ChatGPT

Multi-Tenant SaaS Authentication & Authorization Plan

Overview & Architecture

We will implement a robust **multi-tenant authentication system** for the SaaS app using NextAuth.js (Next.js Auth). The system will enforce **row-level security (RLS)** in Supabase so each user only accesses their tenant's data. All application data tables already include a `merchant_id` (tenant identifier) column, which we will leverage for data isolation. We'll introduce a **centralized user/identity model** with role-based access control (RBAC) to distinguish Super Admins, Investors, Tenant Admins, and Tenant Users. Successful multi-tenant SaaS architectures use a similar approach – having separate **Users** and **Tenants** entities and linking them with membership/role records [1] [2]. This design aligns with industry best practices, allowing scalability to hundreds of tenants while keeping the schema simple [2].

Key goals of our plan:
- **Single Sign-On via NextAuth**: All user types (Super Admin, Investor, Tenant Admin, Tenant User) authenticate through NextAuth (using credentials login, since no public sign-up).
- **Role & Tenant Context in Session**: On login, determine the user's role and tenant associations via database lookup, then include this info in the NextAuth session/JWT for authorization checks.
- **Row-Level Security (RLS)**: Implement Postgres RLS policies on each multi-tenant table (`transactions`, `invoices`, etc.) to ensure a user can only read/write rows for their tenant. Super admins (and investors, with read-only rights) will be given overriding access to all tenants' data via role-based exceptions in the RLS rules.
- **Minimal Schema Changes**: Introduce only essential tables for user management (users, roles/memberships), optimizing for performance and future growth. We will reuse the existing `mx_merchant_configs` table as the Tenants table (each row is a tenant's API credentials and config).
- **Future-Proof Design**: Anticipate integration of additional data sources (GoHighLevel, Lobby EMR, QuickBooks in phase 2) by making the data ingestion **pluggable**. The architecture will allow multiple data sources per tenant without major schema changes, maintaining **tenant_id** as the primary partition key for all data.
- **High Performance & Scalability**: Ensure queries remain efficient as data grows to millions of rows by using indexes on `merchant_id` (already in place) and keeping RLS checks simple (direct `tenant_id` matches) [3] [4]. This single-database, multi-tenant approach with RLS is known to scale well into hundreds of tenants [5]. We'll also design with optional future strategies (like table partitioning or read replicas) in mind for further scale [6].

User Roles & Access Levels

We will define four user roles with distinct access scopes:

- **Super Admin** – *Global administrators* with full access to all tenant accounts and data. In the Super Admin dashboard, they can view the list of all tenants (from `mx_merchant_configs`) and impersonate or access any tenant's data. Super Admins can **create new tenants** (insert new config in `mx_merchant_configs` with credentials), **manage tenant status** (enable/disable access via `is_active` flag), and perform **data onboarding** for a tenant (trigger initial data fetch from MX Merchant API for a specified number of records). Essentially, Super Admins have **read-write access to everything** across all tenants. (Only Super Admins can create or delete tenant accounts and manage global settings.)

- **Investor** – *Read-only global observers*. Investors have access to all tenants' dashboards like a Super Admin, **but cannot modify any data or settings**. They can switch view into any tenant's data (based on permissions set by Super Admin) and browse all analytics, but **cannot add/edit** tenants, cannot trigger data fetches, and cannot change any records. We will implement this by giving investors the same tenant access as Super Admins (all tenants) but enforce **no write permissions** in both the app UI and the database policies. Super Admins can also constrain which pages or sections an Investor can see (e.g. perhaps only financial metrics but not configuration pages), providing a finer-grained access control at the application level.

- **Tenant Admin** – *Administrators of a single tenant (merchant account)*. Each tenant (merchant) will have one or more Tenant Admin users who can **access only their own tenant's data**. In their dedicated Tenant Admin dashboard, they can view and manage all data for their practice. Tenant Admins can trigger data sync for their account (e.g. "fetch the latest N transactions" from MX Merchant via their stored credentials – this will pull new invoices/payments and populate the database for that tenant). They can also manage user accounts **within their tenant**: e.g. add a new user (create a login that is associated with *their* tenant only) or remove an existing user of their tenant. Tenant Admins have full read-write permissions on their own tenant's data (they can update certain fields if the app allows, etc.), but have absolutely **no access to any other tenant's data**.

- **Tenant User** – *Limited user for a single tenant*. These are additional users that a Tenant Admin can create for their organization. A Tenant User logs in and sees **only the data for their tenant** (just like the Tenant Admin), but typically has more limited capabilities. For example, a Tenant User might only view dashboards and perhaps export data, but **cannot add other users or change tenant-wide settings**. Essentially they have read access (and possibly some limited write, if needed for specific features) to their own tenant's data. By default, we will treat Tenant Users as having **no administrative privileges**: they cannot initiate new data fetches from the external API (that will be restricted to Tenant Admin) and cannot manage users. They simply use the application's features (view reports, maybe update a record's status if the app allows, etc.) for their tenant.

Access Matrix: In summary, each API or page will be protected such that users only perform allowed actions: Super Admin [All Tenants: **Read/Write**]; Investor [All Tenants: **Read-Only**]; Tenant Admin [Own Tenant: **Read/Write**]; Tenant User [Own Tenant: **Read (mostly)**, minimal write]. These roles are mutually exclusive for a given login – a user account is either a Super Admin/Investor (global role) or belongs to a specific tenant (tenant role). This separation follows successful SaaS patterns where "tenants are business

units and users belong to them," and global admins are handled as a special category [7] [2]. We will ensure that a Super Admin account is **not tied to any single tenant** – they operate above the tenant layer. (If needed, we could technically allow a user to have multiple memberships – e.g. an external accountant could be linked to two tenants – but by design, our global roles cover multi-tenant access. Regular users will be associated with exactly one tenant in this implementation.)

