ChatGPT

Implementing Secure ABA Session Management with Supabase (Backend Only)

Database Schema Design

Design a **multi-tenant** Postgres schema (managed by Supabase) that separates data by organization (ABA provider) and supports sessions, notes, and scheduling. Key tables might include:

- **Therapists** – holds therapist profiles (e.g. name, credentials) and organizational context. For example: `id` (UUID primary key, often referencing the auth user's ID), `organization_id` (UUID reference to an organization/clinic), `specialties` (array), `availability_hours` (JSON schedule). Each therapist row links to exactly one Supabase Auth user (for login) and is tied to an organization [1].
- **Clients** – stores client (patient) info, with similar fields: `id` (UUID primary key), `organization_id` (UUID reference), personal details, etc. Use a `deleted_at` timestamp or `status` field for soft deletes (archive) rather than outright deletion, since medical records shouldn't be silently removed [1].
- **Sessions** – represents therapy sessions (scheduled appointments). Includes `id` (UUID), `client_id` (UUID ref to **Clients**), `therapist_id` (UUID ref to **Therapists**), `organization_id`, `start_time` and `end_time` (timestamptz for schedule), and fields for tracking attendance and notes. For example, a `status` enum (like `'scheduled'`, `'completed'`, `'cancelled'`) or a boolean `attended` flag can mark client attendance. A `notes` text column (or a separate **SessionNotes** table linking session_id → note text) stores the therapist's session notes. To prevent overlapping bookings, consider a **unique index or check** on sessions (e.g. no two sessions for the same therapist at overlapping times). Index critical fields like `organization_id`, `therapist_id`, `client_id`, and `start_time` to optimize queries (e.g. quickly fetching an org's schedule) [1].

Below is an example schema (PostgreSQL SQL) for these tables:

```sql
-- Example table schemas (simplified)
CREATE TABLE organizations (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  name TEXT NOT NULL
);

CREATE TABLE therapists (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  user_id UUID UNIQUE REFERENCES auth.users (id) NOT NULL,  -- link to Supabase
Auth
  organization_id UUID REFERENCES organizations (id) NOT NULL,
  name TEXT,
  specialties TEXT[] DEFAULT '{}',
  availability_hours JSONB,  -- e.g. store weekly available slots
  created_at TIMESTAMPTZ DEFAULT now()
);

CREATE TABLE clients (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  organization_id UUID REFERENCES organizations (id) NOT NULL,
  full_name TEXT,
  date_of_birth DATE,
  status TEXT DEFAULT 'active',         -- e.g. 'active' or 'archived'
  deleted_at TIMESTAMPTZ,               -- use for soft-delete/archival
  created_at TIMESTAMPTZ DEFAULT now()
);

CREATE TABLE sessions (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  organization_id UUID REFERENCES organizations (id) NOT NULL,
  therapist_id UUID REFERENCES therapists (id) NOT NULL,
  client_id UUID REFERENCES clients (id) NOT NULL,
  start_time TIMESTAMPTZ NOT NULL,
  end_time   TIMESTAMPTZ NOT NULL,
  status TEXT DEFAULT 'scheduled',      -- e.g.
'scheduled','completed','cancelled'
  attended BOOLEAN,                     -- TRUE if client attended (or use
status)
  notes TEXT,                           -- session notes by therapist
  created_at TIMESTAMPTZ DEFAULT now(),
  updated_at TIMESTAMPTZ DEFAULT now()
);
```

In this design, every record ties back to an `organization_id`. This makes it easy to enforce **tenant isolation** – each ABA provider (tenant) only accesses their own clients, therapists, and sessions. We also include **foreign keys** (e.g. each session links to a specific client and therapist) and suggest **composite indexes** such as `(organization_id, start_time)` or `(organization_id, therapist_id)` on the sessions table to accelerate filtering by org, date, or therapist [1].

Role-Based Access Control (RBAC) & Row-Level Security (RLS)

Supabase's Postgres database lets us enforce backend security using **roles and row-level security policies**. Enable RLS on each sensitive table (clients, therapists, sessions, notes) so that no data is accessible by default until policies are in place [2] [3]. All application queries from the client will use the `authenticated` Postgres role (for logged-in users), so we define policies targeting that role.

