average or median. This might be better done outside the core DB (like a periodic job computing last 30-day avg response). - But we can do: have a table `inquiries` with columns `first_response_time` (interval). Each time an inquiry thread closes or gets a response, record the time difference. - Then either compute average via query when needed or maintain an `operators.avg_response_minutes`. - Because this can fluctuate and we might only count recent history (last X inquiries), it might not be a simple trigger sum. A periodic recalculation (like a CRON job daily) might be used to update it. - For MVP, we might skip showing response time until enough data or do a simple approach: monitor timestamp differences in app logic and store an updated average. - If implemented, we'll store it (in minutes or hours) in the operator profile for quick retrieval. The value is updated whenever a new response comes in (using a weighted average formula, or for simplicity a moving average).

5. Booking Confirmation Rate: This is the percentage of booking requests that the operator accepts/confirms. For example, if they received 20 booking requests and confirmed 18, their confirmation rate is 90%. This requires: - Tracking the number of booking requests that were **made** vs **accepted**. Likely the `bookings` table can have a status (e.g. `requested`, `confirmed`, `declined`). - Each time a booking request is made for their tour, increment a counter (requests++). Each time the operator confirms one, increment a confirmed counter. - Then the rate = confirmed / requests (could be stored or computed). - We can store two fields: `requests_count` and `confirm_count` in operators, and possibly a computed column or at least compute the percentage in the app. Or directly store `confirmation_rate` as a decimal. - Triggers: On insert of a new booking (if status = requested and operator_id = X) -> `requests_count++`. On update of booking status to confirmed -> `confirm_count++`. On decline -> could leave requests as is (it was a request, they just didn't confirm; the rate goes down inherently). - If a booking is canceled by user, that might not affect confirmation rate (as it was confirmed already, presumably). - We should also consider time window (maybe lifetime is fine, or we focus on recent history if an operator had early troubles but improved – but that's beyond schema, more of a calculation choice). - For now, lifetime aggregate is simplest.

6. Repeat Customer Rate: This indicates loyalty – what fraction of an operator's customers have booked more than once. To derive this: - We need to know total unique customers and how many of those have at least 2 bookings with this operator. - This is a bit complex to update in real-time via triggers because when a single new booking comes in, it could potentially convert a customer from a one-timer to a repeater. - One strategy: keep a set or count of unique customers per operator and a count of repeaters. * We could maintain an `operator_customers` table listing each customer who ever booked with that operator and how many bookings they've done. On each new booking, upsert the record (increment their count). If their count goes from 1 to 2, that operator's repeat count increases by 1. * Then `repeat_customer_rate = repeat_customers_count / total_customers_count`. - This is a heavier mechanism. For MVP, we might not implement full repeat rate immediately (since it requires historical tracking). Or compute it offline occasionally. - If we do, an `operator_customers` table: (operator_id, user_id, bookings_count, last_booking_date, etc.). When `bookings_count` crosses threshold 2, mark them as repeater. The operator could have columns `total_customers` and `repeat_customers` to compute the percentage. - Alternatively, we just compute it on demand with a query:

```
SELECT COUNT(DISTINCT user_id) as total, COUNT(DISTINCT CASE WHEN
bookings_count>1 THEN user_id END) as repeats
FROM (SELECT user_id, COUNT(*) as bookings_count FROM bookings WHERE
operator_id=X GROUP BY user_id) sub;
```

But doing that for each profile view might be slow if there are many bookings. - Given scale is initially small, it might be acceptable to compute on the fly or periodically cache it.

For now, we can approximate by storing `total_customers` and `repeat_customers` and updating them: - On new booking completion, if it's the customer's first booking with that operator, `total_customers++`. If it's second, `repeat_customers++` (and that customer is now counted as repeat). - We'd need to know if it's their first or second, which we can check via a subquery or an `operator_customers` tracking as mentioned.