Database Schema & Tables

To implement this multi-tenant user system, we will introduce a **User Identity schema** on top of the existing tenant data tables. The new tables will handle users, roles, and their relationship to tenants. We will optimize these tables for quick lookups and minimal bloat, following best practices for SaaS multi-tenancy [2] [1]. Below are the tables (with key columns) we plan to add, and how they integrate with existing tables:

- **Users Table** (`users`) – Stores all application users (of any role). Each user will have a unique ID and credentials for login. Key fields:
- `id` (UUID or bigserial): Primary key user ID. If we use Supabase Auth, this could align with `auth.users` ID; otherwise we generate our own UUIDs.
- `email` (text): User's login email (unique).
- `password_hash` (text): Hashed password for credentials (since we have no public signup, Super Admin or Tenant Admin will create users with a temp password which the user can later change). We'll use a strong hash (e.g. bcrypt).
- `name` (text): Optional, user's display name.
- `global_role` (text or enum): Role of the user if they have a global role. This field indicates if the user is a **"superadmin"**, **"investor"**, or **NULL/blank** if the user is tenant-scoped. For Super Admin/ Investor accounts, we will not associate them with a specific tenant in the membership table (they effectively have access to all).
- `created_at`, `updated_at` (timestamps).

Indexes: We'll add an index on `email` (unique index for fast lookup) and possibly on `global_role` if queries by role are common (not critical now). The `users` table is likely to remain small (a few hundred or thousand entries for our app), so performance impact is minimal.

- **Tenants Table** (`tenants`) – (Optional) This would store tenant organizations (e.g. practice name, etc.) and link to data. However, in our case the existing `mx_merchant_configs` already represents each tenant (with credentials and `merchant_id`). We will treat `mx_merchant_configs` **as the Tenants table** in terms of relationships: each row is one tenant account (medical practice) identified by `merchant_id`. We will likely extend it with a friendly name and any tenant-specific settings if needed. For example:
- *(If needed)* `name` (text): Name of the practice or tenant for display.

- *(Optional)* `owner_user_id` (UUID): If we want to designate a primary admin user who owns that tenant. We can derive this from the first Tenant Admin created, or store for reference.
  In the future, if adding non–MX Merchant data sources, we may add a generic `tenants` table with an internal `tenant_id` and have `mx_merchant_configs` reference it. But for now, **mx_merchant_configs is our authoritative tenant reference**, identified by `merchant_id` (the external ID). Its primary key `id` (UUID) can serve as an internal ID if needed.

- **User-Tenant Membership Table** (`user_tenants` or `roles`) – Maps users to the tenant(s) they belong to, and assigns their role in that tenant. This is critical for multi-tenant RBAC [7] [8]. Each entry represents a user's membership in a specific tenant:

- `user_id` (UUID): Foreign key to `users.id`.
- `merchant_id` (bigint): Foreign key to the tenant's `merchant_id` (or possibly to `mx_merchant_configs.id`). We will use the same `merchant_id` used in data tables to link membership.
- `role` (text or enum): Role of the user in that tenant. For now, likely `"admin"` or `"user"` (and we could also allow "read_only" etc. if needed). For example, a Tenant Admin user will have an entry with role "admin" for their tenant; a regular Tenant User will have role "user".
- Composite primary key on (`user_id`, `merchant_id`) to ensure a user cannot have duplicate memberships to the same tenant.
- We may also include fields like `invited_by` or `created_at` to track membership creation, but not strictly necessary.

Note: We won't create separate entries here for Super Admins or Investors for every tenant – that would be inefficient. Instead, global roles bypass the need for explicit tenant membership (their access is handled in RLS policies via the `global_role` field or JWT claims). However, if an **Investor** should only see a subset of tenants, we could either (a) give them membership entries only for those tenants, or (b) more simply, handle that in application logic (Super Admin toggles which tenants an Investor can access, then our UI/API filters accordingly). For now, we assume Investors can view all tenants by default (read-only).

Relationships & Foreign Keys:
- `user_tenants.user_id` → references `users.id` (on delete cascade, so if a user is removed, their memberships go).
- `user_tenants.merchant_id` → references `mx_merchant_configs.merchant_id` (each membership ties to a valid tenant). Alternatively, we could reference `mx_merchant_configs.id` (UUID PK) as the tenant foreign key if we prefer internal IDs. Using `merchant_id` (bigint) directly is fine since it's unique per tenant [3] and is used in our data tables.
- Data tables (`transactions`, `invoices`, etc.) already have `merchant_id` foreign key to `mx_merchant_configs.merchant_id` [9] [10]. This enforces that each transaction/invoice belongs to a valid tenant. We will build our RLS policies around these foreign keys.

SQL DDL Examples: Below are example SQL schemas for the new tables (to be run on Supabase/Postgres):

```
-- 1. Users table
CREATE TABLE public.users (
  id            UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
  email         TEXT NOT NULL UNIQUE,
  password_hash TEXT NOT NULL,
  name          TEXT,
  global_role   TEXT CHECK (global_role IN ('superadmin', 'investor'))  -- null
if tenant-level user
    DEFAULT NULL,
  created_at    TIMESTAMPTZ DEFAULT now(),
```

```
  updated_at    TIMESTAMPTZ DEFAULT now()
);

-- 2. User-Tenant Memberships table
CREATE TABLE public.user_tenants (
  user_id     UUID REFERENCES public.users(id) ON DELETE CASCADE,
  merchant_id BIGINT REFERENCES public.mx_merchant_configs(merchant_id) ON
DELETE CASCADE,
  role        TEXT NOT NULL CHECK (role IN ('admin','user')),  -- tenant-level
role
  created_at  TIMESTAMPTZ DEFAULT now(),
  PRIMARY KEY (user_id, merchant_id)
);
-- (We can also add an index on merchant_id alone if we often query all users of
a tenant)
CREATE INDEX idx_user_tenants_merchant ON public.user_tenants(merchant_id);
```

(The above uses a text-based role; alternatively, we could normalize roles into a separate table and use a role_id, but given just two values ('admin','user'), a CHECK constraint is sufficient.)

We will also possibly **extend** `mx_merchant_configs`: for usability, add a `tenant_name` column if one doesn't exist, so Super Admin can label tenants (instead of only seeing an ID). For example:

```
ALTER TABLE public.mx_merchant_configs
  ADD COLUMN tenant_name TEXT;
```

This lets us store the practice or business name for display in the Super Admin dashboard. We'll update this when creating a new tenant.

