https://gemini.google.com/u/1/app/d423d51e9f40da78

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

* **Chosen Auth Provider**: **Supabase Auth**
  + **Rationale**: Given the project goal of using Supabase and PostgREST, Supabase Auth is the native and most tightly integrated solution. It handles user management, JWT issuance, and RLS integration seamlessly with PostgREST.
* **Authentication Flows**:  
  + **Signup**:
    1. User provides credentials (email/password, OAuth provider) to the client application.
    2. Client calls Supabase Auth signUp() endpoint.
    3. Supabase Auth creates a new user in the auth.users table.
    4. A public.handle\_new\_user() trigger/function (as per VDF Overview ) populates the corresponding public.profiles record, assigning default roles (e.g., authenticated\_pilgrim).
    5. Supabase Auth sends a confirmation email (if enabled). User confirms.
    6. Upon first successful login after confirmation, a JWT is issued.
  + **Login**:
    1. User provides credentials to the client application.
    2. Client calls Supabase Auth signInWithPassword() or OAuth signInWithOAuth().
    3. Supabase Auth verifies credentials.
    4. If successful, Supabase Auth issues a JWT (access token) and a refresh token. The access token is short-lived.
  + **Token Refresh**:
    1. Client securely stores the refresh token.
    2. When the access token expires, the client uses the refresh token to request a new access token from Supabase Auth's token endpoint (/token?grant\_type=refresh\_token).
    3. Supabase Auth validates the refresh token and issues a new access/refresh token pair.
  + **Logout**:
    1. Client calls Supabase Auth signOut().
    2. Supabase Auth invalidates the current session and associated tokens.
    3. Client discards/deletes the stored JWT and refresh token.
* **JWT Roles & Claims Injection**:  
  + Supabase Auth JWTs inherently contain sub (user UUID), aud (audience, usually 'authenticated'), exp (expiry), iat (issued at), and role (e.g., authenticated).
  + Application-specific roles (e.g., regional\_content\_manager, platform\_admin) are stored in the public.profiles.roles (TEXT array) column.
  + PostgREST RLS policies will primarily use auth.uid() to get the user's UUID and auth.role() for basic authenticated status.
  + Custom RLS helper functions (e.g., public.has\_role\_on\_profile(auth.uid(), 'regional\_content\_manager')) will query public.profiles using auth.uid() to check for specific application roles at query time. This is generally preferred for security and flexibility over embedding many custom roles directly into the JWT for PostgREST, though custom claims *can* be added to JWTs via Supabase Edge Functions or database hooks if needed for other services or more complex scenarios.

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

* **Defined Roles**:  
  1. anonymous: Unauthenticated users.
  2. authenticated\_pilgrim: Basic logged-in user (pilgrim, general user).
  3. accommodation\_host: A user who owns and manages one or more accommodation waypoints.
  4. regional\_content\_manager: Manages content (trails, waypoints, articles) for specific geographical regions.
  5. content\_moderator: Moderates user-generated content (reviews, tips, forum posts if any).
  6. platform\_admin: Super administrator with full platform access and control.
* Permission Mapping:  
  (RLS Policy Groups generally refer to the USING and WITH CHECK conditions in policies; API Scopes refer to HTTP method access on primary entities)  
    
   | Role | RLS Data Access (Examples) | API Scopes (High-Level Examples) | Typical RLS Helper Checks |  
  | :-------------------------- | :------------------------------------------------------------------------------------------------------------------------ | :--------------------------------------------------------------------------------------------------------------------------------- | :---------------------------------------------------------------------------------------- |  
  | anonymous | Read published, non-deleted content (waypoints, articles, master data with is\_active=true). | GET on public data endpoints (e.g., /waypoints, /articles, /master\_data\_types). | auth.role() = 'anon' |  
  | authenticated\_pilgrim | All anonymous access. Create/manage own profile, reviews, tips. Read own private data (e.g., saved itineraries). | All anonymous GETs. POST/PUT/DELETE on own /reviews, /user\_waypoint\_short\_tips. GET/POST/PUT/DELETE on own /profiles/me. | auth.role() = 'authenticated', auth.uid() = profile\_id |  
  | accommodation\_host | All authenticated\_pilgrim access. CRUD on own linked accommodations details and waypoint record. | PUT on /accommodations/{waypoint\_id} (own). POST/PUT/DELETE on /accommodation\_media (own). | public.has\_role\_on\_profile(auth.uid(), 'accommodation\_host'), auth.uid() = owner\_id |  
  | regional\_content\_manager | All authenticated\_pilgrim access. CRUD on content within their assigned region(s) (waypoints, segments, articles). | POST/PUT/DELETE on /waypoints, /segments, /articles (scoped to region). | public.has\_role\_on\_profile(auth.uid(), 'regional\_content\_manager'), user\_manages\_region(auth.uid(), region\_id) |  
  | content\_moderator | All authenticated\_pilgrim access. Read all user-generated content. Update moderation status of reviews, tips. | GET on all user-generated content. PUT on moderation status fields of /reviews, /user\_waypoint\_short\_tips. | public.has\_role\_on\_profile(auth.uid(), 'content\_moderator') |  
  | platform\_admin | Full CRUD access to all data across all tables, bypassing most restrictive RLS. | ALL METHODS on ALL ENDPOINTS. | public.has\_role\_on\_profile(auth.uid(), 'platform\_admin') or is\_platform\_admin(auth.uid()) |