Stored vs Computed Trade-offs: We see a pattern: many of these stats can be derived from transactional tables (bookings, inquiries). For performance, storing them in the `operators` table and updating via triggers is a good approach for frequently accessed data [16] [23]. This is denormalization, but it makes the read (displaying the profile) very fast – just one row with all stats precomputed. The downsides are extra writes and complexity on updates (ensuring counters don't go out of sync).

However, since each update corresponds to a user action (booking made, tour completed, etc.), triggers can handle these in real-time. The overhead is small because each event triggers a simple UPDATE, which is efficient if indexed. It's often worth it for the read efficiency [24]. As an example, Cameron Blackwood's implementation of triggers updates review counts and averages on each insert, which he found to work well for moderate volumes [25] [20].

We should indeed use triggers or stored procedures to maintain: - `operators.tours_completed` - `operators.guests_served` - `operators.requests_count` & `operators.confirm_count` - `operators.total_customers` & `operators.repeat_customers` (if doing repeat tracking) - Possibly `operators.avg_response_time` (though this one might be trickier to do synchronously, might do via an async process)

Using **triggers and real-time updates** ensures the stats are always up-to-date when someone views the profile [26]. The alternative is to run a nightly batch to recompute all stats (which is also viable, especially for things like repeat rate that aren't urgent to update immediately). A hybrid approach: use triggers for straightforward counters (tours, guests, etc. which change per booking) and use a daily cron for more complex metrics (response time, repeat rate) if needed.

We will also keep the normalized data (the bookings, inquiries tables) as the source of truth in case we need to recalc everything from scratch (the denormalized fields can always be recomputed if a bug is found) [23]. But day-to-day, we rely on the stored values for quick reads.

Index and Performance Considerations: - We'll ensure indexes on any foreign keys used in triggers. For example, if a trigger checks past bookings of a user for repeat detection, an index on `bookings.operator_id, bookings.user_id` would help. - The counters themselves will be simple numeric columns. Updating them is a constant-time operation. Even at larger scale, this is fine (millions of bookings might cause big numbers, but that's not an issue for Postgres). - If concurrency becomes a concern (e.g. two bookings at same exact time both trying to update the same operator row), we might have to handle possible race conditions. Typically, Postgres row-level locking will serialize those updates safely. But in a high-volume system, sometimes an aggregate table or periodic batch is used to avoid contention. For our scale, likely not an issue. - We might also use materialized views for some stats if we didn't want triggers. But then we'd need to refresh them and it complicates real-time display. The trigger approach is simpler for now.

To sum up, our plan is to **denormalize frequently used stats into the operators table** for performance, keeping them in sync with triggers or background jobs [26]. This ensures the profile page can load all these stats with one query on `operators`, rather than doing multiple joins/aggregations at runtime. We'll document these triggers clearly and test them to avoid any miscounts. If any stat proves too complex for real-time updates (like repeat rate), we'll note it and perhaps update it periodically.

Multi-Region Considerations

If an operator runs tours in multiple regions or service areas, we need to reflect that in the data model.

Modeling Service Areas: We should have a reference for regions (could be states, countries, or specific birding areas). Likely we'll define a `regions` table (with fields like `id, name, type` where type might distinguish state vs country vs park). For example, region could be "Victoria (state)" or "Kakadu National Park". The operator can then have a relationship to many regions: - We create an `operator_regions` join table: `operator_id` + `region_id` (composite PK). This lists each region an operator covers. - If an operator is based in one place but occasionally does tours elsewhere, they might list multiple regions. - This allows customers to filter by region (e.g. show operators in Queensland). - We'll index `operator_regions.region_id` to quickly find all operators in a given region (useful for search queries).

Alternatively, if regions are broad (like just country/state), we could have columns in `operators` like `country` and `states_served` (array). But a join table with a normalized regions list is cleaner for complex querying and avoiding typos.

Using the tours data: We note that each tour will likely have a specific location/region as well. One could infer an operator's regions by looking at all their tours' locations. However, storing it explicitly in `operator_regions` avoids heavy computation and lets operators highlight areas even if no tour is currently listed there (or if they can operate on request in other areas).