With these tables in place, our data model supports multi-tenant relationships: A user can belong to multiple tenants (e.g., if we had a consultant with access to two merchants, they'd have two rows in `user_tenants`), and each tenant can have multiple users. This approach is consistent with modern SaaS RBAC designs where users have roles scoped per tenant [8].

Authentication Flow with NextAuth

We will use **NextAuth.js** to handle user login sessions. The plan is to use NextAuth's **Credentials Provider** (since we only have email/password logins, no social providers at the moment) to authenticate against our `users` table. Here's how it will work:

1. **Login Page**: The app will present a login form (email and password). We won't have self-service signup (new users are only created by admins). The form sends credentials to NextAuth.

2. **NextAuth Credentials Provider**: In our NextAuth `[...nextauth].ts` config, we'll define a credentials provider that verifies the email/password. This verification logic will:

3. Look up the user by email in the `users` table (using Supabase or a Prisma client, or Supabase's JS client with admin access).
4. If found, compare the provided password with `password_hash` (using bcrypt verify).
5. If match, authentication succeeds. We then retrieve the user's roles: specifically, their `global_role` and any tenant memberships. We'll load `user.global_role` and perhaps the list of `merchant_id` from `user_tenants` for that user.
6. If not found or password mismatch, authentication fails.
7. **Session Payload**: Upon successful login, NextAuth will create a session (JWT or secure cookie). We will include custom claims in the session token to carry authorization info. Specifically, we'll include:
8. `uid` – the user's UUID (this will serve as their identifier in Supabase RLS; we will set this as the JWT subject claim `sub`). Supabase's `auth.uid()` function reads the JWT's `sub` claim to identify the user [11], so it's crucial we set `sub = user.id` in the token.
9. `role` – the user's role. If the user has a `global_role` of "superadmin" or "investor", we put that. If not, we can default to "tenant_user" or "tenant_admin" (we might infer tenant admin if they have a membership role "admin"). We might also include a list or single value of their tenant ID for convenience.
10. Possibly `tenant_id` (or `merchant_id`) – for users who belong to a single tenant, we could include that tenant's ID in the token. However, a user might have multiple (though in our plan, that's only true for global roles, who have all anyway). For simplicity, we might include the tenant context in the session after login or let the user choose if they have more than one. Since our normal users have exactly one tenant, we can safely include `merchant_id` in their JWT claims. For Super Admin/Investor, we might set `merchant_id = "*"`, or just omit it since they have global access.

Using NextAuth's callbacks (e.g. `jwt` and `session` callbacks), we'll attach this data. That way, on the client we know the user's role, and when making Supabase queries we can use the JWT for RLS.

1. **Supabase Integration**: We have two ways to query Supabase data:
2. **Option A:** Use the Supabase JavaScript client in the front-end, authenticated with the JWT from NextAuth. We can configure the Supabase client with the JWT by calling `supabase.auth.setAuth(token)` (or directly initializing with the token). Supabase can be configured to trust external JWTs by using the same signing secret (the Supabase JWT secret) for our NextAuth JWT. If we sign our NextAuth JWT with Supabase's secret, Supabase will treat our token like its own [12]. This allows `auth.uid()` and `auth.jwt()` in RLS to work with NextAuth tokens. In practice: after login, we obtain the JWT from NextAuth (which we sign with Supabase's secret) and pass it to the Supabase client; all Supabase queries from the browser will include this JWT for RLS.
3. **Option B:** Alternatively, perform all data fetching on the Next.js server side (using Supabase's Service Role key which bypasses RLS) and only return allowed data to the client. However, this is less efficient and bypasses Supabase's real-time and RLS features. The preferred approach is **Option A: direct Supabase queries from the client with RLS enforcement**, which is exactly what Supabase is designed for.

We will implement **Option A**. This means configuring NextAuth's JWT to use the Supabase secret (`NEXTAUTH_SECRET` should match Supabase JWT secret or we use Supabase's secret to sign a custom token) [12]. By doing so, each NextAuth session token can directly be used as a Supabase auth token. We must ensure the JWT contains the standard claims that Supabase expects: - `sub`: user's UUID (for

`auth.uid()`). - `role`: typically `"authenticated"` by default for logged-in users; we can also include our custom `role` claim (e.g., `role: superadmin` or add a separate claim like `is_superadmin: true`). Supabase's default RLS can check the built-in `auth.role()` which returns `"authenticated"` or `"anon"` depending on token, but we will define our own policies using either custom claims or by checking against our membership table. - Any other custom claims we want (like `tenant_id` if needed).

Industry Insight: Many successful multi-tenant apps using Supabase integrate custom auth by either syncing users into Supabase Auth or by minting custom JWTs with the Supabase secret [12]. We'll follow this latter approach – NextAuth handles the login UI and session management, and we generate a Supabase-compatible JWT behind the scenes. This gives us the best of both: NextAuth's flexibility and Supabase's secure RLS enforcement.

1. **Role Determination Logic**: On login, how do we decide if a user is Tenant Admin vs Tenant User? We will check the `user_tenants.role` for that user's tenant. If it's "admin", we know they're a Tenant Admin; if "user", they're a regular Tenant User. We might include that info in the JWT (e.g. a claim `tenant_role: "admin"`). For Super Admin or Investor, `user.global_role` is set, so we mark them accordingly (and they won't have a tenant-specific role entry). In the NextAuth session, we could set `session.user.role = "superadmin"` (or "investor", "tenant_admin", etc.), and maybe `session.user.merchant_id` for tenant users.

2. **Post-Login Tenant Context**: If a user has access to multiple tenants (which in our design only global roles do), we may provide a way in the UI to **switch tenant context** for viewing data. For instance, a Super Admin dashboard listing tenants, and on clicking one, we set a context (perhaps a cookie or local state) that tells the front-end which `merchant_id` to filter on. The JWT still grants them all data, but the app will typically query one tenant at a time for manageability. (Alternatively, we could issue a new JWT when a Super Admin "impersonates" a tenant, but that's not necessary – we can just apply a filter on queries). For Investors, similarly, we'll allow selection of tenant to view (with read-only UI).

