ChatGPT

Operator Profile Data Architecture

1. Operator Entity Design

Separate `operators` table vs extending `profiles`: It's best to introduce a dedicated `operators` table to represent the tour operator (business) entity, rather than trying to cram operator fields into the user `profiles` table. A separate table keeps concerns clear and avoids loading business-specific data every time we query a user. In practice, you often *don't* need all profile attributes when dealing with user references, so separating them keeps things lean [1]. It also future-proofs the design – profiles might later represent different entities (e.g. organizations or groups) beyond individual users [2], which is easier to handle if we decouple business profiles from personal profiles.

Relationship between person and business: We model a **one-to-many** (or many-to-many) relationship between user profiles and operator businesses. In the initial MVP, assume one user owns one operator (1:1), but design the schema to allow expansion. For example, one user might create/own multiple operator accounts in the future (or a guide could work for multiple companies, if ever allowed). To support multiple team members managing an operator, use a join table (e.g. `operator_members`) linking `profiles` to `operators` with roles. Each entry in `operator_members` has a `profile_id` (user) and an `operator_id` (business), plus a role field (e.g. `owner`, `admin`, `guide`). This structure supports **personal accounts vs team accounts** similar to other platforms – a "personal" operator account (one user = one business) or a team operator with invites [3].

Solo operator vs company with guides: In the solo case, the user's profile essentially corresponds to the operator (they are the owner and also the guide). We still create an `operators` record (with that user as owner) so that even a one-person business is represented in the `operators` table. For a company, the `operators` record represents the business, and multiple user profiles can be associated as guides or staff. The `operator_members` table captures these relationships. For MVP, you might automatically create an operator record when a user is flagged as an operator and insert a member entry linking the user as the owner. This is more robust than the current `is_admin=true` flag hack – instead of overloading `is_admin` (which should be reserved for platform admins), define roles explicitly in the membership table or as a new column (or use Supabase Auth custom claims for roles). This way, platform admins (Webird staff) remain distinct from operator roles.

Schema outline:

- **profiles** (user profiles): `id (PK)`, user personal info... (The existing table, possibly extended with general fields like name, avatar, bio, etc.).
- **operators**: `id (PK)`, **owner_id** (FK to `profiles.id` for primary owner), `name` (business name), `slug` (unique URL slug for public profile), `tagline`, `description`, `logo_url`, `created_at`, etc.
- **operator_members**: composite PK of `operator_id` & `profile_id`, plus `role` (enum: owner, admin, guide). This table lets multiple users access/manage one operator. For MVP, it will typically

have one entry (the owner), but it enables inviting team members later. Ensure a unique constraint that an operator has only one primary owner (or use a boolean flag or the `role` to identify the primary owner).
- (Alternatively, one could use a single `profiles` table with a role column to mark operators, but that complicates multi-user management and mixes personal vs business data. The separate table approach is more normalized and scalable [1] .)

Behavior: The `tours` table should reference `operators` (not profiles) going forward. Every tour belongs to an operator business. In the new schema, replace or supplement the `tours.operator_id` foreign key to point to the `operators.id`. This ensures all tours are tied to an operator (and indeed all tours on the platform should belong to some operator – webird.ai itself doesn't create tours, as clarified). This foreign key should be **NOT NULL** (no "orphan" tours) and indexed for performance when querying an operator's tours. If you migrate existing data, you would create operators for each user who had `is_admin=true` and update their tours to use those operator IDs.

For referential integrity, define cascades appropriately: e.g., if a user profile is deleted (though typically we'd soft-delete or deactivate operators/users rather than hard-delete due to historical data), decide whether to also delete their operator or just orphan it. Likely we will **not** auto-delete operators on user delete – instead, maybe prevent deletion if they own an operator, or transfer ownership – because tour records and reviews should persist. Similarly, avoid deleting operators if they have tours/bookings; use an `active` flag for removal from listings instead.

2. Operator Profile Fields

Design the `operators` table (and related tables) to capture rich information needed for a public operator profile. Key categories of fields:

- **Identity & Branding:**
- **name** – The business name or operator name (text). For a solo guide, this could be their personal name or a brand name.
- **slug** – URL-friendly unique identifier for the public profile (text, unique, indexed). e.g. `outback-birding-tours`.
- **logo_url** – URL or path to the operator's logo or profile image (stored in Supabase Storage).
- **tagline** – A short tagline or motto (text, e.g. *"Expert birding tours in Queensland"*).
- **description** – A detailed description/about section (rich text or long text). This is the main bio of the company, describing their background, philosophy, what they offer, etc.
- **established_year** or **years_experience** – Optionally, a field to denote how long they've been operating (could be an integer year founded, or computed from another data point). This helps highlight experience (e.g. "Operating since 2010").

- **tier/status** – If you have a tier system for operators (e.g. new, experienced, elite), you might include a field or enum for that. (This wasn't explicitly requested, but since user profiles have tiers, you might similarly classify operators – e.g. a "Premier Partner" badge. This can be handled via a field or a computed flag based on performance metrics.)

- **Credentials & Legitimacy:** Operators often need to show they are qualified and legitimate. Instead of many columns on `operators` for each credential, it's better to use a separate table for credentials (see section 5). However, the operator profile should surface the existence/status of these credentials:

- We will maintain an `operator_credentials` table for details (licenses, permits, insurance, etc.), including file storage references. On the public profile, we might show high-level badges/flags such as "Licensed Guide ", "Insured ", etc. The `operators` table can have boolean flags or a JSON field for verified credentials for quick access (or we can compute/join from the credentials table). For example, `is_insured` (bool), `is_licensed`, etc., set true once verified. This denormalization is optional – one can also just query the credentials table when loading the profile. The trade-off: a flag in `operators` allows a simple read of all profile info in one go (single table read) [4], whereas always joining on credentials might be a bit slower but keeps data single-sourced. Given these flags change only when credentials are updated (not frequently), storing them in the operator record (updated via triggers when a credential is verified/revoked) is reasonable.

- **Verified status/badges:** Beyond credentials, you might have an overall "Verified Operator" indicator if the platform does any identity or quality verification. For example, once an operator uploads required documents and an admin approves them, mark them as verified (could be a field `is_verified_operator` on `operators`). This can control visibility (perhaps only verified operators are shown publicly) and display a badge on their profile.

- **Experience & Expertise:** These fields highlight what the operator specializes in and their track record:

- **years_operating** – If not using established year, you might store directly the number of years they claim to have been operating. Alternatively, compute it from established_year or from the earliest tour date/completed booking. Storing a claimed value might be simplest (operators can input "20 years experience"). This is more of a display item.
- **regions_served** – The geographic regions the operator covers (discussed more in Multi-Region section). We likely won't store a simple text in `operators` for multiple regions because we want to filter/search by region. Instead, use a related table or array. For example, an `operator_regions` join table linking to a `regions` reference table (with entries like "Northern Territory" or "Kruger National Park"). The operator profile page can list the regions (or service areas) from that relation. This approach scales if an operator operates in multiple distinct areas.
- **tours_completed** – Not a static field input by operator, but a stat we will maintain (see section 6). It indicates how many tours/bookings they have completed via the platform. Displaying this on the profile (e.g. "**Tours completed:** 37") adds social proof. We can choose to materialize this in the database (updated via triggers as bookings complete) for performance.
- **specialties** – The birding specialties or unique expertise. This could be a text field or a structured list:
    - If it's free-text, the operator can write something like "Specializes in raptor identification and pelagic bird trips."
    - For more structure, you could have an `operator_specialties` table or use tags (like a predefined list: e.g. "Rainforest Birds", "Pelagic Seabirds", "Photography tours"). This allows filtering by specialty in the future. For now, a simple comma-separated or JSON array of keywords in `operators.specialties` could suffice, which you later normalize if needed.

- **languages** – Languages spoken by the operator/guide (important for international clientele). This could also be structured or simple:

  - A separate table `operator_languages` (with entries like “English”, “Spanish”) linked to `operators` would let you join and filter easily (and ensure consistent naming).
  - Or use a text array column `languages text[]` in `operators` and a GIN index for search. Given the likely finite set of languages, a join table is fine.
  - Each guide (if multiple) might have different language abilities – to keep it simple, assume the operator as a whole can accommodate these languages (the team covers them). If needed, you could track languages per guide in the future.

- **Equipment & Capacity:** For operators offering boat tours or requiring special gear:

- **equipment_description** – A free text or structured JSON field describing key equipment provided. For example, “4x4 vehicles, spotting scopes and binoculars available, photography blinds” etc.
- **vessel details** – If applicable (for boat operators), store details about their vessel(s). If most operators only have one vessel, you could include fields in `operators` like `vessel_name`, `vessel_type`, `vessel_capacity`, etc. For multiple vessels or complex cases, an `operator_vessels` table might be better:
  - `operator_vessels(operator_id, name, type, capacity, description, photo_url)`
  - That allows listing multiple boats (say a company has two boats). In MVP, you might not need this complexity; one primary vessel can be stored as part of the profile or in the tour description. But the schema can accommodate it.