Region-Specific Credentials: As mentioned, some credentials tie to regions. For instance, to guide in a particular national park, the operator needs a permit for that park. In our `operator_credentials` table, we included `region_id` for this scenario. This way, we can store: operator 5, type = "permit", region_id = KakaduPark, expiring on date X, verified. If the operator also has a permit for another park, that's a separate row with that other region_id.

This granularity ensures we know which credentials apply to which area. Potentially, when displaying credentials, we can label them like "Permit – Kakadu National Park (Verified)".

It also helps if, say, a certain region's tours should only be shown if the operator has a valid permit for that region. We could enforce in business logic: if an operator's Kakadu permit expired, maybe their Kakadu tours are temporarily not bookable (just a thought). The data model would at least allow such checks.

Multi-Region Tours: If a single tour covers multiple regions (less likely for a day tour, but possible for multi-day expeditions), we might also need a `tour_regions` table. But that's more on the tours side. From the operator perspective, covering multi-regions is handled by listing them.

Data Visibility: Region info is not sensitive – it's meant to be public. So both `regions` and `operator_regions` data can be readable by anyone (to show on profiles or filter). We just need to control who can modify operator_regions: - Likely only the operator (owner) or admin can set which regions they operate in. Possibly via a multi-select in their profile edit form. - We could also auto-populate operator_regions from the tours they list (like each time they add a tour in a new region, add that region). But manual control is fine too.

Example: Operator 5 runs tours in **Victoria** and **Tasmania**. In `regions`: we have entries for Victoria (id=VIC) and Tasmania (id=TAS). In `operator_regions`: we add (operator_id=5, region_id=VIC) and (5, TAS). If later they add a tour in NSW, we add (5, NSW). If they stop serving Tasmania, we could remove that entry (or mark inactive if we wanted history).

Search: This model will support queries like "find operators who serve region X" easily via a join.

In summary, **we incorporate a region mapping** to reflect multi-region operation. This makes the platform scalable beyond single-location operators and sets the stage for possibly region-specific content (like showing region-based credentials as above, or region-specific reviews, etc.).

From a schema viewpoint, it's a straightforward many-to-many relation, which is appropriate for this kind of data (one operator, many regions; one region, many operators). If needed, we can extend regions to hierarchical (country -> state -> park), but that's beyond MVP scope.

Privacy & Visibility

Not all operator data should be public. We must carefully separate what information is shown to end-users versus what is kept private or only for internal use. Here's how we'll handle visibility and privacy:

Public vs Authenticated vs Admin Data:

- **Publicly Visible (to all site visitors):** Essentially the profile fields that help customers decide on a tour:

- Operator name, tagline, description.
- Experience and specialties.
- Regions served.
- Verified badges (but not the underlying documents).
- Media (photos, videos).
- Guides info (names, bios) – these are public as part of profile.
- Reviews and ratings.
- **Contact button** – an interface to send a message, but not the actual contact details.

In general, anything that markets the operator is public. We will implement RLS policies such that these fields can be selected by any user (or we might create a view to limit to these columns for a truly public endpoint).

- **Authenticated Users:** If a user is logged in, they don't necessarily see much more on the profile page itself compared to public. However, being logged in allows them to **send a message or booking request**. We may show a "Contact Operator" form for logged-in users (and prompt login for guests who click it). The actual profile info remains the same. We might later show to logged-in customers if they have any mutual connections or past tours with this operator, etc., but that's beyond current scope.

So, there's not a separate class of data for regular logged-in users vs the public, except enabling interactivity. All sensitive info remains hidden from them too.

- **Operator Themselves (Owner/Guide login):** When an operator logs in to their dashboard, they should see and edit all their own info, including some things the public doesn't:
- Their detailed contact info on file (email, phone) – so they know what the platform has.
- Their credentials list and statuses.
- Possibly internal notes or stats not shown outside (e.g. maybe revenue, etc., though that's more admin).
- They obviously won't have access to other operators' private data, only their own.

- RLS will allow a user to select and update their own operator row (including fields hidden from others) if `operator_members` contains that user and role appropriate.