3. **No Public Signup**: Since only Super Admin can create new tenants and Tenant Admins can invite/create users for their tenant, we do not expose any registration flow. All accounts are provisioned internally. This simplifies NextAuth usage (only login needs to be handled). We will however implement secure flows for **password management** (Super Admin sets an initial password which users can change; or an invite email with a secure link for the new user to set their password).

4. **Multiple Super Admins**: The system allows multiple super admins. To add a new Super Admin, an existing Super Admin could have a UI form to create a user with global_role = "superadmin". That user then can log in and will have full rights. Similarly for adding an Investor (global_role = "investor"). We might manage Super Admin creation either directly in the database (initial seeding) or via an admin interface protected to existing superadmins.

Security Considerations:
- We will enforce strong hashing for passwords and possibly 2FA if needed in future. NextAuth can integrate with 2FA or one-time codes if required.
- **Session security**: NextAuth's session cookies/JWT will be configured as HttpOnly and secure. The JWT expiration will be moderate (e.g. 1 hour) with automatic refresh to balance security and UX.
- **NextAuth Adapter**: (Note for completeness) We could use the official Supabase Adapter for NextAuth,

which would store NextAuth's own tables (`accounts`, `sessions`, etc.) in our database [13] [14]. However, that adapter still assumes using Supabase as a traditional OAuth provider or storing sessions separately; it doesn't leverage Supabase RLS directly. Our approach keeps things straightforward: use NextAuth purely for session management, and directly leverage our tables for credentials. No additional NextAuth-specific tables are needed unless we add OAuth providers later.

By following this approach, **each user's identity is clearly mapped to their tenant access**. The NextAuth login will identify "who you are and what you can access," and Supabase's RLS will double-enforce that "you can only query what you're allowed to." This layered approach (JWT claims + RLS policies) is a robust industry practice for multi-tenant security [15] [16].

Row-Level Security (RLS) Policies in Supabase

We will enable RLS on all relevant tables to enforce tenant isolation and role-based permissions at the database level [17] [18]. With RLS, even if a faulty query or malicious attempt is made, the database will **automatically filter or reject unauthorized access**. Here's the plan for RLS policies on key tables:

1. Enable RLS: First, ensure RLS is turned on for each table:

```
ALTER TABLE public.transactions ENABLE ROW LEVEL SECURITY;
ALTER TABLE public.invoices ENABLE ROW LEVEL SECURITY;
ALTER TABLE public.product_categories ENABLE ROW LEVEL SECURITY;
ALTER TABLE public.mx_merchant_configs ENABLE ROW LEVEL SECURITY;
ALTER TABLE public.user_tenants ENABLE ROW LEVEL SECURITY;
ALTER TABLE public.users ENABLE ROW LEVEL SECURITY;
```

By default, once RLS is enabled, no rows can be accessed until policies are added.

2. Define Helper Functions/Claims: Supabase provides `auth.uid()` to get the user's ID from the JWT and `auth.role()` for the Supabase auth role. Since we are using custom JWT, `auth.uid()` will still work (if `sub` is set) [11]. We can also access custom JWT claims via `current_setting('request.jwt.claims')` or define a function. For simplicity, we might not need a custom function – we can do subqueries against our membership table using `auth.uid()`. However, we might define a convenience function to check membership, e.g.:

```
CREATE FUNCTION public.user_is_member_of_tenant(p_user_id UUID, p_merchant_id
BIGINT)
RETURNS BOOLEAN AS $$
  SELECT EXISTS (
    SELECT 1 FROM public.user_tenants
    WHERE user_id = p_user_id AND merchant_id = p_merchant_id
  );
$$ LANGUAGE SQL STABLE;
```

This function returns true if the given user is a member of the given tenant. We'll use it in policies. (We will also consider Super Admin/Investor bypass in the logic below.)

3. Policies for Data Tables (transactions, invoices, etc.):
These tables all have a `merchant_id` column. We want a policy like: *"User can SELECT a row if they belong to that row's tenant OR they are a Super Admin or Investor."* Similarly for INSERT/UPDATE: *"User can modify a row if they belong to that tenant and have appropriate role (Tenant Admin or Super Admin)."* We'll create separate policies per operation for clarity.

For **SELECT** (read) access on a table (e.g. `transactions`):

```
CREATE POLICY "tenant_select" ON public.transactions
FOR SELECT
USING (
  -- Allow if user is member of this row's tenant
  user_is_member_of_tenant(auth.uid(), merchant_id)
  -- OR allow if user is global superadmin or investor
  OR auth.jwt() ->> 'role' IN ('superadmin', 'investor')
);
```

Explanation: `auth.jwt() ->> 'role'` pulls the custom role claim from JWT (assuming we put it there). If the role claim is "superadmin" or "investor", we allow access to all rows [8]. Otherwise, we require that `auth.uid()` (the user's ID) has a membership entry with `merchant_id` matching the row's `merchant_id`. The `user_is_member_of_tenant` function does that check (alternatively, we could inline the EXISTS subquery). This ensures tenant users only see their tenant's data [2], while superadmins/investors see all.

For **INSERT/UPDATE/DELETE** on data tables: we'll be stricter:
- **Inserts**: Generally, app users won't insert transactions or invoices directly – those come from webhooks or admin actions. But if needed (say a tenant admin manually adds a transaction record), we allow it only for those who belong to that tenant *and* likely only if they're tenant admin or superadmin. We can check the user's membership role. One way: include the membership role in JWT as well (like `tenant_role`). Alternatively, join to `user_tenants` inside the policy. For simplicity, we might trust application logic for now (only Tenant Admin UI will call insert APIs), and allow any member for insert (or restrict to admin via a check). Example policy:

```
CREATE POLICY "tenant_insert" ON public.transactions
FOR INSERT
WITH CHECK (
  user_is_member_of_tenant(auth.uid(), NEW.merchant_id)
  OR auth.jwt() ->> 'role' = 'superadmin'
);
```

The `WITH CHECK` ensures any new row's `merchant_id` must be one that the user is a member of, unless they're superadmin. (We exclude investor because they shouldn't insert at all; but an investor wouldn't call an insert anyway due to app logic, and we could also exclude them by not granting insert rights to the investor role at the DB level.)