User Roles: Define application roles like "therapist", "admin" (clinic administrator), etc., so that permissions can differ. For example, an **admin** might see or edit all data in their organization, whereas a **therapist**

should only see their own sessions/notes and clients. One approach is to store roles in a join table (e.g. a `user_roles` table mapping each user to a role and organization) or in the user's JWT via Supabase custom claims [4]. Using a custom JWT claim for role and even `organization_id` can simplify policies (the claim is accessible via `auth.jwt()` in SQL) and avoid heavy subqueries [4].

RLS Policies: With RLS enabled, create policies to enforce that users can only read or write allowed rows. For example, to ensure therapists only access their own sessions, and admins can access all org sessions, you could write:

```
-- Enable RLS on relevant tables
ALTER TABLE public.sessions ENABLE ROW LEVEL SECURITY;

-- Policy: Therapists can view their own sessions, Admins can view all sessions
in org
CREATE POLICY "Sessions_select_own_or_org"
ON public.sessions
FOR SELECT
TO authenticated
USING (
  -- Therapist can see if they are assigned to the session
  (therapist_id IN (SELECT t.id FROM public.therapists t
                    WHERE t.id = public.sessions.therapist_id
                      AND t.user_id = auth.uid()))
  OR
  -- Org admin can see any session in their organization
  (EXISTS (
     SELECT 1 FROM public.user_roles ur
     WHERE ur.user_id = auth.uid()
       AND ur.role = 'admin'
       AND ur.organization_id = public.sessions.organization_id
  ))
);

-- Similarly, define INSERT/UPDATE policies (with CHECK) to restrict who can
create or modify sessions.
```

In the above policy, the condition uses a subquery to check that either (a) the session's therapist matches the logged-in user, or (b) the logged-in user has an "admin" role for that session's organization. This ensures **therapists see only their own sessions**, while org admins see all sessions for their clinic [1]. We would create analogous policies for **Clients** and **Therapists** tables (e.g. therapists can see client data only if they are assigned to that client or in the same org, etc.) [1]. Insert/Update policies should use `WITH CHECK` conditions to ensure new or changed rows still belong to the user's org and respect their role.

> **Tip:** Supabase's `auth.uid()` returns the user's ID, and custom claims can be read via `auth.jwt()`. As an alternative to subqueries, you can use a function or JWT claim to get the user's org ID and role. For example, a policy could do `USING (organization_id =`

`(auth.jwt() ->> 'org_id')::uuid)` if you stored the org in the token [5]. This approach avoids a join for every query. Just ensure the JWT claims are set on sign-in (Supabase provides an Auth hook for custom claims).

Also consider using **Postgres security definer functions** for any privileged operations. For instance, an RPC (remote procedure) that an admin calls to modify data can be created with `SECURITY DEFINER` and limited to users with the admin role. In our design, we could have a stored procedure like `archive_client(client_id)` marked as definer that first checks `auth.uid()` has admin rights for that client's org, then updates `clients.deleted_at`. This lets admins perform tasks beyond a therapist's privilege, while still keeping RLS enabled on the tables [1]. Always **grant EXECUTE** on such functions only to appropriate roles (e.g. the `authenticated` role) and perform role checks inside the function before executing sensitive updates.

Secure Storage of Session Notes & PHI

Session notes contain sensitive health information (considered PHI under HIPAA), so protecting them is paramount. In the schema above, session notes are stored in a text column within the `sessions` table (or a related `session_notes` table). Supabase's managed Postgres already ensures data is **encrypted at rest and in transit** (AES-256 at rest, TLS in transit) [6]. This meets baseline encryption requirements. However, to further protect highly sensitive notes, you might additionally encrypt the notes at the application level (for instance, using a library to encrypt the text before saving, and storing encryption keys securely) [7]. This would render the data unreadable even if the database were directly accessed, adding a defense in depth.

Use RLS to strictly limit who can read or write the `notes` field. The policies described above already ensure only the session's therapist or an org admin can fetch that session row [1]. We should also ensure that if a **client portal** exists (for example, parents viewing session summaries), those users have a limited role that perhaps can `SELECT` certain fields but not the full confidential note unless authorized. This can be achieved with a separate RLS policy or a database view that filters out sensitive columns.