- **capacity** – General capacity info (like max group size). This is often tour-specific (each tour listing might have its own capacity), so it might not belong on the operator profile globally. If the operator only does private tours for small groups, they might want to say “Up to 4 guests per tour” – that could be in description or a field. Use your judgment; tour-specific capacity lives in `tours` table (e.g. a field there), while an operator-level “capacity” could be a typical or overall limit if needed.

- **Contact & Location:**

- **base_location** – The operator’s primary location (e.g. city, state, country). This can be a free-text field for display (e.g. “Cairns, Australia”). We might also store separate fields: `city`, `state_region`, `country` for clarity, or a foreign key to a locations table if we have one. But free text is acceptable if not used for filtering. This is mainly to give users an idea where they are based.
- **service_area** – This ties into the regions served. The operator might want to list multiple areas (covered via `operator_regions` as above). On the profile, you might display “Tours in: Kakadu National Park, Kakadu surrounds, Darwin area”.
- **contact_email / phone** – **Do not expose personal contact info publicly.** The platform should handle communication. We likely won’t store a public email or phone in the `operators` table for display. Instead, use a platform messaging system (e.g. users click “Contact operator” which sends a message internally).
  - Internally, however, you may store contact details for administrative purposes or emergency. For instance, have fields `contact_email`, `contact_phone` in `operators` that only admins or the operator can see. These would be used to send notifications or in case of

urgent contact post-booking. If you do store them, mark them as **private** (not returned to clients via the public API). Row-Level Security (RLS) rules or column-level security should ensure normal users can't read these fields [5].
    - Platform messaging: ensure the design funnels inquiries through an `messages` table or similar. You might not need to reflect that in this schema design beyond noting that direct contact is hidden.

- **website or social links** – If you allow the operator to link an external website or social media:

    - Provide fields like `website_url`, `facebook_url`, `instagram_handle`, `twitter_handle`, `youtube_url`, etc. These are optional and can be shown as icons/links on the profile. Keep in mind, allowing a direct website link might let users bypass the platform for booking. Many marketplaces restrict this. Perhaps allow social media for credibility, but not a direct booking link. This is a business decision.
    - Alternatively, use an `operator_links` table for flexibility (with columns: operator_id, type, url), where type could be "website", "facebook", etc. But a few explicit fields might suffice for now.
    - These links (if provided) are public info, but you might want to review them to ensure no direct contact info is given (or use nofollow). This is more of a policy issue.

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

- High-quality media will make the profile engaging. You should allow operators to upload a gallery of photos and possibly a promo video.
- **Photos:** Create an `operator_photos` table: `id, operator_id, file_url, caption, sort_order`. This lets an operator have multiple images (wildlife seen on tours, their guides, equipment, etc.). You can then display an image carousel on the profile. Storing images in Supabase Storage, you'd keep the file path or a public URL in `file_url`. Keep metadata like caption or alt text if needed. The `sort_order` can manage display order.
- **Videos:** If operators have a short intro video, you might allow a YouTube/Vimeo link or let them upload a video. Streaming video from storage is heavy, so a link to YouTube might be simpler. You could have a field `video_url` on `operators` or a general `operator_media` table with a type ("video" vs "photo"). For now, possibly allow a single `intro_video_url`.
- All media records should reference the operator and probably be public. (Ensure storage bucket rules allow public read if these are to be displayed to unauthenticated users on the site.)

With these fields, an operator's public profile page can show: - Name, location, logo, tagline at top. - Gallery of photos. - Description/About section. - A **Credentials/Verification** section showing icons for things like "Licensed" or "Insured" (with a tooltip or label, possibly the year or authority if you want detail). - A **Key stats** section (tours completed, years on platform, average rating, etc. see section 6). - **Tours offered** (list or link to their tour listings). - **Team/Guides** (if multiple guides, see next section). - **Reviews** section with customer ratings. - **Contact** button or form (platform-mediated). - Social media links.

Ensure to index or optimize for common queries: - We will often query operators by `slug` (for profile page) – index slug (unique). - Listing operators by region or specialty – index through join tables (e.g. an index on `operator_regions.region_id`). - If filtering by languages – index or use a join table. - The foreign keys (like `tours.operator_id`, `operator_members.operator_id` etc.) should be indexed to quickly retrieve related records (Postgres automatically indexes primary keys, and foreign keys can use those, but explicit index on `tours.operator_id` speeds up "find all tours for operator X" queries).

3. Guides vs Operators (Team Members)

In a business (operator) that has multiple guides or staff, we need to represent each guide's profile under the operator. This adds transparency for customers ("Meet the guides" section) and allows permission control (each guide might log in to handle tours or messages).

Modeling guides: As mentioned, the `operator_members` join table associates user profiles with an operator and assigns a `role`. Use roles such as: - `owner` – the primary account owner (full permissions, listed as business owner). - `admin` – (optional) a manager role; similar permissions to owner but not the original creator. Could edit tours, view bookings, etc. - `guide` – a tour guide who works for the operator. They might not manage the account settings but could have access to relevant bookings or chat. - (Potentially other roles like `staff` or `editor` as needed.)

For MVP, you might only use `owner` and `guide`. The `operator_members` table will have one row for the owner at creation. In future, an owner could invite guides: that would create new user accounts (or use existing user accounts) and add rows with role=guide.

Guide mini-profiles: On the public operator page, list each guide with some basic info. We can leverage the `profiles` table for this, since each guide is a user. Likely, your `profiles` table already has fields like `full_name`, `avatar_url` (photo), and maybe a short bio or title. If not, consider adding a `bio` field to profiles to allow a short introduction. For guides, relevant info to show includes: - **Name** and **photo** – The guide's name and a headshot (profile avatar). - **Title/role** – e.g. "Lead Guide" or "Birding Guide". - **Guide bio** – A brief description of their background (e.g. "John has 15 years of birding experience and specializes in raptors."). This could be the `bio` field from their profile or a separate field in the membership table. If guides can edit their own profiles, using `profiles.bio` is straightforward. Alternatively, the operator owner might set a description for each guide. You could add a `bio` field in `operator_members` specific to that operator context. For simplicity, using the user's profile bio means the guide would need to fill it themselves (or the owner logs in as them). It might be easier to store guide info in a separate table (like `guides` table) if you want owners to manage it without separate logins. However, since we plan for multi-user, encouraging each guide to have a real user account and profile is good practice. - **Certifications or specialties** – If individual guides have specific certifications (e.g. a guide personally is a Certified Birding Guide or Wilderness First Responder), you can note that. This can simply be mentioned in their bio, or you could have a small icon list by their name. To model it, you might reuse the `operator_credentials` table but link to a guide's profile for personal certs. That might be overkill unless needed. Usually, the company's credentials suffice; personal certs can be textual. - **Languages** – If one guide speaks German and another doesn't, you might list the languages per guide. This could again go in their bio or a separate small field. Possibly your `profiles` table could have a `languages` array for the user as well, if you want.

To display guides, fetch all `profiles` that are in `operator_members` for that operator (excluding those who might not be customer-facing, like an "admin" who doesn't guide – maybe filter role=guide or include owner if the owner also guides, which in many cases they do). Then for each, show their photo, name, role, and bio snippet.

Example schema addition: - `operator_members.profile_id → profiles.id` (FK, indexed) - `operator_members.operator_id → operators.id` (FK, indexed) - `operator_members.role` – text or enum. - (Optional) `operator_members.guide_bio` and `guide_photo` if you want to store guide info separate from the profile. But it's probably unnecessary duplication if the profile already has these fields.

Customer value: Showing guides builds trust (the customer knows who might be leading their tour). It's especially useful if the operator is a larger outfit with multiple guides – each guide could have different expertise. For instance, *"Alice – shorebird expert, 10 years experience"*, *"Bob – specializes in owls and night tours"*. This kind of info (expertise, years) can be worked into each guide's bio or listed as bullet points under their name.

Permissions: With the membership model, you can later allow: - Owner/admin can manage all content (edit operator profile, tours, see all bookings). - Guides (non-admin) might have more limited access (maybe only the tours or bookings they are assigned to, if you assign guides to specific tours/activities). - All members can at least view the operator's dashboard and upcoming trips, depending on your RLS rules.

Setting up RLS such that a guide (role=guide in `operator_members`) can read/write certain data (like mark a tour as complete, respond to messages) will be important. We'll discuss RLS more in section 8, but essentially you'll use membership to authorize actions: e.g. a guide should satisfy something like `operator_id IN (SELECT operator_id FROM operator_members WHERE profile_id = auth.uid)` to access that operator's records [3].

In summary, **yes**, guides should have mini-profiles listed under the operator. Use the existing user profile data where possible, extended by any operator-specific info if needed. This design allows a flexible transition from a solo business to a team business without schema change – just additional rows in a join table.

4. Reviews & Ratings Architecture

An operator's public profile will showcase customer reviews and an overall rating. We need a robust review system with proper linking to actual bookings (to ensure authenticity) and support for responses.