- **Updates/Deletes**: Similar to insert. Tenant Admins (membership role "admin") can update data for their tenant (for instance, maybe update an invoice's status or mark something as reviewed). Tenant Users might have read-only rights, so we likely restrict updates to admins. We cannot easily know from JWT if a user is tenant admin unless we put that in claims. It might be easier to check via a join: does `user_tenants` have role='admin' for this user and merchant? We can do:

```
CREATE POLICY "tenant_update" ON public.transactions
FOR UPDATE
USING ( user_is_member_of_tenant(auth.uid(), merchant_id) OR auth.jwt() -
>> 'role' = 'superadmin' )
WITH CHECK (
  -- Only allow update if user remains within their tenant
  user_is_member_of_tenant(auth.uid(), merchant_id)
  OR auth.jwt() ->> 'role' = 'superadmin'
);
```

  And then optionally add a check on membership role if needed (maybe a separate policy for admin vs user). Another approach is to embed role in JWT. For instance, for a tenant admin, we could include a claim like `tenant_role: "admin"`. Then the policy could be: `OR (auth.jwt() ->> 'tenant_role' = 'admin' AND user_is_member_of_tenant(auth.uid(), merchant_id))` for update. This would let tenant admins write, but not tenant users. We will implement such nuance as needed.

- We will also add **SELECT policies on** `mx_merchant_configs` so that:

- Super Admin can select all (to list tenants).
- Investors can select all (if we allow them to see tenant list, likely yes read-only).
- Tenant Admins/Users should probably **not** select this table at all (they don't need to see API credentials of their tenant). We might entirely disallow tenant-level users from selecting `mx_merchant_configs` (or only allow a safe subset if needed). Possibly we create a limited view if needed. For now, we'll say no SELECT on this for normal tenants (or only permit if `merchant_id` is theirs but excluding sensitive fields).

- **Insert** on `mx_merchant_configs` (to add new tenant) – allow only Super Admin (maybe via a policy checking `auth.jwt() ->> 'role' = 'superadmin'`). Similar for update/delete (only superadmin).

- **Policies on** `user_tenants` **(membership table)**:

- *Select:* Super Admin can see all (to manage any tenant's users). Tenant Admin should be able to see memberships of their tenant (so they can list their users). So a SELECT policy: allow if user is

superadmin **or** the membership's `merchant_id` is one that the user is an admin of. We might need to join back to check the role. Alternatively, we might treat that Tenant Users (non-admin) don't need to read this table at all. So:

```
CREATE POLICY "read_memberships" ON public.user_tenants
FOR SELECT
USING (
  auth.jwt() ->> 'role' = 'superadmin'
  OR (
    user_is_member_of_tenant(auth.uid(), merchant_id)
    AND ( -- check that the requesting user is an admin in that tenant
         EXISTS (
           SELECT 1 FROM public.user_tenants ut2
           WHERE ut2.user_id = auth.uid()
             AND ut2.merchant_id = user_tenants.merchant_id
             AND ut2.role = 'admin'
         )
      )
  )
);
```

This means: superadmin sees all memberships; otherwise, a user can see membership rows for their tenant only if they themselves are an admin of that tenant. That allows Tenant Admin to see their users; Tenant regular users won't satisfy the `role='admin'` check, so they won't be able to list users (which is fine).

- *Insert:* Adding a membership (which effectively is what happens when a Tenant Admin creates a new user for their tenant, or Super Admin assigns a user to a tenant). We allow Insert if:
  - Superadmin (they can add any user to any tenant, e.g., when creating a tenant and its first admin).
  - Tenant Admin adding a user to **their** tenant. So:

    ```
    CREATE POLICY "add_membership" ON public.user_tenants
    FOR INSERT
    WITH CHECK (
      auth.jwt() ->> 'role' = 'superadmin'
      OR (
        NEW.merchant_id = ANY(SELECT merchant_id FROM
    public.user_tenants ut2
                              WHERE ut2.user_id = auth.uid() AND
    ut2.role = 'admin')
        -- ^ ensures the actor is admin of the merchant in the new row
      )
    );
    ```

    This uses a subselect to ensure the `NEW.merchant_id` is one where the inserting user is an admin. Thus, a Tenant Admin can only add users to their own tenant.

- *Delete:* Removing a membership (e.g., deleting a user or revoking their access). Only Super Admin or the Tenant Admin of that tenant should do this. We'll write a similar policy using a check like above.

- *Update:* Likely not needed much (maybe to change a user's role within a tenant, e.g., upgrade a user to admin). We can allow if the user is superadmin or tenant admin (and probably disallow tenant admin from changing someone else to admin unless we want to allow that – that's a business decision). Possibly, only superadmin can promote roles to avoid privilege escalation issues.

- **Policies on `users` table**:

- Regular tenant users probably shouldn't query the `users` table directly at all. They only need to get user info via safer routes. We can restrict `users` table heavily.
- Super Admin may need to select from `users` (to view all users or manage global accounts).
- Tenant Admin might not need direct access to `users` table if they can get their tenant's user list via the membership join (which can include user details through a join or a view). However, for convenience, we might let tenant admin select basic fields of users that are in their tenant. We could do this via a complex policy or simply avoid it by providing a server API that fetches the joined data.
- For now, likely: **No direct SELECT on users for tenant roles** (we handle in backend if needed), and allow Super Admin to select all. If needed, we create a secure view that joins `user_tenants` with `users` to get users per tenant and apply RLS on that view.
- Insert into `users`: Who can create a new user? Super Admin (for global roles or possibly for any tenant) and Tenant Admin (for their tenant's users). We will allow insert on `users` if the inserting user is superadmin OR (if we want Tenant Admins to create users, we allow them to insert but then we must immediately also insert a membership linking that new user to their tenant). This likely will be done via a stored procedure or transaction in the backend – when a Tenant Admin creates a user, our API route will create the user row and the membership row together. It might be safer to do user creation via a privileged service (to handle hashing password, etc.) rather than letting the client directly insert into `users`.
- So perhaps we won't open up `users` table inserts via RLS at all; instead, we'll use a custom Supabase Function (RPC) or Next.js API to handle user creation. That function can enforce proper tenant assignment and hashing. This avoids exposing raw password hashes via client.