- **Admins:** Admin users (platform staff) have full visibility into all data:

- They can see operators' contact info, documents, internal notes, etc.
- They can verify credentials, manage any operator's details.
- We will likely have an `admin` role in Supabase (could use JWT claims or a separate table mapping user to admin role).
- RLS policies will grant admins select/update on all tables unconditionally (or we'll use Supabase's service role for backend admin tasks outside RLS).

Contact Information Protection: We do **not** show operators' email addresses, phone numbers, or exact addresses on the public profile [12]. This is both to prevent spam/unwanted contact and to **prevent off-platform bookings (leakage)**. Platforms like Airbnb conceal contact details in messages until a booking is confirmed [12]. We'll adopt similar logic: - The profile page will have a **"Contact Operator"** button instead of any direct info. This launches an on-platform messaging thread. - Our messaging system can even automatically block attempts to share phone/email until after booking, if we want to strictly enforce it (common in marketplaces [12]). - After a customer makes a booking (or at least a firm inquiry), we could then share the operator's phone (for coordination), likely via the booking confirmation page or email. That part is transactional, not on the public profile.

In the database, we'll still store contact info: - e.g. `operators.business_email`, `operators.phone_number`, `operators.address` (for admin use; address might be needed for insurance or legal). - These columns will be protected by RLS: only admins and the operator themselves can select them. We can implement column-based RLS by splitting sensitive fields into a separate table (say `operator_private`) with one-to-one relation to operators, and then limit access to that table. This physically separates public vs private fields. Alternatively, keep them in `operators` and use a policy that returns NULL or nothing to unauthorized queries. Simpler: use a separate table or simply ensure our API never exposes those fields except on privileged routes.

Business Registration Details: If we require operators to provide an ABN, tax ID, or other registration info, that will also be stored privately. For instance, `operator_private` could have `company_number`, `registered_name`, etc. These are for compliance and not shown to users. Only admins might use it (e.g., to verify the legitimacy of the business). We will not expose it in any public context.

Profile Editing: Operators will edit their profile through a secure form. They can change public fields (description, etc.) and upload documents. We'll enforce via RLS that one can only update their own operator row and related data. For example,

```
UPDATE operators SET ... WHERE id = X
  ALLOWED IF X IN (SELECT operator_id FROM operator_members WHERE profile_id =
auth.uid AND role in ('owner','admin'))
```

(This ensures guides maybe can't change core profile unless we give them permission. We might restrict editing to the owner or a designated manager role.)

Visibility of Stats: Most of the stats (tours completed, etc.) are fine to show publicly – they act as proof of experience. However, if an operator has a mediocre stat (like a low confirmation rate), showing it could be sensitive. Since the question includes them, presumably they want them public as part of transparency. We will show aggregate stats to users (like "50 tours completed, 200 birders served, 95% confirmation rate, responds in 4 hours on average"). These serve as quality signals. If any stat is too internal (maybe repeat customer rate might not be shown publicly, but could be used internally or to award badges), we can decide that in the product spec. From a data perspective, we have them, but decide later which to expose.

Sensitive Media or Info: If operators upload any documents, those are never public. If they upload extra photos or content not intended as marketing (unlikely, since media gallery is explicitly for public), we'll treat all media uploads to the gallery as public by default. We will guide them to not upload anything confidential there.

Admin Notes: We might maintain an internal notes field (e.g., if an admin wants to record an issue about an operator). That would be another admin-only field or table (like `operator_notes(operator_id,` `note, author, created_at)`). Only admins can see this. Not core to architecture, but worth noting as a possibility.

Citations & Precedents: The rationale for these privacy measures is backed by industry practice. Airbnb's concealment of contact info was mentioned: *"Airbnb's strategy of concealing contact details in messages aims to protect users… and ensure transactions occur within their ecosystem."* [12]. This aligns with our plan to keep direct contact info hidden until necessary. Similarly, many platforms provide a messaging system to handle inquiries safely [12].