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

* **Required Headers/Cookies Per Request Type**:  
  + **Anonymous Public Requests (e.g.,** GET /waypoints?limit=10**)**:
    - apikey: <SUPABASE\_ANON\_KEY> (Typically handled by Supabase client libraries)
  + **Authenticated Requests (e.g.,** POST /reviews**,** GET /profiles/me**)**:
    - apikey: <SUPABASE\_ANON\_KEY> (Client still identifies itself to the Supabase endpoint)
    - Authorization: Bearer <SUPABASE\_JWT\_ACCESS\_TOKEN>
  + **Token Refresh (Client to Supabase Auth Endpoint)**:
    - Typically handled by Supabase client libraries using secure HttpOnly cookies for the refresh token or other secure storage mechanisms. The direct request to /token would involve sending the refresh token in the request body.
* **Standard Security Headers (automatically managed by Supabase/browser or good practice for client to send)**:  
  + Content-Type: application/json (for POST/PUT requests with JSON body)
  + Accept: application/json (client indicating it accepts JSON responses)
  + Supabase platform itself will handle many standard security headers like X-Frame-Options, X-Content-Type-Options, Strict-Transport-Security for its services.
* **Sample** curl **with Authorization Header**:
* Bash

curl -X GET \

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

-H "apikey: <SUPABASE\_ANON\_KEY>" \

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

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

* Standard Error Object Structure (PostgREST Example):  
  PostgREST provides detailed JSON error responses. A typical structure is:
* JSON

{

"message": "Error human-readable message", // e.g., "new row violates row-level security policy for table \"your\_table\""

"details": "More specific details, often from PostgreSQL", // e.g., null

"hint": "Hint on how to solve it, if available", // e.g., null

"code": "PGRSTXXX" // PostgREST specific error code (e.g., "PGRST400", "23503" for FK violation)

}

* Custom errors raised by triggers/functions (e.g., RAISE EXCEPTION '...') will also be wrapped in a similar structure by PostgREST.
* **HTTP Status Codes Usage**:  
  + 200 OK: Successful GET, PUT, PATCH.
  + 201 Created: Successful POST.
  + 204 No Content: Successful DELETE, or PUT/PATCH that doesn't return content.
  + 400 Bad Request: Client error (e.g., malformed JSON, invalid query parameters).
  + 401 Unauthorized: Missing or invalid authentication token (JWT).
  + 403 Forbidden: Authenticated user does not have permission for the action (RLS denial).
  + 404 Not Found: Resource does not exist.
  + 409 Conflict: Resource creation conflict (e.g., unique constraint violation not caught by upsert logic).
  + 422 Unprocessable Entity: Semantically incorrect request (e.g., invalid data values failing a CHECK constraint not caught client-side).
  + 429 Too Many Requests: Rate limit exceeded.
  + 500 Internal Server Error: Unexpected server-side error (e.g., unhandled PostgreSQL error, bug in a DB function).
* **Basic Global Rate-Limit Policy (Conceptual for Supabase)**:  
  + Supabase has built-in rate limiting capabilities, configurable per project.
  + **Anonymous Users (IP-based)**: e.g., 60 requests per IP address per minute.
  + **Authenticated Users (User ID-based)**: e.g., 120 requests per user ID per minute.
  + **Specific Endpoints**: More critical or expensive endpoints might have stricter limits.
  + **Auth Endpoints**: Supabase Auth endpoints (login, signup, refresh) have their own specific, more stringent rate limits to prevent abuse.
  + **Mechanism**: Supabase typically handles this at the edge/gateway level. Exceeding limits results in an HTTP 429 response.

### **5. Impact Checklist**

* **New DB Columns, Indexes, or RLS Helpers Required?**:  
  + 🔴 public.profiles.roles **(TEXT[])**: Must exist and be reliably populated (e.g., by public.handle\_new\_user() trigger on auth.users insert). Default roles for new users (e.g., {'authenticated\_pilgrim'}) should be defined. This column was in the VDF Overview ERD and its consistent use is critical.
  + 🔴 **RLS Helper Functions (e.g.,** public.has\_role\_on\_profile(profile\_uuid UUID, role\_to\_check TEXT) RETURNS BOOLEAN**,** public.is\_platform\_admin(profile\_uuid UUID) RETURNS BOOLEAN**)**: These must be securely implemented (typically SECURITY INVOKER with careful SQL, or SECURITY DEFINER if absolutely necessary and hardened). They will read from public.profiles.roles based on auth.uid(). Their existence and consistent use across all RLS policies is crucial.
  + 🟠 public.profiles.last\_activity\_at **(TIMESTAMPTZ)**: Optional. Could be useful for tracking user activity, potentially updated by API calls or DB triggers.
  + 🟠 public.regional\_assignments **table**: Optional. If regional\_content\_manager access is complex (e.g., many-to-many regions per manager), a dedicated table linking profile\_id to region\_id might be better than trying to store regional assignments directly in profiles. RLS helper functions would then query this table.
* **Any changes to existing specs?**:  
  + 🔴 **All Table Specs with RLS**: Must be reviewed to ensure their RLS policies consistently use the defined RLS helper functions (e.g., public.has\_role\_on\_profile(auth.uid(), 'desired\_role')) and correctly reference the public.profiles.roles column for role checks, rather than relying solely on auth.role() if application-specific roles are needed.
  + 🔴 public.profiles **Table Spec**: Needs to be finalized ensuring the roles TEXT[] column is primary for application role management and handle\_new\_user() populates it correctly. The VDF Overview profiles spec is a good base.
  + 🟠 **Audit Trails for Sensitive Actions**: For critical admin actions (e.g., changing user roles, deleting major content), consider if the standard created\_by/updated\_by is sufficient or if a dedicated audit log table (beyond PostgREST's capabilities, perhaps using DB triggers) is needed for higher-level actions. (Likely out of scope for this phase but worth noting).