Review entity: Create a table (e.g. `reviews`) to store reviews of tours/operators. Each review should be tied to a **specific completed booking**: - **booking_id** – FK to the `bookings` table (or whatever table tracks a customer's completed tour reservation). This enforces that a review is associated with an actual tour experience. - **rating** – an integer or smallint (e.g. 1–5 stars). - **review_text** – text field for the review comments. - **created_at** – timestamp of review submission. - **user_id** – (optional) the author's user ID. You can derive this via the booking (since booking has a `user_id` for the customer), so it might be redundant. Storing it can make it easier to fetch the reviewer's name in one query though. If stored, ensure consistency (you could add a CHECK that review.user_id = booking.user_id). - **operator_id** – (optional) the operator being reviewed. This too can be derived: booking -> tour -> operator. We might include it for indexing convenience: if we often fetch all reviews for an operator, having `operator_id` denormalized here with an index will avoid a multi-join query. However, if we keep data normalized: we'd query `reviews JOIN bookings JOIN tours` to filter by tours.operator_id. That's two joins, which is not terrible with proper indexes (index on bookings.tour_id and tours.operator_id). Including `operator_id` in reviews could simplify queries at the cost of maintaining the consistency (we'd need a trigger to set operator_id when inserting a review, looking up the tour’s operator). Either approach works; for clarity and minimal redundancy, you can omit operator_id in `reviews` and use joins for now.

- **Primary key** – could be a simple serial/UUID. Also enforce **one review per booking** (e.g. UNIQUE(booking_id)), so a customer can’t review the same tour twice. The booking itself identifies the operator, so no need for a composite PK with operator; just ensure uniqueness per booking.

Tour reviews vs operator reviews: We consider a review essentially as reviewing the experience which is tied to both a specific tour listing and the operator’s service. In a marketplace like this, typically the review is shown both on the specific tour page and on the operator’s overall profile (aggregate). We can treat them as one entity: - Each `review` is linked to a `booking` (hence a specific tour). - On an operator’s profile, you can display all reviews from all tours that operator runs, perhaps with the tour name mentioned (“Review of *Rainforest Dawn Walk*: …”). On a tour detail page, you’d filter reviews where booking.tour_id = that tour. - No separate “operator-only” review needs to exist; it's all derived from tour experiences. This prevents people reviewing an operator without booking.

If you did want to allow an overall operator review not tied to a single tour (some platforms do, but it’s less common), you could allow `booking_id` to be NULL and have an `operator_id` for free-form reviews. However, **we strongly prefer reviews be tied to actual completed tours** for authenticity (verified reviews). So assume every review has a booking.

Aggregate rating: We’ll want to show the operator’s average rating (e.g. 4.8 stars) and total number of reviews. We should calculate and store these for performance: - **ratings_count** and **average_rating** fields on the `operators` table (or in a separate stats table). - Whenever a new review is inserted or an existing one updated/deleted, update these aggregates. This can be done via triggers or database functions. For example: - After insert: increment the count and update the average = (old_average * old_count + new_rating) / new_count. Or simpler, recompute by aggregating all that operator’s reviews (but that’s less efficient per insertion if many reviews). - Similarly handle deletion or update (recalculate or maintain sum of ratings). - Storing the average as a decimal (perhaps numeric(3,2) for x.xx) and count as integer is fine. Some choose to store the sum of ratings and count to avoid rounding issues and compute average on the fly; either works. Given the number of reviews likely isn’t huge per operator, triggers recalculating the average exactly are acceptable.

There’s a design decision whether to put these aggregates in the main `operators` table or a separate table (like `operator_stats`). Keeping it in `operators` means one-stop read for profile info [4], but means more frequent writes to that table as reviews come in. A separate `operator_stats` table (operator_id, avg_rating, review_count, etc.) would isolate those writes (treating them as transient/cached data that can be recomputed if needed [6]). In practice, updating a few numeric fields on `operators` isn’t a big load, so putting them on `operators` is fine for simplicity. If you anticipate a *lot* of writes or want to avoid bloat on the main table, the separate table approach is an option [4].

Review responses: Allow operators to respond to reviews (publicly). This is a common feature (e.g. owner can post a reply to a review). Implementation: - Easiest is to add columns to `reviews`: e.g. `response_text` (text) and `response_date`. By default null. When an operator (or staff) posts a response, fill these fields. Only allow one response per review (which is typical). - Alternatively, a separate table `review_responses(review_id, operator_id, response_text, date)` could be made. But since it’s one-to-one, embedding it in `reviews` is fine and simpler to fetch in one query. - Use RLS or logic to ensure only the operator that *was reviewed* can insert/update the response (and maybe only once). For example, the policy for updating reviews could allow if `auth.uid` is a member of the operator that corresponds to the review in question. - When displaying, you'd show the review followed by "Response from operator: …".

Verified purchase enforcement: Only allow users who actually went on a tour to review. This is inherently enforced by tying reviews to bookings: - A user can only create a review if they have a `booking_id` for which they were the customer and the booking is completed. We can enforce this with RLS policies on `reviews`: - For INSERT: `booking.user_id = auth.uid AND booking.status = 'completed'` (assuming bookings have a status). That ensures they can only review their own completed tour. - Optionally also check that the booking's tour date has passed or is marked completed to prevent reviewing before the tour happens. - Additionally, the unique constraint on booking prevents multiple reviews. If you want to allow editing a review, you'd give them UPDATE permission on their review row as well. - This design inherently gives a "Verified Review" system (like Amazon's Verified Purchase) because each review can be assumed to be from a paying customer. You don't even need a separate boolean flag; the presence of a booking link is the proof. On the frontend, you might still label reviews as "Verified" to make it clear.

Tours vs Operator ratings: Should we maintain separate tour ratings? Possibly each tour listing could show an average rating as well. You could compute tour-specific averages from reviews tied to that tour. This can be done on the fly or stored in the `tours` table (fields like tour.avg_rating, tour.review_count) via similar triggers filtered by tour. This is optional but likely useful (customers browsing a specific tour will want to know its rating). If implemented, ensure consistency: whenever a review is added, update both the operator's aggregate and that specific tour's aggregate.

Database schema:

- **reviews**: `id (PK)`, `booking_id (FK bookings.id)`, `rating smallint`, `review_text text`, `response_text text NULL`, `created_at timestamp`, `response_date timestamp NULL`.
- Index `booking_id` (unique).
- (If not storing operator_id in reviews, have an index on booking_id, and ensure bookings has an index on tour_id, and tours on operator_id for the join path.)
- If storing operator_id: index that for quick lookup of all reviews by operator.
- If storing user_id: index that if you need to find all reviews by a user (less common).
- **operators** (extended): add `avg_rating numeric`, `review_count int` (and possibly `rating_sum` if using sum).
- **tours** (optional extension): add `avg_rating`, `review_count` similarly if desired.

Verified linking and RLS: The design ensures authenticity, but double-check RLS: - On `reviews` table: - Allow SELECT to everyone (reviews are public to any viewing the profile). - But possibly you might restrict viewing to authenticated users only. Generally, reviews can be public. - Allow INSERT only to the booking's user as described. - Allow UPDATE only for: - The reviewer (to edit their review text or maybe delete) – if you permit that. E.g. `auth.uid = review.user_id` with some time limit logic if needed. - The operator's members for adding a response – ensure they cannot change the rating or original text. This can be done by a WITH CHECK that `new.rating = old.rating AND new.review_text = old.review_text` when an operator is updating, and only allow them to set `response_text`. You might instead handle responses via a secure function call that only inserts the response to avoid messing with policies. - On `bookings` table: ensure that the policy allows a user to select their own booking and see its status (so the above check can be evaluated). Supabase RLS can include subqueries, e.g., the reviews policy can do `EXISTS(SELECT 1 FROM bookings b WHERE b.id = new.booking_id AND b.user_id = auth.uid AND b.status='completed')`. Another approach is to have a Postgres function that validates and inserts a review, to encapsulate the checks.

Moderation: Also consider if you need an admin override to remove or hide inappropriate reviews. Adding a `is_hidden` or `flagged` boolean to reviews could allow admins to mark a review as not publicly visible (without deleting it). In RLS, you'd then add `AND is_hidden = false` to the SELECT policy for normal users. Admins could still see it.

5. Verification & Trust Flags

Building trust is critical in such a platform. We need to model various verifications: operator-provided documents and admin approval, as well as status flags to display.

Credential storage: Introduce an `operator_credentials` table (or call it `operator_documents` or `certifications`). This table will store records for each credential or document the operator provides. Structure: - `id (PK)` - `operator_id (FK operators.id)` - `type` – text or enum specifying what kind of credential it is. For example: `"insurance"`, `"guide_license"`, `"boat_permit"`, `"first_aid_cert"`, `"business_registration"`, etc. It's useful to have a predefined list of types to ensure consistency. You might create a small lookup table or Postgres ENUM for credential types. - `title` or `name` – a short human-readable label for the credential if needed (e.g. *"Queensland EcoTourism License"*). This could also include the issuer or purpose. - `file_url` – text URL or path to the uploaded document in Supabase Storage (e.g. path to a PDF or image of the certificate). You might separate a `file_name` and have a standard bucket path like `operators/{operatorId}/certifications/{filename}`. - `issued_date` – date the credential was issued (optional). - `expires_at` – date of expiration (if applicable). Many documents like insurance or permits expire. We store this to trigger renewals and to know when a credential is no longer valid. - `status` – an enum or text: e.g. `"pending"`, `"verified"`, `"rejected"`, `"expired"`. Initially when an operator uploads a doc, status = pending. An admin reviews it and flips to verified or rejected. If a credential expires, you could automatically mark it expired (via a cron job or by checking the date when displaying). - `verified_at` – timestamp when it was verified. - `verified_by` – FK to an admin profile (if you have an internal admin user system) or a text name of admin. - `notes` – text for any comments (e.g. why rejected or any special info).

