<https://gemini.google.com/u/1/app/117b6ecbb3fe7734>

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

* **Chosen Auth Provider**: **Supabase Auth**
  + **Rationale**: The project context strongly indicates Supabase as the backend platform. Supabase Auth is tightly integrated, provides JWT-based authentication, supports role-based access, and allows for easy user profile management via triggers to a public.profiles table.
* **Authentication Flows & JWT Role Injection**:
* Code snippet

sequenceDiagram

participant ClientApp as Client Application

participant SupabaseAuth as Supabase Auth

participant Database as PostgreSQL Database (Supabase)

%% Signup Flow

ClientApp->>SupabaseAuth: 1. Signup (email/pass or OAuth)

SupabaseAuth->>Database: 2. Create user in auth.users

Database-->>Database: 3. Trigger: public.handle\_new\_user()

Database-->>Database: 4. Create profile in public.profiles (with default role, e.g., 'pilgrim')

SupabaseAuth-->>ClientApp: 5. Return User Session (Access Token, Refresh Token)

%% Login Flow

ClientApp->>SupabaseAuth: 6. Login (email/pass or OAuth)

SupabaseAuth->>Database: 7. Verify credentials & Fetch user roles from public.profiles (via custom claim function or similar mechanism)

Database-->>SupabaseAuth: 8. Return roles

SupabaseAuth-->>SupabaseAuth: 9. Inject roles (e.g., ['pilgrim', 'regional\_manager']) into JWT `app\_metadata.roles` or custom claim

SupabaseAuth-->>ClientApp: 10. Return User Session (Access Token with roles, Refresh Token)

%% Authenticated API Request

ClientApp->>Database: 11. API Request with JWT (e.g., to PostgREST)

Database->>Database: 12. Validate JWT, Extract user\_id & roles

Database->>Database: 13. Apply RLS policies based on user\_id & roles

Database-->>ClientApp: 14. Return API Response (data or error)

%% Token Refresh Flow

ClientApp->>SupabaseAuth: 15. Request new Access Token (with Refresh Token)

SupabaseAuth->>Database: 16. (Optional) Re-fetch roles if they can change frequently

SupabaseAuth-->>SupabaseAuth: 17. Inject roles into new JWT

SupabaseAuth-->>ClientApp: 18. Return new Access Token

%% Logout Flow

ClientApp->>SupabaseAuth: 19. Request Logout (invalidate session)

SupabaseAuth-->>SupabaseAuth: 20. Invalidate Refresh Token (if possible/configured)

ClientApp-->>ClientApp: 21. Discard Access & Refresh Tokens

* **Notes on JWT Role Injection**:  
  + User roles, stored in public.profiles.roles TEXT[], are injected into the Supabase JWT as a custom claim (e.g., app\_metadata.roles or a namespaced claim like https://<your-app-url>/roles).
  + This injection can be achieved using PostgreSQL functions triggered during token generation or by configuring custom JWT claims in Supabase if the feature directly supports dynamic database lookups for claims. RLS policies will then use functions like auth.jwt() ->> 'app\_metadata.roles' or custom helper functions to access these roles.

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

| **Role** | **Description** | **RLS Policy Group Examples** | **API Scope Examples** | **Source Document Reference** |
| --- | --- | --- | --- | --- |
| anonymous | Unauthenticated public user. | USING (is\_publicly\_visible = true AND deleted\_at IS NULL) on content. | GET /articles (published), GET /waypoints (published), GET /waypoint\_categories, GET /tags. | General RLS design |
| authenticated\_user | Basic logged-in user (Pilgrim). | USING (auth.uid() = created\_by\_profile\_id OR (is\_publicly\_visible...)) for own content; access to profiles for self. | All anonymous scopes; POST /reviews; PUT /profiles/me; GET /user/itineraries/my\_saved\_itineraries. | (Pilgrims) |
| accommodation\_host | User managing specific accommodation(s). | USING (user\_manages\_accommodation(auth.uid(), waypoint\_id)) on accommodations table. | All authenticated\_user scopes; PUT /accommodations/{my\_accommodation\_id}; GET /accommodations/{my\_accommodation\_id}/bookings. | (Accommodation Hosts) |
| regional\_content\_manager | User managing content for assigned geographical region(s). | USING (user\_manages\_region(auth.uid(), region\_id)) on waypoints, articles, segment\_warnings. | All authenticated\_user scopes; POST /waypoints (in their region); PUT /articles/{article\_id\_in\_region}; DELETE /segment\_warnings/{warning\_id\_in\_region}. | (Regional Content Managers) |
| admin | Administrator with broad content and user management capabilities. | USING (has\_role('admin')) granting wider access, bypass some ownership checks. | All regional\_content\_manager scopes (potentially cross-region); PUT /users/{user\_id}/roles; DELETE /reviews/{review\_id} (moderation). | Based on is\_platform\_admin() etc. |
| platform\_admin | Super administrator with full system control. | USING (has\_role('platform\_admin')) - often results in near-unfettered access for RLS (or bypasses RLS for some ops). | All admin scopes; CRUD on \*\_master tables (e.g., user\_roles\_master, languages\_master); manage platform settings; manage other admin accounts. | (Platform Administrators) |

* **RLS Helper Functions like** public.has\_role(TEXT) will be crucial for implementing these policy groups.

