<https://gemini.google.com/u/1/app/2d30425fc8a5984a>

### **1. Auth Flow Diagram**

* **Chosen Auth Provider:** **Supabase Auth**
  + Rationale: Supabase Auth is tightly integrated with the Supabase ecosystem, providing JWT-based authentication that works seamlessly with PostgreSQL Row-Level Security (RLS) and PostgREST. It supports various providers (email/password, OAuth, magic links) and handles user management.
* **Authentication Flows:**
* Code snippet

sequenceDiagram

participant C as Client (Web/Mobile App)

participant SA as Supabase Auth API

participant DB as Supabase DB (PostgreSQL)

participant P as Profiles Table (in DB)

%% Signup Flow %%

C->>+SA: Signup with email/password or OAuth

SA->>+DB: Create user in auth.users

DB->>SA: User created (auth.uid(), email)

SA-->>-C: Confirmation email sent / OAuth redirect / Session (JWT)

Note over SA,DB: public.handle\_new\_user() trigger/fn on auth.users creates entry in public.profiles, assigns default role(s).

%% Login Flow %%

C->>+SA: Login with email/password or OAuth

SA->>+DB: Verify credentials / OAuth provider token

DB->>SA: Credentials valid / User identity confirmed

SA-->>-C: Session (Access Token JWT, Refresh Token)

Note over SA,P: JWT contains auth.uid(), role ('authenticated'), email, etc. User's specific application roles are in public.profiles.roles TEXT[].

%% Token Refresh Flow (typically handled by Supabase client libraries) %%

C->>SA: Request with expired Access Token (or proactively using Refresh Token)

SA->>SA: Validate Refresh Token

SA-->>C: New Access Token JWT (and potentially new Refresh Token)

%% API Request Flow %%

C->>DB: API Request (PostgREST/GraphQL) with Authorization: Bearer <Access\_Token\_JWT>

DB->>DB: Validate JWT; Apply RLS policies based on auth.uid() and helper functions querying public.profiles.roles.

DB-->>C: Response (Data or Error)

%% Logout Flow %%

C->>+SA: Logout request

SA->>SA: Invalidate session/tokens

SA-->>-C: Logout confirmation

* **JWT Roles/Claims Injection:**
  + Supabase Auth JWTs inherently contain sub (user ID, i.e., auth.uid()), role (e.g., authenticated, anon), email, aud, iss, iat, exp.
  + Application-specific roles (e.g., pilgrim, regional\_manager, platform\_admin) are stored in the public.profiles.roles TEXT[] column.
  + RLS policies will **not directly use a custom roles claim in the JWT by default with Supabase.** Instead, they will use auth.uid() from the JWT to call PostgreSQL helper functions (e.g., public.has\_role(TEXT)). These functions will query the public.profiles table to check the user's roles. This is the standard and recommended Supabase pattern for RLS with custom roles.

### **2. Roles & Permission Matrix**

* **Roles Defined:**
  + anonymous: Unauthenticated users.
  + authenticated\_user (or pilgrim): Basic logged-in user.
  + accommodation\_host: Manages their accommodation listings.
  + regional\_content\_manager: Manages local information (trails, POIs, articles) for assigned regions.
  + moderator: Moderates user-generated content (e.g., tips, reviews).
  + platform\_administrator (or admin): Manages platform-wide content, users, and configurations. This is the highest application-level role. *(Note: The prompt listed admin and platform\_admin. I'm using platform\_administrator as the primary top-tier application role based on RLS stubs like is\_platform\_admin(). If admin is a distinct, lesser role, its permissions would need to be defined separately. For now, I'll assume they are largely synonymous or platform\_administrator is the specific Supabase role name we'll use.)*
  + (Supabase system roles: service\_role for backend operations bypassing RLS, supabase\_admin for project infrastructure management – these are not user-facing roles.)
* **Permission Mapping (Conceptual - RLS policies define the specifics):** | Role | RLS Policy Group Access (Examples from Module 2 & general concepts) | Example API Scope / Capability (Conceptual) |  
  | :--------------------------- | :----------------------------------------------------------------------------------------------------- | :------------------------------------------------------------------------------------------------------------------------------------------------------------------------------ |  
  | Anonymous | Read-only on published & non-deleted trails, routes, segments, waypoints, articles, master data (is\_active=true). | read:public\_content, read:master\_data |  
  | Authenticated User/Pilgrim | All Anonymous + Create/Read/Update/Delete own user\_waypoint\_votes, user\_waypoint\_short\_tips, accommodation\_reviews. Read/Update own profiles record. | read:public\_content, read:master\_data, write:own\_ugc, manage:own\_profile |  
  | Accommodation Host | All Authenticated User + Create/Read/Update/Delete own accommodations waypoints and associated details. | All Pilgrim + manage:own\_accommodations |  
  | Moderator | All Authenticated User + Update status of user\_waypoint\_short\_tips, accommodation\_reviews (moderation status). Access to moderation queues/views. | All Pilgrim + moderate:ugc |  
  | Regional Content Manager | All Authenticated User + Create/Update/Delete trails, routes, segments, waypoints, articles within assigned regions. Manage segment\_warnings for their regions. | All Pilgrim + manage:regional\_content(region\_ids), manage:regional\_warnings(region\_ids) |  
  | Platform Administrator | Full CRUD on most tables (trails, routes, segments, waypoints, articles, all master data, user\_roles\_master, translations, media). Manage user roles. | manage:all\_content, manage:users, manage:translations, manage:media, manage:platform\_settings |  
    