In summary, our RLS policies enforce that:
– *Tenant-scoped tables:* Only rows with `merchant_id` matching a tenant the user is associated with are visible [2]. Users cannot access other tenants' rows. Super Admin/Investor roles bypass the tenant check (all rows visible) [8].
– *Write operations:* Only allowed within one's tenant, and often only for admin roles. No cross-tenant writes. No writes for Investor role.
– *Administrative tables:* Only Super Admin can view/modify `mx_merchant_configs` (tenants) and perhaps `users`. Tenant Admins have limited visibility (e.g., can list their own users via membership table, but not touch other tenants).
– *Security fallback:* If somehow a bug in the front-end allowed a request for data from another tenant, the RLS policy will return no rows or deny the operation, protecting the data by default [19] [18]. This "defense in depth" is a big win for using RLS in multi-tenant SaaS.

We'll test these policies thoroughly. For example, when a user logs in and queries `transactions`, the SQL executed might implicitly include `WHERE merchant_id = <their tenant>` due to the RLS using the

membership check. Supabase's query planner will use the index on `merchant_id` [20], so performance remains high even with RLS.

Future Data Source Integration

To accommodate **phase 2 requirements** of integrating other data sources (GoHighLevel, Lobby EMR, QuickBooks, etc.), our design will remain **extensible**:

- We treat `merchant_id` as a universal tenant key across all data. For new data sources that don't inherently have the same ID, we will map them to the tenant's `merchant_id` or internal tenant ID. For example, if QuickBooks integration is added, we might have a `quickbooks_config` table with its own credentials, linked to the same `mx_merchant_configs.id` or `merchant_id`. This way, each tenant can have multiple data source credentials, but all data tables still carry the *same tenant identifier*. This makes enforcement of RLS and querying easier – no matter the source, any record can be tied to `merchant_id` (or a tenant UUID).

- **Pluggable Ingestion**: We will design the data import logic in a modular way. Currently, we fetch transactions from MX Merchant. In future, we'll create similar pipelines for other sources. Possibly we introduce a column in `transactions` (or separate tables) to indicate `source` (e.g., `source = 'mx' or 'quickbooks'`). If data schemas differ greatly, we might keep separate tables per source (e.g., `qb_transactions`), but ideally we normalize as much as possible so the dashboard can show combined info. Regardless, **the RLS policies will be similar**: include tenant id on every new table and restrict by it. We can also add a higher-level `data_source` table and link credentials to tenants through that.

- **Prototype for Multi-Source**: Initially, our focus is MX Merchant. But we will keep the architecture open. For example, adding a `tenant_id` primary key for each tenant (uuid) could be wise. Then `mx_merchant_configs` and future `qb_configs` can both link to `tenant_id`. If we foresee the same tenant needing mapping across systems, an internal `tenant_id` is useful. We might retrofit that: e.g., add `tenant_uuid` to `mx_merchant_configs` and also use it as foreign key in `user_tenants`. This way, the tenant is not tied to the MX `merchant_id` alone. This is a design choice: to keep things simple, we may stick with `merchant_id` now and later migrate to a unified tenant id.

- The **dashboard UI** might allow toggling data sources (say viewing combined metrics or switching context between MX Merchant data vs. QuickBooks data). Because we plan for similar multi-tenant isolation for each source, adding new sources won't break the security model. We just replicate the pattern: each new dataset has a tenant key and RLS by that key.

In short, the system is prepared to "plug in" new sources by adding new config tables and data tables, all keyed by tenant. The Super Admin interface can manage credentials for new sources per tenant. We will also incorporate any differences in how we fetch data (e.g., QuickBooks might not have a numeric merchant_id – we might store its OAuth tokens keyed by tenant). Our architecture's modular nature ensures this is manageable.

Security & Performance Considerations

Row-Level Security Efficiency: Postgres RLS policies are checked for each row access, but when written with simple conditions on indexed columns, they are very efficient [20] [16]. Our policies primarily do a subquery on `user_tenants` by `merchant_id`, which is indexed. For example, the subquery `EXISTS(SELECT 1 FROM user_tenants WHERE user_id=X AND merchant_id=Y)` is a quick index lookup. We have added indexes on `user_tenants(merchant_id)` and could add on `(user_id)` too for completeness. With 600+ rows per table per tenant (as mentioned ~600 rows/table for each tenant initially), even at scale of 500 tenants (300k rows), these lookups remain extremely fast (a index query on a few hundred membership rows, which is negligible).

Scalability: This single-database multi-tenant design can handle **millions of rows** [21] [22]. Supabase (Postgres) can scale vertically to handle more load and supports partitioning if needed. If we reach tens of millions of transactions, we might consider *table partitioning by merchant_id* to keep performance optimal – for instance, one partition per merchant or per few merchants, which was suggested as a future scaling strategy [6]. However, premature partitioning isn't needed until we see performance limits. For now, proper indexing and query patterns suffice.

Supabase also offers features like read-replicas for heavy read traffic, which we could utilize for investor or analytics queries that span tenants.

Evaluation of Architecture:
- *Architecture Soundness Score:* **9/10** – The proposed architecture closely follows proven multi-tenant SaaS patterns [7] [8]. It cleanly separates concerns: authentication via NextAuth, authorization via RLS, and structured role hierarchy. By not reinventing identity management (leveraging stable libraries and DB constraints) and by using row-level security at the database level, we reduce complexity in the application layer [23]. The only reason it's not 10/10 is that managing custom JWT integration with Supabase adds a bit of complexity, and we must carefully coordinate NextAuth and Supabase secrets. But overall, it's a robust, maintainable design. It avoids unnecessary tables (we add only what's needed for users/roles) and sticks to **"secure by default"** principles (no data leaks across tenants). This architecture is highly aligned with what successful multi-tenant products implement [5] [8].