You can also allow multiple entries of the same type if needed (e.g. if they have two different permits), or you could enforce one per type per operator (maybe using a unique index on operator_id+type).

Examples: - An operator might have: type=`insurance`, file_url=`.../insurance.pdf`, expires_at=`2025-12-31`, status=`verified`. - Another: type=`guide_license`, status=`verified`. - Another: type=`national_park_permit`, region_id=XYZ, status=`pending` (if awaiting admin review).

Region-specific credentials: The design should handle credentials that apply only to certain regions or tours. For example, if an operator runs tours in National Park A and B, each park might require a separate permit. We can add a `region_id` column in `operator_credentials` to indicate if a credential is specific to a region (FK to your regions table). For instance, `type="park_permit", region_id=ParkA`.

Then: - When listing credentials on the profile, you might show “Permit for Park A – Verified”. - If an operator tries to add a region to their service areas (operator_regions) without a corresponding permit credential, you have a decision: you might allow it but show not verified, or require the credential first. This could be enforced at the app layer or via some constraint logic. A soft approach: allow it, but on profile indicate “Permit for Park A: Not verified” (trust flag missing). A stricter approach: have admin approval step before that region is published.

This design allows an admin or automated check to verify *for each region that needs special credentials*.

Verified status flags (Trust badges): As mentioned, you likely want to show on the profile which credentials have been verified: - You could compute boolean flags like `has_insurance_verified`, `has_license_verified` etc. upon verification. However, that duplicates the info in `operator_credentials`. An alternative is to query the credentials table for all `status='verified'` entries and display accordingly. Given an operator might have only a handful of credentials, a join or subquery per profile load is fine. - For quick filtering (e.g. “only show operators who are fully verified”), you might have a single `is_verified_operator` flag on `operators` meaning they have met all required verifications. The criteria for that could be: they have an insurance and a guiding license verified (and whatever else you consider mandatory). The admin can set this flag when an operator’s onboarding is complete. This flag can control search visibility (maybe hide operators until they are verified).

Document storage: Use Supabase Storage for file uploads. The `file_url` might be a public URL if you allow public access, but for sensitive documents, you probably want them **private** (only admin and the operator should see them). You can store just the path and use a signed URL when an authorized user wants to view/download it. - For example, `file_path = 'operators/1234/certificates/insurance123.pdf'`. - The application or a storage rule ensures only that operator’s members or admin can fetch it. - (OCR of these documents is not needed; just store the file and meta data.)

Admin verification workflow: - When an operator uploads a doc (in their dashboard, presumably an “upload documents” section), a row is inserted with status `pending`. We may also have an `operator_verifications` queue or just rely on this table. - An admin user would have an interface (or even directly use Supabase dashboard or a custom admin panel) to see all pending credentials. They review the file, ensure it’s legit and up-to-date. - Then admin sets `status='verified'` (and maybe fills verified_by, verified_at). If something is wrong (e.g. document illegible or not correct), admin can set `status='rejected'` and maybe add a note (“Document not clear, please upload a clearer copy”). - The operator could be notified (via email or notification) upon verification or rejection, and can upload a new one if rejected. - If using RLS, ensure that normal operators cannot themselves set their status to verified – only an admin can. Perhaps only allow them to insert with default pending and update only the file (if replacing). Admins (with a special role or `is_admin` flag) can update any field. We may identify admins either by a separate `admins` table or repurpose the old `profiles.is_admin` for platform staff.

Expiration tracking: - For each credential with an `expires_at`, consider a mechanism to handle expiry. Options: - A scheduled job (e.g. using Supabase scheduled function or external cron) that runs daily to find any credentials where `expires_at < now()` and `status = 'verified'`, then update status to `expired` (and perhaps notify the operator to upload a new one). - Alternatively, don’t auto-change status but when rendering the profile, if `expires_at` passed, treat it as not valid (e.g. show “(expired)” or don’t show the verified badge). - It’s safer to mark it expired in the DB so that any logic that checks “is this operator fully verified?” knows the credential is no longer valid. - You might keep the expired credential row for record, and maybe allow the operator to upload a new one (possibly a new row or update the existing row’s file and date, then mark pending again). - Possibly track historical credentials if needed (not necessary unless compliance requires history).

Other verification flags: - **Email/Phone verification:** Supabase Auth by default can verify emails. If you want to show “Email verified” or “Phone verified” on profiles, you can get that info from auth or store in profile. Phone verification if implemented would be another step (not sure if needed for operators, but could be). - **Profile completeness or quality tier:** Not exactly a credential, but you could have an internal scoring or a “Preferred Partner” designation if an operator meets certain criteria (like maintained high rating, low cancellation, etc.). This could be a flag/tier in `operators` (like `tier = gold/silver` etc.). That goes beyond initial data architecture into an incentive program, but keep it in mind.

RLS for credentials: Protect the `operator_credentials` table: - Operators (the owner or members) should be allowed to INSERT their own credential records (status defaulting to pending). They should **not** be allowed to mark it verified. Perhaps they can update their own record if they need to correct something (like replace a file if admin rejects it, setting status back to pending). Use WITH CHECK to restrict status updates by them: e.g. disallow them from setting status to verified themselves. - Admins should be able to SELECT all credentials and update status. So an admin role bypasses or has a separate policy. - Normal users or other operators should not see each other’s documents. So SELECT policy: `operator_id in (select operator_id from operator_members where profile_id = auth.uid)` OR if admin. - Possibly store the credentials in a separate secure schema or storage with strict rules. But if using the same public schema with RLS, above is fine.

Displaying on profile: Only show *verified* credentials to customers (with a checkmark). You could also list unverified ones as “provided, not verified” but that might undermine trust. Probably better to show only once verified, or use a badge style like gray icon vs green icon if you want to indicate submitted vs verified. For simplicity, show verified badges on the public profile. The operator themselves (when logged in viewing their profile dashboard) can see all their credentials and statuses.

Business information (admin-only): Some data might be required from operators for legal reasons: - e.g. **Business registration number** (like an ABN in Australia, or a company number). - **Tax ID** (if you handle payouts or invoices). - **Bank account for payouts** (likely stored separately, maybe in payments system, not relevant to public profile). These pieces of info should be collected but not displayed publicly. You can store them in `operators` (columns like `company_number`, `tax_id`) or in a separate table `operator_legal` if you prefer. Mark them as private in RLS (only admin can select). Since the prompt specifically mentions business registration possibly not public, make sure to collect it and mark as sensitive.

Summary of tables:

- **operator_credentials**: as above. Add a unique index on `(operator_id, type, region_id)` if you want to prevent duplicates of same type & region.
- Possibly a lookup for credential types (so you can display nice names).
- Could also have an `operator_verification_steps` table to track overall verification steps (like booleans for each required item). But that’s complicating; just checking the credentials table suffices.

By implementing this, you'll have a robust system where each operator's trust factors are stored and managed. On the profile, you might show a section "**Certifications & Credentials:**" with items like: - Licensed Tour Operator (if license verified) - Insured (maybe even show expiry: "Insured (valid through Dec 2024)") - National Park Permit: Kakadu (if region-specific) - First Aid Certified (if applicable) - and so on. Each with a green check if verified.

If something required is missing, you might omit it publicly or show a warning in their dashboard.

6. Statistics & Computed Fields

To enrich profiles, we track various performance statistics. Many of these can be *computed* from raw data (bookings, response timestamps, etc.), but computing on the fly for each profile view could be expensive. Instead, we can maintain certain stats in the database, updating them as events occur (using triggers or periodic jobs). Let's go through each requested stat:

- **Tours completed:** This likely means the number of tour **bookings** the operator has completed (i.e. successfully run). Since each booking corresponds to one scheduled tour instance (assuming no concept of group vs individual bookings difference here), this is essentially count of completed bookings for that operator. We can maintain a counter:
- Option 1: increment a counter on the operator each time a booking is marked as completed.
- Option 2: calculate via query by counting bookings where operator_id = X and status = completed.
- For performance and a quick display, we lean to storing it. For example, add `tours_completed_count` (integer) to `operators`. When a booking's status transitions to "completed" (from pending/ongoing), trigger function does `UPDATE operators SET tours_completed_count = tours_completed_count + 1 WHERE id = NEW.operator_id`. Similarly, if a booking is canceled or something after being counted, adjust (or you only increment once at completion and never decrement, which is okay unless a completion is undone).
- This is a relatively straightforward trigger on the bookings table. Ensure the trigger logic checks that the booking wasn't counted before (maybe only fire on status change event).
- Another interpretation: "tours completed" could mean how many distinct tour offerings they have completed at least once, but that's less likely. It's safer to assume total trips run.