   *RLS policies on each table will use helper functions like public.has\_role(TEXT), public.is\_platform\_admin(), user\_manages\_region(UUID, region\_id) etc., which query public.profiles.roles using auth.uid().*

### **3. Security Headers & Cookies**

* **Required Headers/Cookies per Request Type:**
  + **Authenticated API Requests (to PostgREST/GraphQL):**
    - Authorization: Bearer <SUPABASE\_ACCESS\_TOKEN\_JWT>: Essential for authenticating the user and applying RLS.
    - APIKey: <SUPABASE\_ANON\_KEY>: Supabase requires this key even for authenticated requests to identify the project.
    - Content-Type: application/json (for POST/PUT/PATCH requests).
    - Accept: application/json.
  + **Anonymous API Requests (to PostgREST/GraphQL):**
    - APIKey: <SUPABASE\_ANON\_KEY>: Identifies the Supabase project.
    - Accept: application/json.
  + **Supabase Auth Endpoints (e.g.,** /auth/v1/token**,** /auth/v1/signup**,** /auth/v1/logout**):**
    - Handled by Supabase client libraries. Typically involve APIKey, Content-Type. Cookies (e.g., sb-refresh-token) are managed by Supabase for session persistence and token refresh, usually HttpOnly and Secure.
* **Sample** curl **for Authenticated Request:**
* Bash

curl -X GET \

'https://<YOUR\_SUPABASE\_PROJECT\_REF>.supabase.co/rest/v1/trails?select=\*&id=eq.1' \

-H "APIKey: <YOUR\_SUPABASE\_ANON\_KEY>" \

-H "Authorization: Bearer <USER\_ACCESS\_TOKEN\_JWT>" \

-H "Accept: application/json"

### **4. Error & Rate-Limit Strategy**

* **Standard Error Object (JSON):**
* JSON

{

"code": "ERROR\_CODE\_STRING", // e.g., "PGRST116", "AUTH\_INVALID\_CREDENTIALS", "VALIDATION\_ERROR"

"message": "A human-readable summary of the error.",

"detail": "Optional: More specific details or field errors.", // Can be an object for validation errors

"hint": "Optional: Suggestion on how to resolve the error."

}

* *PostgREST provides errors in a similar structure (code, details, hint, message). Custom errors from DB functions or API gateway should align.*
* **HTTP Status Codes Usage:**
  + 200 OK: Successful GET, PUT, PATCH, DELETE.
  + 201 Created: Successful POST.
  + 204 No Content: Successful DELETE with no body returned.
  + 400 Bad Request: General client-side error (e.g., malformed JSON, invalid parameters).
  + 401 Unauthorized: Missing or invalid authentication credentials (JWT).
  + 403 Forbidden: Authenticated user does not have permission to access the resource (RLS denial).
  + 404 Not Found: Resource does not exist.
  + 409 Conflict: Resource creation/update failed due to a conflict (e.g., unique constraint violation).
  + 422 Unprocessable Entity: Semantic errors in the request payload (e.g., validation failed for a field).
  + 429 Too Many Requests: Rate limit exceeded.
  + 500 Internal Server Error: Unexpected server-side error.
* **Basic Global Rate-Limit Policy:**
  + **Supabase Auth Endpoints:** Supabase has built-in rate limiting for authentication actions (e.g., login attempts, password resets, email sending).
  + **API (PostgREST/GraphQL) Endpoints:**
    - Default: Supabase provides some level of abuse prevention.
    - Recommended: Implement IP-based rate limiting for anonymous users (e.g., 60 requests per minute per IP).
    - Authenticated users can have higher limits (e.g., 240 requests per minute per user).
    - Consider using an API Gateway in front of Supabase for more sophisticated rate limiting, quotas, and DDOS protection if needed at scale.

### **5. Impact Checklist**

* **New DB Columns, Indexes, or RLS Helpers Required?**
  + 🟠 profiles.roles TEXT[] **Index:** If RLS helper functions (like has\_role()) frequently query the public.profiles.roles TEXT[] column, a **GIN index** on this column is highly recommended for performance: CREATE INDEX IF NOT EXISTS idx\_profiles\_roles\_gin ON public.profiles USING GIN (roles); (The profiles table itself is from Module 1, but its roles column is central to auth across all modules).
  + 🔴 **RLS Helper Functions Implementation:** The conceptual RLS policies rely on various helper functions (e.g., public.has\_role(TEXT), public.is\_platform\_admin(), public.user\_manages\_region(UUID, BIGINT), public.check\_user\_can\_edit\_trail(BIGINT), etc.). These functions **must be implemented securely and efficiently** in PostgreSQL. Their DDLs should be part of the database schema scripts.
  + 🟠 user\_roles\_master **and** profiles.roles **Sync:** Ensure the mechanism to keep profiles.roles synchronized with user\_roles\_master (potentially via the public.check\_profile\_roles() function mentioned in the overview) is robust and tested.
* **Any changes to existing specs?**
  + 🟠 **All Table Specs (RLS Sections):** The RLS sections in individual table specifications that currently list placeholder helper functions should be updated to reference the actual, implemented helper function names once defined.
  + 🟠 **Module 1** profiles **table spec:** Ensure the GIN index on roles is documented if added.
  + No other direct changes to existing Module 2 table structures seem immediately necessary *solely* from this auth architecture, as the design already anticipated role-based access via profiles.roles.