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

* **Required Headers/Cookies per Request Type**:
  + **All API Requests (to Supabase PostgREST/GraphQL)**:
    - apikey: <your-supabase-anon-public-key>: Identifies the Supabase project. This key is public and allows access based on RLS policies for anonymous users if no Authorization header is present.
  + **Authenticated API Requests**:
    - Authorization: Bearer <supabase\_jwt\_access\_token>: The JWT access token obtained after login.
  + **General Web Application Security Headers** (typically set by a reverse proxy or web server):
    - Content-Security-Policy (CSP): Defines allowed sources for content, mitigating XSS.
    - Strict-Transport-Security (HSTS): Enforces HTTPS.
    - X-Content-Type-Options: nosniff: Prevents MIME-sniffing.
    - X-Frame-Options: DENY or SAMEORIGIN: Protects against clickjacking.
    - Referrer-Policy: strict-origin-when-cross-origin.
* **Cookies**:
  + Supabase Auth uses secure, httpOnly cookies primarily for managing refresh tokens and session information on the auth server itself. API requests from client applications (SPAs, mobile apps) will typically use the Authorization: Bearer token header. If the application is a server-side rendered web app interacting directly with Supabase Auth, cookies might play a more direct role in session management for the browser.
* **Sample** curl **Request (Authenticated)**:
* Bash

curl -X GET \

'https://<your-project-ref>.supabase.co/rest/v1/waypoints?select=name,description&limit=5' \

-H "apikey: <your-supabase-anon-public-key>" \

-H "Authorization: Bearer <user\_jwt\_access\_token>"

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

* **Standard Error Object**:
  + Supabase (PostgREST) provides a standard error response format. We should align with or consistently wrap this:
  + JSON

{

"message": "A human-readable message explaining the error.",

"code": "INTERNAL\_ERROR\_CODE\_OR\_PGRST\_CODE", // e.g., "PGRST116", "22P02", "AUTH\_FORBIDDEN"

"hint": "Optional: A hint on how to solve the error.", // Provided by PostgREST

"details": "Optional: More specific details, or structured validation errors."

// For validation errors: "details": {"field\_name": ["error description 1", "error description 2"]}

}

* **HTTP Status Codes**:
  + 200 OK: Successful GET, PUT, PATCH, DELETE.
  + 201 Created: Successful POST.
  + 204 No Content: Successful DELETE (if no body returned).
  + 400 Bad Request: Invalid request syntax, missing parameters, validation errors (e.g., invalid data type for a field).
  + 401 Unauthorized: Authentication token is missing, invalid, or expired.
  + 403 Forbidden: Authenticated user does not have permission to access or modify the resource (RLS denial).
  + 404 Not Found: The requested resource does not exist.
  + 409 Conflict: Attempt to create a resource that already exists or would violate a unique constraint.
  + 422 Unprocessable Entity: Request was well-formed but contains semantic errors (e.g., business rule violation not caught by basic validation).
  + 429 Too Many Requests: Rate limit exceeded.
  + 500 Internal Server Error: Unexpected server-side error.
* **Basic Global Rate-Limit Policy**:
  + **Anonymous Users**: e.g., 60 requests per IP address per minute.
  + **Authenticated Users**: e.g., 120 requests per user ID per minute.
  + **Sensitive Operations** (e.g., login attempts, password resets): Stricter limits (e.g., 5 attempts per 15 minutes).
  + Implementation: Via Supabase's built-in abuse prevention or an API gateway if one is used in front of Supabase.

### **5. Impact Checklist**

* **New DB Columns, Indexes, or RLS Helpers Required?**:
  + 🔴 public.profiles.roles TEXT[]: Ensure this column is definitively part of the profiles table spec and managed by public.handle\_new\_user(). (This seems to be covered by Module 1 specs as per overview ).
  + 🔴 **RLS Helper Function:** public.has\_role(role\_code TEXT) RETURNS BOOLEAN: This function, which checks if auth.uid() has a specific role in public.profiles.roles, is essential and must be robustly defined and secured. (Marked as critical in VDF Overview).
  + 🔴 **RLS Helper Function:** public.user\_manages\_region(profile\_id UUID, target\_region\_id INTEGER) RETURNS BOOLEAN:
    - Requires a new junction table: profile\_managed\_regions (profile\_id UUID FK public.profiles(id), region\_id INTEGER FK public.regions(id), PRIMARY KEY (profile\_id, region\_id)).
    - This helper function would query profile\_managed\_regions.
  + 🔴 accommodations.host\_profile\_id UUID FK public.profiles(id): Required for the accommodation\_host role to identify ownership. RLS helper public.user\_manages\_accommodation(profile\_id UUID, accommodation\_waypoint\_id BIGINT) would use this. (This likely exists in the Module 4a spec you showed).
  + 🟠 **GIN Index on** public.profiles.roles: CREATE INDEX IF NOT EXISTS idx\_profiles\_roles\_gin ON public.profiles USING GIN (roles);
    - **Rationale**: Can improve performance of RLS policies or helper functions that frequently query this array column. Optional, but good for performance as user base and role complexity grow.
* **Changes to Existing Specs?**:
  + 🔴 **Review All RLS Policies**: All existing table RLS policies must be reviewed and updated to consistently use the public.has\_role(TEXT) helper function and auth.uid() instead of any direct (and potentially inconsistent) checks against profiles table columns from within the RLS USING clause itself, unless absolutely necessary for very simple ownership checks.
  + 🔴 user\_roles\_master **Table**: Ensure codes for accommodation\_host and regional\_content\_manager are defined in user\_roles\_master (Module 1).
  + 🔴 **Audit Field Consistency**: Ensure created\_by\_profile\_id is consistently linked to public.profiles(id) in all user-editable tables to support ownership-based RLS.

This security and authentication architecture leverages Supabase's strengths while providing a clear framework for role-based access control throughout the platform.