We will implement content filters in messaging if needed to prevent obvious sharing of emails/phones until booking (if we foresee leakage issues). But even without that, by not putting contacts on the profile, we drastically reduce off-platform communication.

RLS Implementation Summary: - Table `operators`: One policy for public select (allow everyone to select specific columns that are non-sensitive). This might require defining a Postgres view or using column-level privileges, since RLS conditions can't easily allow only certain columns. A simpler approach: we use the backend (API level) to only expose those fields to normal users. For example, our supabase query for public profiles will explicitly select allowed columns. Meanwhile, we have a stricter policy that disallows selecting the row entirely unless: - condition A: TRUE for admin or operator member (full access) - condition B: (maybe TRUE for all for now, since blocking it would block profile viewing – instead, we might not rely on RLS for column filtering, but on the API). - Alternatively, maintain two separate endpoints: one RPC or view for public consumption (which selects only safe fields), and the base table secured for normal users. This is a common pattern to safely expose a subset of data. - Table `operator_credentials`: No select for anonymous or regular users. Only operator's members and admins can select their rows (and even then, operator members might only see `type/status`, not the file itself – though they provided it, so it's fine). Admin can select all. - Table `operator_members`: Probably not exposed publicly except via join to show guide names. We can allow a safe view for showing guides (which joins with profiles and returns name, role). But the raw membership table might be restricted to members and admin only (since it's mostly internal). - The rest (reviews, media, etc.): Reviews are public data (but we keep reviewer personal info limited to maybe username). Media images are public links.

Conclusion: By designing with privacy in mind, we ensure that operators' sensitive info is protected, building trust both with operators (that their private details are safe) and with customers (who get the info they need without extraneous details). Our data model separates public-facing profile content from confidential records, and we leverage RLS policies to enforce these boundaries.

In practical terms, an admin interface will have access to everything (bypassing RLS via service role or by being an admin user), whereas the front-end client will only request and get the intended public fields. This dual approach keeps our platform compliant with privacy expectations and platform integrity (preventing users from taking transactions off-platform easily).

---

[1] [2] [3] Choosing Between Database Tables — Separate Vs Single for Admins and Users | by Mohammed Muwanga | Medium
https://muwangaxyz.medium.com/choosing-between-database-tables-separate-vs-single-for-admins-and-users-97673e8de457

[4] [5] [6] [7] Building the optimal user database model for your application - DoneDone
https://donedone.com/building-the-optimal-user-database-model-for-your-application/

[8] Receiving Custom Quote Requests – SafariBookings Help Center
https://help.safaribookings.com/hc/en-150/articles/18462550286365-Receiving-Custom-Quote-Requests

[9] [10] [13] [14] Make a Profile That Dominates TripAdvisor | Tourism Tiger
https://tourismtiger.com/blog/tour-operators-12-steps-to-make-a-profile-that-dominates-tripadvisor/

[11] Profiles Explained - Trip Solutions
https://trip-solutions.helpscoutdocs.com/article/1228-profiles-explained

[12] Understanding Platform Leakage and Steps to Prevent it
https://www.cometchat.com/blog/platform-leakage

[15] [16] [19] [23] [24] [26] mysql - Denormalize to reduce the use of aggregate functions (calculations) such as COUNT and AVG: worth it? - Stack Overflow
https://stackoverflow.com/questions/60932123/denormalize-to-reduce-the-use-of-aggregate-functions-calculations-such-as-coun

[17] [21] database - How to store feedback like stars or votes of users with efficiency? - Stack Overflow
https://stackoverflow.com/questions/54571991/how-to-store-feedback-like-stars-or-votes-of-users-with-efficiency

[18] [20] [25] Using Supabase Triggers To Effortlessly Update Your Database
https://www.cameronblackwood.xyz/blog/using-supabase-triggers-to-effortlessly-update-your-database

[22] database - How to structure a table for detecting expired licences in mysql? - Stack Overflow
https://stackoverflow.com/questions/11322686/how-to-structure-a-table-for-detecting-expired-licences-in-mysql