Because this is healthcare data, implement auditing where feasible. For example, add a Postgres trigger on the sessions or notes table to record changes (who made an edit to a note and when). An audit log table with entries like `session_id`, `changed_by`, `changed_at`, `old_value`, `new_value` can help with compliance audits. This ensures there is a trail of who accessed or modified PHI, aligning with HIPAA's security rule requirements for record integrity and audit controls.

Finally, **never expose PHI via insecure channels**. That means any logs, error messages, or front-end analytics should avoid containing session notes or personal details. If you use Supabase Storage for file uploads (e.g. therapy documents or images), keep those buckets **private** (no public URL) and serve files to authenticated users on demand. Supabase allows securing bucket objects with policies similar to RLS – e.g. only allow download if `auth.uid()` matches the client's therapist or is an admin. And as the Supabase docs emphasize, **do not store PHI in public buckets** (publicly accessible storage) [8].

Session Scheduling & Attendance Tracking

The **Sessions** table design already supports scheduling by storing start and end timestamps. To manage therapist-client scheduling, you will likely implement logic (in application code or with database constraints) to ensure no conflicts. Some best practices:

- **Unique or Exclusion Constraint:** Prevent double-booking by ensuring a therapist can't have two sessions that overlap in time. In Postgres, an exclusion constraint on `(therapist_id, tstzrange(start_time, end_time))` with the `&&` (overlap) operator can enforce no overlapping sessions for the same therapist. Alternatively, use a simpler approach: before inserting a new session, query for any existing session for that therapist with timing overlap and reject if found.
- **Therapist Availability:** If therapists have working hours or days off, maintain an **availability schedule**. For example, a JSONB `availability_hours` in the therapists table could store available times, or use a separate table like `therapist_availability` (with columns: therapist_id, weekday, start_time, end_time) and an `availability_exceptions` table for time-off/holidays [9] [10]. This isn't strictly required by Supabase, but having this data in the backend lets you enforce scheduling rules (via queries or maybe future Supabase Edge Functions to check availability before booking).
- **Recurring Sessions:** If ABA sessions recur (e.g. same time every week), you might store a recurrence rule or generate future sessions in advance. Supabase doesn't have a built-in scheduler for recurring events, so your backend logic (or cron jobs) would handle creating those sessions. The database design might include a `recurrence_id` to link sessions that are part of a series, but that's an application-specific detail.

Attendance tracking can be handled via the `status` or `attended` fields in the sessions table. For example, when a session is completed, the therapist (or an automated process) marks it as `completed` and maybe sets `attended = TRUE` or records a `no_show_reason`. It's often useful to distinguish cancellations vs. no-shows vs. attended, so a small lookup table or enum for "attendance status" (e.g. *Attended, Cancelled by Therapist, Cancelled by Client, No-Show*) can be used. This makes reporting easier (e.g. how many sessions were attended this month). Ensure that updating attendance status is protected by RLS as well – e.g. only the session's therapist or an admin can mark a session as attended or cancelled. You might create a separate policy for `UPDATE` on the sessions table: e.g. therapist can update the `status` of their own session, but not someone else's.

Because the **schedule and attendance data** are sensitive (they reveal client participation in therapy), RLS policies should also cover the sessions table for *SELECT*. The earlier example policy `Sessions_select_own_or_org` already does this. In practice, you might even split policies by operation: one for SELECT (data viewing) and one for UPDATE (marking attendance or editing notes) to fine-tune permissions [11] [12]. For instance, a therapist might have SELECT and UPDATE on their sessions, but perhaps not allowed to DELETE sessions (maybe only admins can delete or cancel sessions). Supabase allows multiple policies per table, so you can say "therapists can update their own session rows" (with a `WITH CHECK` ensuring they can only change the `status/notes` fields, not assign themselves to someone else's session) and another policy "admins can update all sessions in their org". This aligns with the principle of least privilege.