- *Scalability Score:* **8.5/10** – This design will scale well for the foreseeable future (hundreds of tenants, millions of rows). Using a single database for all tenants with indexed tenant keys is the most straightforward scaling model and is known to handle a large number of tenants efficiently (e.g., tens of thousands of tenants) as long as queries are indexed by tenant. We've optimized critical queries with indexes on `merchant_id` and composite indexes where needed [3] [20]. The use of RLS does add a slight constant overhead to each query, but in practice this is negligible with proper indexing and has been used in large-scale Supabase apps. As we integrate more data sources, the load will increase, but the modular nature means we can scale horizontally (by splitting read load, caching, etc.) if needed. Eventually, if one database becomes a bottleneck, a higher complexity approach (like sharding by tenant or separate DB per tenant) could be explored [24], but that is unlikely necessary until extremely high scale. For now, our approach balances simplicity and performance, and Supabase itself is built to scale to millions of rows easily on a single instance. Real-time features and the use of Redis for webhooks also alleviate pressure on the DB for writes [25] [26]. Overall, the system should comfortably support the expected growth in tenants and data volume with minor tuning.

Implementation Plan (Step-by-Step)

Finally, we outline a concrete plan (as a **plan.md** for development) that details the implementation tasks. This plan can guide developers or AI agents to build the features as specified:

1. **Database Migrations**:
2. Create the `users` table for storing user credentials and roles [1]. Ensure to hash passwords and not store any plaintext.
3. Create the `user_tenants` (membership/roles) table linking `users` and `mx_merchant_configs` (tenants) [2]. Add appropriate foreign keys and indexes.
4. Add a `tenant_name` column to `mx_merchant_configs` for readability (optional, for Super Admin UI).

5. (Optional) Insert initial data: e.g., create a Super Admin user in the `users` table with a known email/password, and insert a `global_role='superadmin'`. Also create any test tenants in `mx_merchant_configs` if needed for dev.

6. **Supabase Configuration**:

7. In Supabase settings, add our NextAuth JWT signing key as an accepted JWT secret if possible (or use the Supabase provided JWT secret in our NextAuth). Basically, configure JWT so that our NextAuth tokens will be recognized.
8. Enable RLS on all relevant tables (as listed above).

9. Write RLS policies:

    - On `transactions`, `invoices`, etc. for tenant isolation (using `auth.uid()` and membership checks) [2]. Start with simple SELECT policy as described, then add write policies.
    - On `mx_merchant_configs` to restrict to superadmin (and possibly tenant's own row if needed).
    - On `user_tenants` for read/write as described (superadmin or tenant admin).
    - On `users` for superadmin only (and possibly self-access if we allow users to update their own profile/password – e.g., `WITH CHECK (auth.uid() = id)` for updates on their own user row [27]).
    - Test the policies with different roles by creating JWTs manually or using Supabase's testing features to ensure the logic is correct.

10. **NextAuth Setup**:

11. Install NextAuth and configure a Credentials provider. Set `NEXTAUTH_SECRET` to the Supabase JWT secret (so that tokens are compatible).
12. In the credentials authorize callback, implement logic to authenticate:
    1. Fetch user by email from `users` table (use Supabase client with service role, or a secure API route).
    2. Verify password hash.

3. If OK, return a user object containing at least `id`, `name`, `email`, and `role` info. Possibly also return `merchant_id` if the user is tenant-scoped (we can fetch the first `merchant_id` from `user_tenants` where user_id = ... if needed).

13. Implement NextAuth JWT and session callbacks:
    - **JWT callback**: when a user logs in, embed `uid` (user.id) as `sub` claim [11], and attach `role` (global role or derived tenant role). For example:

      ```
      token.sub = user.id;
      token.role = user.global_role || (user.isTenantAdmin ?
      "tenant_admin" : "tenant_user");
      token.merchantId = user.merchant_id || null;
      ```

      (We determine `isTenantAdmin` by checking if the user has a membership with role 'admin'; we might need to fetch that in authorize and attach to user.)
    - **Session callback**: map the token fields to session: e.g., `session.user.id = token.sub; session.user.role = token.role; session.user.merchantId = token.merchantId;`. This makes the info available on the client.

14. Ensure NextAuth uses JWT strategy (which it does by default for Credentials). We might increase the JWT size limit if needed to include claims.

15. **Frontend Integration (Supabase client)**:

16. After login, initialize the Supabase JS client with the user's token. For example, upon successful NextAuth sign-in, retrieve the JWT (NextAuth provides it via `getToken()` or we might need to expose it) and call `supabase.auth.setAuth(token)` or instantiate the client with `{ global: { headers: { Authorization: 'Bearer <token>' } } }`.
17. Verify that making a query like `supabase.from('transactions').select('*')` returns only that user's tenant data (test for a tenant user). Also test that a superadmin's query returns all tenants' data. These should be automatically enforced by RLS.

18. Implement UI logic:

    - Super Admin dashboard: fetch all tenants from `mx_merchant_configs` (allowed only for superadmin via RLS) and display. Provide controls to add new tenant (which calls an API route we'll implement). Provide "impersonate" or "view" action that sets a context to show that tenant's data (e.g., navigate to a route like `/admin/tenants/[merchantId]` which loads that tenant's dashboard using filtered queries).
    - Investor dashboard: similar to superadmin but without any create/edit buttons. Possibly the same pages but rendered read-only (no edit forms, and the Next.js API will additionally check role to reject any non-read operation).
    - Tenant Admin dashboard: on login, they directly see their own data (since they have only one tenant, no selection needed). Provide UI for managing users: list users (via membership table join) and invite/create new user. Provide a button to fetch new transactions (this will call a backend function).
    - Tenant User dashboard: similar to admin but without admin options (no user management, maybe no manual data fetch if that is restricted). Mostly just viewing data.