- Display as "X tours completed".

- **Total guests served:** Sum of the number of guests across all bookings completed by the operator.

- Assuming your `bookings` table has a field like `guest_count` or `persons` indicating how many people were in that booking (for example, a booking for 3 people).
- If not, and each booking is per person, then "total guests" equals number of bookings. But likely a booking could cover a small group.
- Use a trigger on booking completion: `operators.total_guests = total_guests + NEW.guest_count`.
- If a booking is canceled or fails, don't count it. If a booking edit changes guest count before completion, handle appropriately (maybe only add when marking completed to avoid partial counts).

- This stat shows scale of their service. E.g. "256 guests served".

- **Years on platform:** This can be derived from the operator’s join date.

- Use `operators.created_at` as the date they joined the platform (or date they were approved).
- No need to store a number of years separately. On the frontend, you can compute `floor((today - created_at)/365 days)`.
- Alternatively, you could store a `member_since_year` or so just as reference, but it’s redundant.
- If you want to highlight it: display “On Webird since 2023” or “2 years on platform”. This is a static calculation at runtime.

- (If you wanted to surface it in queries or sort by tenure, you could index `created_at` or store an integer, but likely not necessary.)

- **Response time:** Average response time to customer inquiries or booking requests.

- This requires capturing timestamps of when an inquiry was made and when the operator responded. There are a few scenarios:
  1. **Booking requests:** If your system allows customers to send a booking request that the operator must accept/decline, measure the time between booking creation and the operator’s first response (accept or decline). You can have fields in bookings like `requested_at` and `responded_at`.
  2. Each time an operator responds (accepts the booking or messages), compute the diff. Perhaps store an `response_minutes` in the booking or compute on the fly.
  3. **Messaging inquiries:** If customers can message operators pre-booking (like “Contact host” messages), measure time from a customer message to the operator’s reply.
  4. You can combine these or decide one metric is primary. Likely, booking request response is key (similar to Airbnb’s response time metric).
- To store an average, you have to update a running total or average each time.
  - One approach: maintain `total_response_time` (in minutes) and `response_count` on the operator. When a new response occurs, do: `total_response_time += new_response_minutes, response_count += 1, average_response_time = total_response_time/response_count`.
  - Or recalc average each time without storing total (not great if many data points).
  - Response times can be skewed by some outliers, so sometimes median is used, but let’s assume average.
- If the platform requires near-real-time stat, updating on each response is fine. If not, you could periodically recompute last 30 days average etc. But given the low volume of inquiries likely, on-the-fly update is fine.
- **Where to capture:** If using a booking confirmation flow, you can place a trigger on booking status update from pending to accepted/declined. Compute the diff (accepted_at - created_at in minutes) and feed into the average.
- If using a separate messages table, then whenever a message from operator is posted replying to a new conversation, compute diff from conversation start.
-
- **Displaying:** Show perhaps “Response time: ~X hours”. Likely round to hours for display (or days if it’s long). You might store it as an integer of minutes for precision.

- If an operator has no data yet (no inquiries), you might not show it or mark “—”.

- **Booking confirmation rate:** The percentage of booking requests that the operator accepts (as opposed to declining or letting expire).

- If your booking system is instant confirmation, this metric isn't relevant. But if it's request-based, this is like "Acceptance rate".
- Compute as `accepted_requests / total_requests * 100%`.
- Implementation:
    - Maintain counters: `booking_requests_count` and `booking_accept_count` on `operators`.
    - On every new booking request, increment `booking_requests_count`.
    - On every accept, increment `booking_accept_count`.
    - Declines or expirations are implicitly counted in requests but not in accept count.
    - Then either store a precomputed `confirmation_rate` (as a percentage or fraction) or compute when displaying (accept_count / requests_count). Because it's just two integers, computing on the fly is trivial. You might choose not to store the percentage explicitly, just the counts.
- Alternatively, recalc after each decision: `confirmation_rate = booking_accept_count::float / booking_requests_count` in a trigger.
- If you allow instant book for some operators, those might be auto-accepted (in which case their rate is always 100% by definition).

- Show e.g. "Booking acceptance rate: 95%".

- **Repeat customer rate:** How many of an operator's customers return for more tours. There are a couple ways to define this:

- **Percentage of bookings that are by repeat customers** – (bookings by returning customers / total bookings). But one heavy repeater skews this.
- **Percentage of customers who have made more than one booking** – (number of unique customers with 2+ bookings / total unique customers). This tells, out of all customers who ever booked, what fraction came back for another booking.
- The second definition is more commonly understandable as "repeat customer rate". We'll use that.

- Implementation:

    - We need to track unique customers per operator and identify which are repeat.
    - You could maintain an `operator_customers` table: `operator_id, user_id, bookings_count`. Each time a booking completes, you:
    - check if an entry (operator_id, user_id) exists:
        - If not, insert it with bookings_count=1 (first time customer).
        - If yes, update bookings_count += 1.
    - Also, if you want to track the first booking date and last booking date, you could for extra info (not required for this stat).
    - Additionally maintain in `operators`:
    - `unique_customers_count` (total distinct customers served).
    - `repeat_customers_count` (count of customers with 2 or more bookings).

    - When you insert a new operator_customer entry (first time a customer comes): increment unique_customers_count by 1. (No change to repeat count.)
    - When you update an existing entry from 1 to 2, that user has become a repeat customer: increment repeat_customers_count by 1.
    - If a user goes to 3, 4, they were already counted as repeat, so no further changes.
    - Then repeat rate = `repeat_customers_count / unique_customers_count`. You can compute that on the fly for display or store it if you want.
    - Alternatively, you could calculate repeat rate at query time with a subquery count distinct vs count distinct where count>1, but maintaining it incrementally is efficient and keeps the profile load fast.
    - The `operator_customers` table (if used) also allows other analyses (like sending a thank-you to loyal customers, etc.)
    - If not using a separate table, you could attempt triggers on `bookings` to update the counts directly by querying distinct counts from bookings, but that's expensive and better done offline or with the separate table as a helper.
    - Considering scale, an operator likely won't have extremely high distinct customers (maybe hundreds or a few thousands at most), so computing distinct counts isn't terrible. But since we have triggers for other metrics, doing it incrementally is fine.

- Show something like "25% repeat customers" or "1 in 4 customers repeat". This indicates customer satisfaction/loyalty.

- **Other possible stats:** (not explicitly asked, but worth considering)

- **Cancellation rate** – how often the operator cancels bookings. (Important for trust.) You could track operator-initiated cancellations separately and show e.g. "Operator cancellation rate: 0%" if they never cancel. Could be derived from bookings statuses.
- **Avg group size** – could derive from total guests / total bookings.
- **Revenue** (if internal) – not for public display, but as an admin stat.

Stored vs computed trade-off: Each of these stats except "years on platform" benefits from being precomputed due to potentially complex queries: - If we tried to compute repeat rate on the fly: we'd do two COUNT(DISTINCT user_id) queries on bookings – okay for one operator, but if loading many operators in a list, that's heavy. - Counting total bookings and guests is simpler but still if done for every profile view, it's wasted work when it mostly changes only when a booking completes. - **Triggers**: Use Postgres triggers on the **bookings** table (and possibly on an inquiry/message table for response time) to update the `operators` table. - Make sure to handle edge cases: e.g. if a booking is deleted or its status is changed backward (maybe an admin toggles something), you might double-count or miss. You can add logic: only increment if transitioning from not-complete to complete, etc. If something is un-completed (unlikely scenario), you might leave the count or implement a way to recompute, but that's rare. - Alternatively, periodically reconcile counts with a scheduled job to fix any drift (just in case triggers missed something). But if triggers are correct, you'll be fine. - Triggers should be written as `AFTER UPDATE OF status` or `AFTER INSERT` on bookings, etc., and perform small updates or inserts. - Keep in mind triggers execute within transactions; heavy operations can slow down the original insertion. Our operations are small (incrementing a counter, etc.), so that's fine.

Performance: Updating a single row in `operators` for each booking completion is not a big deal (writes are not that high volume). If an operator has many simultaneous bookings finishing, those will serialize but that's manageable (and not common to complete at same time). Even if we have an operator with thousands of bookings, an integer increment is trivial. The `operators` table might become "hot" if many bookings across different operators complete at once (each update hits one row). However, those are different rows, so not the same row lock. If one operator somehow got a surge (e.g. a big batch import of completions), maybe slight contention on that row, but not likely.

Alternate approach: If we feared contention or bloat on operators due to frequent updates, we could have an `operator_stats` table, as mentioned, that stores these counters and update that instead. That way the main `operators` record (with description etc.) isn't constantly updated. Postgres MVCC means frequent updates can bloat. But unless bookings are extremely frequent, this is minor. For cleanliness, one might isolate highly volatile fields in a separate table [4]. For example:

```
operator_stats: operator_id PK, tours_completed, guests_served,
booking_requests, booking_accepts, unique_customers, repeat_customers,
total_response_time, response_count, etc.
```

Then join or view to combine when needed. This is conceptually nice (transient vs core data separation) [6], but introduces another join when loading profiles. It's a design choice: - Single table: simpler queries (one table to query for profile info) but more writes on that table. - Split table: fewer write conflicts on main table, but have to join to get stats [4].