Compliance with California Healthcare Privacy Regulations

Building on the above security measures, ensure the solution aligns with **HIPAA** and California-specific laws (such as the **California Confidentiality of Medical Information Act (CMIA)**). Supabase as a platform can be used in a HIPAA-compliant way – but you **must configure it appropriately**:

- **Business Associate Agreement (BAA):** If using Supabase's hosted service, sign a BAA with Supabase (available on the Enterprise or appropriate plan) and mark your project as a HIPAA project [13] [14]. This is required for Supabase to legally handle Protected Health Information on your behalf.
- **Enable HIPAA safeguards:** Supabase's shared responsibility model lists steps you should take for a HIPAA app [15]. For example, **enforce Multi-Factor Authentication** for all users (especially admins) [16], enable **Point-in-Time Recovery** for the database (so you can recover data in case of issues) [17], and turn on **SSL-only** enforcement. Also use **Network Restrictions** (ip allowlists or similar) if possible to limit access to the database [18].
- **Encryption & Storage:** Ensure all connections use TLS and data stays encrypted at rest – Supabase covers this by default (AES-256 at rest, TLS in transit) [6]. Avoid storing any PHI in a form that's publicly accessible. For instance, do not put client-identifiable data in URLs or in public storage buckets [8]. If you use Supabase Storage for files, use private buckets and serve files via signed URLs or through the Supabase API with RLS checks.
- **Data Access Controls:** The RLS policies we set up are a major part of HIPAA compliance, ensuring only authorized personnel access patient data (meeting the Privacy Rule's "minimum necessary" access principle). Regularly review these policies. For example, verify that when a therapist leaves or changes role, their user account is updated (or deactivated) such that RLS prevents further access to client records. The `app.user_has_role_for_org` approach shown earlier centralizes this logic – if a user's role or org membership is revoked, that function returns false and RLS denies access [1].
- **Auditing and Breach Monitoring:** Implement logging of access to PHI. Supabase provides automatic logging of all queries in the dashboard, but you might supplement this by logging application-level actions (e.g. an edge function logs when a session note is created or viewed, tagging which user did it). California law and HIPAA both require notifying patients in case of breaches, so having detailed logs helps in forensic analysis. Consider setting up alerts or reports for unusual access patterns (for example, if a therapist suddenly accesses an abnormal number of client records, an admin could be notified).

By following these practices – strong data isolation (RLS), encrypted storage, careful role management, and adhering to Supabase's HIPAA configuration checklist – your ABA platform's backend will securely handle session notes, scheduling, and attendance in compliance with HIPAA and California regulations. Supabase's official stance confirms it **is HIPAA-compliant** as a host once the proper agreements and configurations are in place [13], so with a solid implementation, you can use it to store and manage sensitive ABA therapy data with confidence.

Sources:

1. Supabase Documentation – *Row Level Security (RLS) and policy examples* [3] [19]
2. Supabase Documentation – *Custom Claims & Role-Based Access Control* (using JWT for roles) [4]
3. Supabase Documentation – *Security & HIPAA Compliance (Shared Responsibility Model)* [15] [8]
4. Supabase Security Notes – *Encryption at rest & in transit; HIPAA compliance with BAA* [6] [13]

5. Internal Architecture Review – *Example multi-tenant RLS policies for therapists, clients, sessions* [1] (demonstrating org-based access control in a Supabase ABA app)

---

[1] clients_supabase_static.md
https://github.com/Jeduardo622/AllIncompassing/blob/98dbacdcc48cff8f54c5bc47a6b76f81976a7feb/reports/clients_supabase_static.md

[2] [3] Row Level Security | Supabase Docs
https://supabase.com/docs/guides/database/postgres/row-level-security

[4] [19] Supabase Row Level Security Explained With Real Examples | by debug_senpai | Medium
https://medium.com/@jigsz6391/supabase-row-level-security-explained-with-real-examples-6d06ce8d221c

[5] Multi-Tenant Applications with RLS on Supabase (Postgress) | Build AI-Powered Software Agents with AntStack | Scalable, Intelligent, Reliable
https://www.antstack.com/blog/multi-tenant-applications-with-rls-on-supabase-postgress/

[6] [13] Security at Supabase
https://supabase.com/security

[7] [8] [14] [15] [16] [17] [18] Shared Responsibility Model | Supabase Docs
https://supabase.com/docs/guides/deployment/shared-responsibility-model

[9] [10] therapists_supabase_static.md
https://github.com/Jeduardo622/AllIncompassing/blob/98dbacdcc48cff8f54c5bc47a6b76f81976a7feb/reports/therapists_supabase_static.md

[11] [12] 20251223131500_align_rls_and_grants.sql
https://github.com/Jeduardo622/AllIncompassing/blob/98dbacdcc48cff8f54c5bc47a6b76f81976a7feb/supabase/migrations/20251223131500_align_rls_and_grants.sql