19. **APIs / Server-side Functions**:

20. **Create Tenant API** (Super Admin only): Next.js API route (e.g., POST `/api/tenants`) that checks `session.user.role` is superadmin. It will accept tenant details and credentials. It will: insert into `mx_merchant_configs` (new merchant_id, keys, etc.), and possibly create an initial tenant admin user for that tenant. Since this needs to insert multiple things (tenant config, user, membership), it should use the Supabase service role (bypassing RLS) or perform inserts as superadmin (which RLS would allow anyway for that role). Using service key on backend is simplest for multi-step transaction. Hash the provided password for the new admin user. Return success/failure.
21. **Toggle Tenant Access API**: For superadmin to set `is_active` false/true in `mx_merchant_configs`. This can be a protected route that updates the field. RLS will allow superadmin to update that table. (We should also consider that if `is_active=false`, our RLS on data tables could also check `JOIN mx_merchant_configs on merchant_id and ensure is_active=true` to automatically hide data for deactivated tenants, adding an extra layer. This is optional for now.)
22. **Fetch Transactions API**: Both Super Admin (for onboarding any tenant) and Tenant Admin (for their own tenant) can trigger fetching of transactions via MX API. We will implement this as a serverless function (Next.js API route or even better, use the existing Upstash Redis + Worker approach already partially in place [28] [26]). For phase 1, perhaps a simpler approach: Next.js API route `/api/fetchTransactions` that accepts a tenant id and number of records. It will verify the requester's role:
    - If superadmin: can fetch for any tenant (tenant id in request).
    - If tenant admin: can only fetch for their own tenant (we cross-check `session.user.merchantId`).
      Then it will use the stored credentials from `mx_merchant_configs` for that tenant, call the MX Merchant API for transactions (and subsequent invoice details, etc., as described). Insert the data into `transactions` and `invoices` tables. Because these inserts happen server-side with the service key, they bypass RLS but we ensure to set the correct `merchant_id` so that when the data is viewed, RLS allows the appropriate user.
      This process might already be implemented partially (webhook handling). We ensure that manual fetch uses the same logic safely (or just relies on the background worker by enqueueing a job as shown in the plan doc [29]).

23. **User Management APIs**:

    - Tenant Admin creating a new user: an API route `/api/tenantUsers` (POST) that accepts new user email, name, etc. It will check `session.user.role` is tenant_admin and create the user: insert into `users` (with a temp password or invite token) and insert into `user_tenants` with `merchant_id` = that admin's merchant and role = 'user'. It could email the credentials to the new user. We will ensure to hash passwords and possibly generate a random password if admin doesn't provide one (then email it).
    - Tenant Admin removing a user: e.g., DELETE `/api/tenantUsers/[userId]` – checks role and that the target user belongs to their tenant (we can verify by attempting to delete the membership row via RLS – if not allowed, it won't delete). Likely we allow cascade deletion: remove from `user_tenants`, and perhaps from `users` if that user isn't in any other tenant. Or we could simply mark them inactive. Simpler is to delete both membership and

user (since the user cannot exist without a tenant unless global, and tenant admins won't be deleting global users anyway).
    - Super Admin could have similar APIs to manage any user (like promote an investor or create another superadmin).

24. **Testing**:

25. Create a Super Admin user, an Investor user, and a Tenant with one admin and one user. Populate some data for that tenant. Then simulate login for each role and ensure:
    - Super Admin can query any tenant's data (e.g., via a combined dashboard or by switching context). Verify in the network calls or via pSQL that RLS is allowing them (their JWT role claim should satisfy the OR condition).
    - Investor can fetch the same data but try a write operation (like an update via an API) and confirm it's rejected (either by RLS or by our API returning 403).
    - Tenant Admin can get only their data. Try to access another tenant's endpoint (by ID) – should fail (RLS denies query or API checks fail). Ensure they can add a user and that user can then log in.
    - Tenant User can view data but not perform admin actions. Try an unauthorized action (like call the add-user API) and expect a 403.
26. Also test RLS by querying the DB as an anon or other role to ensure nothing leaks when not logged in or when logged as the wrong user.

Throughout this plan, we have drawn on **latest best practices** for multi-tenant SaaS authentication/authorization. Multi-tenant RBAC is implemented by scoping roles to each tenant [8], which we achieve with the membership table. Using NextAuth with a custom credential system gives us flexibility to manage roles and integrate with Supabase's JWT-based RLS [11]. And by keeping our schema lean and leveraging the existing structure, we avoid unnecessary complexity while ensuring **strict data isolation** (a must for any multi-tenant system) [30].

---

[1] [7] Multi-Tenant Not Possible with Auth.js (next-auth)? : r/nextjs
https://www.reddit.com/r/nextjs/comments/1am8804/multitenant_not_possible_with_authjs_nextauth/

[2] [5] [6] [24] How to Structure a Multi-Tenant Backend in Supabase for a White-Label App? : r/Supabase
https://www.reddit.com/r/Supabase/comments/1iyv3c6/how_to_structure_a_multitenant_backend_in/

[3] [4] [9] [10] [20] [21] database_schema.md
https://github.com/mudassar003/saas/blob/dd5b3fa95b60a192211839f9b4052c342e8b0f82/database_schema.md

[8] [15] [30] Best Practices for Multi-Tenant Authorization
https://www.permit.io/blog/best-practices-for-multi-tenant-authorization

[11] How can I create a custom auth.uid() for NextAuth? : r/Supabase
https://www.reddit.com/r/Supabase/comments/1jcogdz/how_can_i_create_a_custom_authuid_for_nextauth/

[12] [27] node.js - How to make custom jwt auth work on Supabase? - Stack Overflow
https://stackoverflow.com/questions/76649583/how-to-make-custom-jwt-auth-work-on-supabase

[13] [14] Data Schema - Saas Starter Kit
https://www.saasstarterkit.com/docs/database/schema

16 17 18 19 23 Authorization via Row Level Security | Supabase Features
https://supabase.com/features/row-level-security

22 25 26 28 29 plan.md
https://github.com/mudassar003/saas/blob/dd5b3fa95b60a192211839f9b4052c342e8b0f82/plan.md