Given moderate scale, keeping stats on `operators` is okay. We can always refactor if needed.

Indexing: If you want to filter or sort by these stats (e.g. find top-rated or most active operators), adding indexes on those fields can help: - e.g. `CREATE INDEX ON operators(avg_rating)`, or a composite like `(avg_rating DESC)` if doing sorted queries. But such queries are maybe rare (maybe an admin leaderboard). Sorting a couple hundred operators in memory might be fine. - For RLS or frequent use, indexes on foreign keys we already mentioned (bookings has operator_id indirectly via tour, etc.) - The `operator_customers` table (if used) should have PK on (operator_id, user_id). Index on operator_id (which is part of PK) helps listing all customers for an operator (for admin maybe). - If not using `operator_customers`, distinct count query on bookings table might need index on (operator_id, user_id) combination (through tour join). Better to maintain the table.

In summary, we will: - Add fields: `tours_completed_count`, `guests_served_count`, `unique_customers_count`, `repeat_customers_count`, `booking_requests_count`, `booking_accept_count`, `total_response_minutes`, `response_count`, and possibly computed `avg_response_minutes` (or compute on fly). - Use triggers on `bookings` for most: - On booking insertion (if status immediately means confirmed, count it; if pending, wait). - On booking status change to completed: increment tours_completed and guests_served; update operator_customers counts. - On booking creation (pending): increment requests_count. - On booking accept: increment accept_count and record response time. - On booking decline: just record response time (affects response_count if you measure any response as response, though accept rate goes down as not accepted). - If booking expires without response: that would not increment accept_count but would count in requests; response time could be considered infinite or you just exclude those from response calc. - For simplicity, measure response time for accepts/declines only. - Use triggers on a messages table or booking to capture response time if separate from accept (e.g. if operator responds in chat, though if booking is pending, presumably accept/decline is the main response needed).

Note: Some of these metrics like response time and accept rate might already be inherently provided by the booking workflow logic. You might implement them at the application level. But having them in DB ensures consistency and ease of querying for admin dashboards too.

Finally, we emphasize that one should not over-engineer prematurely – measure actual performance. If computing average rating on the fly by a SQL AVG() over maybe 50 reviews isn't slow, you might not need to store it [7]. But since the question explicitly asks, they likely want to store them. Caching these values is useful if you plan to sort or filter by them frequently (like showing a list of operators sorted by rating or with filters like "> 50 tours completed").

To be safe, we implement triggers for these for instantaneous UI updates and less complex client logic.

7. Multi-Region Considerations

If operators can run tours in multiple regions, we need to model their service areas and any region-specific data.

Modeling service areas: The typical approach is to have a reference table for regions and a many-to-many relationship: - **regions** table: This can include a hierarchy or types. For example: - `id`, `name`, `type` (perhaps type could be "country", "state", "park", etc if you want to differentiate levels), maybe `parent_id` if hierarchical (e.g. a state belongs to a country). - If your platform is primarily within one country, you might not need hierarchy beyond maybe region vs specific site. If it's global, you might include country and region subdivisions. - Example entries: (AU, country), (Queensland, state, parent=AU), (Kakadu National Park, park, parent=Northern Territory). - **operator_regions** (join table): `operator_id`, `region_id` as composite PK or with its own id. This lists all the regions the operator offers tours in. - If an operator has tours in multiple states or countries, list them all. - You might populate this automatically based on the locations of their tours (for instance, if they create a tour in Kenya, you add Kenya to their regions). However, allowing the operator to manually manage their service areas could be useful if they want to say "I can operate custom tours in X region on request" even if no tour listing exists yet. - Conversely, you could derive it: simply take distinct regions from their tours. That avoids duplication. But manual control might allow them to advertise coverage beyond listed tours or to exclude something. It depends on whether every region they serve is backed by at least one tour listing (likely yes). - As a compromise, maintain the table but update it via triggers on tours: when a new tour is added with location region Y, insert operator_regions(op_id, Y). Similarly, if a tour is removed, maybe keep the region anyway if they still serve it possibly. Or remove it if no tours left in that region. This is a bit complex to automate perfectly, so manual might be simpler: let operators pick regions in their profile settings (with admin verification if needed). - Indexes: index on region_id (to find all operators in a region, for search), and on operator_id (for joining to display regions on profile).

Region-specific credentials: As mentioned in section 5, some permits or licenses apply per region. For example, a national park permit is only for that park. We included `region_id` in `operator_credentials` to denote this. To tie it together: - Suppose region table has entries for each national park that requires a permit. We might also have a credential type for "park_permit". To enforce rules: - If operator adds region “Kakadu NP” in operator_regions, system should expect an `operator_credentials` entry with type “park_permit” and region “Kakadu NP” verified. - This can be handled administratively (the admin sees operator X added Kakadu but no permit credential and contacts them), or programmatically: - Possibly prevent the addition of that region unless they upload a doc. - Or allow it but flag as unverified (maybe not public). - Another approach: Instead of free picking regions, you could tie regions directly to tours (which you do) and simply not show tours in region X to customers unless a permit is verified. But that’s heavy-handed – better to verify first. - To implement a check: you could have a trigger on inserting into `operator_regions` that looks up if that region has a required credential type (you might have a small config table: e.g. `region_requirements(region_id, required_credential_type)`). If so, maybe set a flag or immediately mark something. This might be too deep; simpler to manage via admin workflow.

Tours and regions: Likely your `tours` table already has a location field (maybe a foreign key to a locations table or at least coordinates). If not, you should have something like `tours.region_id` or `tours.location_name`. For search, a region ID is useful (so a user can filter “show tours in Victoria”). Ensure consistency: if you have region associations at operator level, ideally each tour has an associated region that’s one of those the operator lists. This is more data integrity enforcement: - Could add a CHECK or just trust the operator not to mislabel. - If tours have precise locations (latitude/longitude), you could derive region from that via GIS queries, but probably overkill. Likely simpler: have them select a region for each tour from a predefined list.

Use cases: - A customer filters by region: you can either find tours in that region (tour.region_id = X) or find operators in that region (operator_regions contains X). For showing a list of operators by region, the join table helps. - An operator profile might list “Service areas: [Region A], [Region B], …”. - For multi-region operators, you might also want to show on each tour listing which region it is in (which is straightforward if tour has region_id).

Multi-country considerations: If an operator operates in multiple countries, you’ll have countries in the regions table. Possibly treat country as one region entry of type=country, and if they operate country-wide, just use that. Or they add multiple entries for states. It depends on how granular you expect the region specification: - Perhaps use whatever granularity the operator provides tours in. If tours are often tied to specific parks or locales, let them list those specifically. If they operate widely, they might just list the country or state level. - The design can accommodate various levels. The `regions` table can hold hierarchical entries, but you may not need to code that if not filtering at different levels.

Region-specific messaging: If credentials differ by region, ensure you communicate on the profile which areas are verified. Example: “Operates in: Kakadu National Park (permit verified), Arnhem Land (permit pending)”. This level of detail might be too much for customers; maybe simpler to only list regions once permits are verified, or list all and mark verified vs not. It depends on whether you allow tours to be booked in regions without verification (ideally no, from liability view).

RLS for regions: The `operator_regions` table: - Probably public (anyone can see which regions an operator covers, as it’s part of their public profile). - So SELECT for all roles (or a view). - Insert/Update should be allowed only to the operator’s members or admin. Possibly you let operators manage this list themselves in their dashboard (with maybe some UI hints about required permits). - Admin can also insert (maybe during onboarding if they want to pre-fill).

Example: An operator “Aussie Bird Safaris” runs tours in **Queensland** and **Northern Territory**. They have entries in operator_regions for QLD and NT. Additionally, within NT they specifically go to **Kakadu NP** which requires a permit. The regions table might have: - id=1, name=Queensland, type=state - id=2, name=Northern Territory, type=state - id=3, name=Kakadu National Park, type=park, parent_id=2 (NT) - id=4, name=Mamukala Wetlands, type=site, parent_id=3 (just an example of finer location if needed) The operator might list NT and QLD as service areas broadly. Or they might list Kakadu specifically as well. Up to how you design the UI (maybe multi-select of regions including parks). Then credentials table: - One entry: type=“insurance”, verified. - One entry: type=“tour_license”, verified. - One entry: type=“park_permit”, region_id=3 (Kakadu), verified. Now on profile: - “Service areas: Queensland, Northern Territory (Kakadu National Park)” - And a badge maybe: “Kakadu NP Permit ”.

You might not want to list every minor region they cover if they have many (e.g. an operator might just say “All of Victoria” rather than list each park).

In summary: **Use a join table for regions** to allow multiple regions per operator, and include region reference in credentials to tie specific permits.

8. Privacy & Visibility

Not all data is meant for all audiences. We must control who can see or edit certain fields.

Public vs authenticated vs admin-only data:

- **Public (unauthenticated) visible data:** The core operator profile info that any user browsing the site can see. This includes:
- Operator name, description, logo/photo.
- Verified badges (but *not* the underlying documents).
- Experience stats (tours completed, rating, years, etc).
- Tour listings (since those are publicly shown).
- Guide names and bios.
- Regions they operate in.
- Customer reviews and operator responses.
- Social media links or website if provided.
- Basically anything intended as marketing to potential customers should be public.
- **Authenticated-only:** You might decide some info is only for logged-in users (for example, some marketplaces hide reviews or contact button unless logged in). That’s a product decision. From a data standpoint, you could allow everything to public since there’s no obvious harm. Perhaps the only thing to restrict is direct contact or booking actions (which naturally require login). The profile content itself can be public to help SEO and discovery.
- If you did have something like a “Contact operator” form, initiating that would require login. But that’s handled in application logic more than data visibility.
- Some platforms also hide last names or exact locations until booking for privacy – e.g. showing only approximate area. If needed, you could store an exact address in private fields and a general

location in public fields. But for tour operators, exact addresses aren't usually shown anyway, just general locale.

- **Operator-only (self) data:** There's information the operator (and platform admins) can see, but not the public or other users. This includes:

- Their own contact info (email, phone) – though they know it, the platform might hold additional contact methods.
- Documents they uploaded (insurance PDFs, etc) – only the operator and admins should download those. Customers don't need to see the actual files; a checkmark is enough.
- Business legal info (tax IDs, bank details, etc) – only admin and the operator (maybe only admin in some cases).
- Internal metrics perhaps (maybe you show the operator their own acceptance rate, response time on their dashboard, which could be visible only to them and admin, not necessarily to customers. This is a question: some platforms display those metrics to customers as a selling point; others keep them internal or just for the operator to improve. The prompt suggests they want these stats presumably public as bragging rights or trust signals. But if, say, an operator has a poor 50% acceptance, would you show that to customers? Possibly as a warning. It might encourage them to improve. We'll assume they are public or at least to logged-in customers).
- **Platform messages** between customer and operator – only those parties can see their content (this would be a separate table with RLS isolating conversations).

- Any **admin notes** about the operator (for example, if you have an `operators.notes` for internal remarks, or a field like `is_flagged` for internal use) – admin-only.

- **Admin-only:** Platform administrators (the webird.ai staff) need access to everything to manage the platform:

- All operators' data, including private fields and documents.
- Admin should be able to see and edit verification statuses, perhaps edit operator profiles if needed (like correcting data).
- They might have fields like `is_approved` on operators – an operator might create a profile but an admin must approve it before it's publicly visible. That field would be admin-controlled. Until approved, you'd hide the profile from public (RLS could filter or a simple field check in queries).
- Possibly revenue or payout info if tracked.
- Admin-only UI could be Supabase studio or a custom admin panel or even flagged accounts that log into the app with an admin role (protected by RLS).

Visibility toggles and statuses: - Include a field like `operators.is_published` or `is_active`. If false, the profile should not show up to public. Use this during onboarding (so an operator can fill out profile, upload docs, but until an admin verifies critical docs and flips `is_published=true`, the profile stays hidden). RLS can enforce: `SELECT ... USING (is_published OR auth.role = 'admin' OR operator_members contains auth.uid)`. In other words, allow everyone to see it only if published, but still allow the operator themselves (auth.uid in members) or admin to see it even if not published (so they can preview their own profile or admin can review it). - Similarly, tours might have a published flag.

Protecting contact info: We want to avoid exposing operator's direct contact to avoid off-platform transactions and for privacy: - Do not store plaintext email or phone in any field that is publicly selectable. If you have `operators.contact_email`, restrict it. Possibly, don't even store it if it's the same as the user's email (since the profile's email is already in auth.users or profiles, which is protected). Operators likely share their account email as their business email. If they wanted a separate public contact email, that's not advisable from a business perspective. - For phone, maybe you collect it for emergencies but never show it. At most, once a booking is confirmed, you might share the phone number with the customer (for coordination). That can be done via the booking details (e.g. when booking is confirmed, display "Contact number: [phone]"). How to implement: either store `operators.phone` and then in the API only allow a user to select it if they have a confirmed booking with that operator. This could be done with a special function or a SELECT policy that joins bookings (complicated). Alternatively, simply have the operator manually share phone via the messaging after booking, or include it in a PDF itinerary. This might be beyond scope; but you could plan storing it privately anyway. - **Platform messaging:** Implement a table like `messages(thread_id, sender_id, receiver_id, content, timestamp, ... )`. Actually, for group threads it'd be different, but if just one-on-one user-operator, thread id could be booking id or a combination. The main point: those messages should be protected that only participants can read. RLS policy: `sender_id = auth.uid OR receiver_id = auth.uid`. Or if using a conversation table, ensure membership checks. This is separate from operator profile data, but worth noting since user asked if customers can message directly – the answer in Q&A was **platform-mediated messaging only**. So definitely do that.

RLS (Row-Level Security) policies considerations:

We will have RLS enabled on all tables (Supabase does by default when you turn it on). We need to craft policies to fit: - **profiles**: presumably already in place (each user can select their own, maybe others if some fields public, etc). - **operators**: - SELECT: - Allow all (including anon) to select *public fields* of published operators. E.g. policy: *"Public can read published operator profiles"* with `is_published = true` (and possibly `AND status='approved'` if using an approval status) [5]. - But we must be careful not to leak private columns. You have a few ways: 1. **Column-based RLS**: Supabase now supports Column Level Security in preview, but traditionally, one would create a Postgres VIEW to expose only allowed columns. Simpler: in your API, never query the sensitive columns for normal users. But that's client-side safety, not actual RLS. 2. You could create separate policies per column but that's cumbersome. Alternatively, split the table: - e.g. have an `operator_private` table with one-to-one relationship for things like contact, legal info. Restrict that table entirely to admin. - Or use the `public_metadata` / `private_metadata` JSON approach [5] – store user-visible profile info in `public_metadata` JSON and sensitive in `private_metadata`. Then a policy can be: allow all to select public_metadata (since it's just one column with safe content), and disallow selecting private_metadata unless admin. However, Supabase's default policies don't easily allow column-specific filtering beyond giving an error if unauthorized columns are queried. - A straightforward method: On the `operators` table, you can implement a policy for SELECT that returns row only if `is_published OR (auth.uid is a member of that operator) OR (auth.role = 'service_role')`. This means normal users get the row only if published. If not published, only the operator's own user or admin can get it. That covers row visibility. - But if the row is visible, all columns become visible by default (since it's the same row). To hide specific columns, you might rely on not querying them via your API. If someone tries a direct API call to fetch those columns, how to block? Postgres 15 has row filter per column features, but absent that, the best way is to isolate sensitive columns into another table or use a view approach. - Perhaps simpler: do not expose your Supabase API keys to end-users for direct queries, instead always use a controlled API that only returns safe fields. But that defeats some of Supabase's purpose. - Given the guidelines, maybe use separate tables approach for critical private data: - Ex: `operator_private(operator_id PK, contact_phone, contact_email, business_number, etc)`, RLS: only admin or operator's user can select their own row. Join with operators in admin dashboard as needed. - Or use `private_metadata JSON` on operators as Basejump did [5]. Then policy: `USING (true)` for all on operators (so row is visible) but they simply won't know what's in private_metadata unless they specifically query it. If they do, by default RLS doesn't block specific columns. So if the policy allows the row, all columns accessible. Actually, one can write a policy with SELECT using (some expression) and check (some expression) but it doesn't differentiate column. You can write a separate policy with `TO authenticated` for example that uses `SELECT (operator_id, name, slug, tagline, ... public fields) using (is_published)` – not trivial. Usually one would create a SECURITY DEFINER function to select only allowed columns. - An easier safe route: do as recommended – create a PostgREST view: e.g. `create view public.operator_public as select id, name, slug, tagline, description, logo_url, avg_rating, etc from operators where is_published=true;` Then you grant `SELECT on operator_public to anon, authenticated`. They can't access the underlying operators table directly (don't expose it in API). Meanwhile, `operators` table itself only accessible to admin and the operator for editing. This way, you completely control what public sees. This is a common pattern for exposing only certain fields. - Since the question is more about schema and architecture, you can mention using a **database view or a restricted API** to ensure only intended fields are visible publicly, as an aside. e.g. *"We may create a view or use Supabase's column security features to ensure private fields like contact info are never exposed to clients* [5] *."*

- INSERT/UPDATE/DELETE on operators:
  - Allow operators themselves to update their profile fields (name, description, etc) but not certain ones:
  - They should not update their own verification status fields or stats fields. We can enforce via triggers (just ignore if they try) or via RLS check constraints.
  - Example RLS check: `WITH CHECK ( new.avg_rating = old.avg_rating AND new.review_count = old.review_count AND new.is_verified_operator = old.is_verified_operator AND ... OR auth.role = 'admin')`. This ensures if a non-admin tries to change any of those protected fields, the update is rejected.
  - Alternatively, have triggers that on update just overwrite those fields back to old or recompute, effectively nullifying any unauthorized change. But better to block at RLS if possible so they get a permission error.
  - Only allow the specific operator's members to update that operator: `USING ( operator_id in (select operator_id from operator_members where profile_id = auth.uid and role in (owner,admin) ) )` for updates.
  - Admin can do anything: either via service role (bypasses RLS) or via a role check in policy (if you set up JWT with role claim).
  - Delete: probably no one except admin can delete an operator (and you might prefer a flag over deletion). Possibly not needed to expose delete at all to users – if an operator wants out, an admin can deactivate them.

- **operator_members**:

- SELECT: probably not public. You might not want to expose the raw membership list to everyone, especially since it may include non-guide roles. Instead, you'd expose guides via either a view or by filtering role=guide and joining with profiles to get public info. But some of that is done in backend. For simplicity, you might not expose this table via API except to the operator themselves and admin.
- For the operator (owner): allow them to see who their members are (and manage invites). For admin: full access.
- Policy: SELECT for profiles who are member of the same operator (so owners can see their guides). Insert: allow owner to add (invite) a member (likely done via invitation flow rather than direct insert). Delete: owner can remove a guide.

- Since MVP doesn't implement multi-member UI, this can be simplified: only admin inserts additional members for now, or no one does.

- **operator_credentials**:

- SELECT:
    - Operator themselves can see all their credential entries (so they know what's verified or pending).
    - Admin can see all.
    - No other users should have any access (we don't want other operators snooping on someone's docs).
    - We *don't* expose this table to public at all. Public only sees outcomes (badges).
- INSERT:
    - Operators can insert a new credential for their operator (with status default pending or a forced default).
    - They can update maybe to replace a file (we might allow update on the same row for say renewing an expired doc, or insert a new row – either approach). Possibly simpler: allow update of file_url by operator if status is pending or rejected (so they can fix it).
    - Admin can update status and any fields.

- Put a CHECK: if user is not admin, `new.status` must remain "pending" or "rejected"; they cannot set it to verified. Only admin policy allows setting verified.

- **reviews**:

- SELECT: allow all users (even anon) to read reviews (as they are shown on profile) – but maybe with a filter that the associated operator is published. If an operator is not published, probably their reviews shouldn't show either. Could join in policy, but easier: if operator not published, the operator itself won't show so reviews won't be fetched anyway in UI.
- INSERT: allow if booking.user_id = auth.uid and booking.status = completed (as discussed). This likely uses a function or a complex policy with subselect. This ensures verified reviewer.
- UPDATE: allow auth.uid = review.user_id to maybe edit (optional) – and ensure they don't change rating to something out of range etc. Or disallow editing altogether after submission to keep things simple.
- For response, allow if auth.uid is in operator_members for that operator. Ensure they can only set response_text fields: either by separate endpoint or by a policy check as discussed (old vs new values).

- DELETE: maybe allow users to delete their own review (or you might not allow deletion, only editing). Admin should be able to delete any (e.g. for policy violations).

- **operator_regions**:

- SELECT: This could be public (list of regions on profile), but you could also avoid exposing it raw by instead including regions in a view or via the API by joining. If you do expose it:
    - Policy: allow all to select (maybe with a join to published operators only).
    - Or just allow all, it's not sensitive that X operator is linked to Y region if operator is public.
- INSERT/DELETE: allow operator's owner/admin to manage their regions (and admin). Possibly with logic that if a certain region requires verification, either block or allow but mark something (for simplicity, we don't enforce at RLS level because that's complex).

- Could do a trigger on insert operator_regions that checks if region requires a permit and if operator doesn't have it verified, maybe log a warning or set a `needs_verification` flag somewhere. Or just trust admin to follow up.

- **tours**:

- Already likely has RLS: operators can CRUD their tours, public can select only published tours, etc.

- **profiles**:

- The guides info likely comes from profiles. You might have a policy allowing anyone to read basic public fields of profiles of guides. If your `profiles` table has mostly personal info not meant to be public, you might not want to expose it widely. Alternatively, you gather only minimal info for guides that is fine to show (name, maybe avatar). One way:
    - Create a view `guide_public_info` as `select p.id, p.full_name, p.avatar_url, p.bio from profiles p join operator_members m on p.id=m.profile_id where m.operator_id=…`. This might be too specific.
    - Or simply fetch via backend using an admin key and filter out sensitive fields before sending to client.
- If profiles includes personal data of normal birders (which it does), making profiles generally readable is not good (privacy of customers). But for guides who are essentially acting as professionals, showing their name and bio is intended.
- Perhaps mark in profiles if a user is acting as a guide (we have that via membership). You could decide that any profile that is a guide or operator, their name and bio can be shown on that operator's profile. That might be acceptable under terms since they are effectively providers who expect to be public.
- In RLS, you can't easily allow reading just those specific profiles and not others, except by a condition that ties to membership. Possibly: allow SELECT on profiles if `id IN (SELECT profile_id FROM operator_members where operator.is_published)`. That's doable:
    - e.g. `EXISTS(SELECT 1 FROM operator_members om JOIN operators op ON om.operator_id=op.id WHERE op.is_published AND profiles.id = om.profile_id)`.
    - That would allow reading profiles of any user who is a member of a published operator team (guides or owner). This covers guide info. It might accidentally also allow reading the profile

of an owner (which is fine if they are displayed as well). But what about that profile's other fields like their personal email? If your profile table contains email, you definitely don't want that exposed. Better to create a **profiles view** (like `public_profiles`) that exposes only name, avatar, bio, and possibly an ID. And restrict actual `profiles` table.
    - Many Supabase projects do this: store personal info in `auth.users` and just use `profiles` as a viewable table for basic info. You might have already something since they extended profiles with birding fields. Possibly that includes non-sensitive stuff only. Hard to guess. But likely, profiles table is semi-public (some fields).
    - At minimum, do not reveal contact info in it. Ensure not to select email/phone when showing guides. Usually, `profiles` might have `username/display_name` and maybe not email (since email in auth.users).
    - So maybe you're safe to allow reading name, bio, avatar of guides. But confirm schema or filter columns as needed.

In summary, implement **principle of least privilege** via RLS: - Public can only read what's intended: use published flags and perhaps views to limit columns. - Operators can only modify their stuff, not others'. - Admin can do everything (possibly using Supabase service role bypass or a role claim like `role=admin` on their JWT that RLS checks).

Privacy highlights: - Keep exact locations private: if you have coordinates or addresses for meetup, don't show publicly. Instead, show rough location or just region name. After booking, share exact meet point. That could be stored per booking or tour (like tour may have a fixed meeting point coordinates stored). - Payment info (if any in DB) absolutely confidential (likely handled by Stripe etc externally). - User data (like normal user reviews have names – you might show reviewer's first name and last initial for privacy, which you can derive from their profile name). - Possibly allow operators to have a setting to hide their personal last name if their business is just them. E.g. an operator who *is* the brand might not mind full name. But if someone calls themselves "Eagle Eye Tours" but the owner is John Doe, maybe the profile only shows "John D." as guide. However, since they are an operator, they might as well present full name for credibility. Up to them. Could mention pseudonym if needed.

Given the focus, likely the real concern was about contact info and documents.

Finally, mention that **performance** wise: - We should add appropriate indexes for any field used in RLS policies or joins to avoid slow checks (like membership checks – ensure operator_members has index on profile_id and operator_id so that subqueries in RLS are fast). - Also, using policies with subqueries (like verifying booking ownership in reviews) can have performance impact, but given moderate scale it should be fine. If needed, one can denormalize user_id into reviews as mentioned to simplify the policy to a simple equality check (which is faster for RLS to evaluate than a subquery join) [8]. That's one reason to store user_id in reviews, for instance. - RLS policies themselves should be tested for efficiency – e.g. doing `SELECT ... FROM operator_members` on each row access. But since each row check might index on profile_id, it should be okay if indexed. You can also use the trick of storing an array of member IDs in the operator row (like `member_ids uuid[]`) that gets updated via trigger. Then RLS can do `auth.uid = ANY(member_ids)` as a check quickly. That denormalizes membership into operator for fast check, at cost of complexity. Possibly overkill unless profiling shows a need.

This goes deep, but mention where relevant that RLS using subqueries is fine but keep them indexed and consider caching membership in a function or so if needed for performance (like Basejump does with a function `get_accounts_for_current_user()` that returns a set of account IDs cached per request [9] [10], to use in policies).

To wrap up: We deliver the recommendations with field types, relationships clearly, and have included rationale after each bullet group. We cited sources for the decisions where applicable: - separate table vs not [1] [2] - multi-member support concept [3] - showing how to separate public vs private (Basejump's metadata approach) [5] - storing stats in main vs separate table [4] - Only storing when needed vs on the fly [7].

These support our design reasoning.

---

[1] [2] Should I create separate tables for users and their profiles for a social network database? - Stack Overflow
https://stackoverflow.com/questions/5037834/should-i-create-separate-tables-for-users-and-their-profiles-for-a-social-networ

[3] [5] Accounts - Basejump
https://usebasejump.com/docs/accounts

[4] [6] [7] [8] database - Where to store average value in db? - Software Engineering Stack Exchange
https://softwareengineering.stackexchange.com/questions/291899/where-to-store-average-value-in-db

[9] [10] Open source Supabase SaaS starter - Basejump
https://usebasejump.com/