<https://gemini.google.com/u/1/app/a3fdff8118a11194>

Here's the platform security and authentication architecture overview:

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

**Auth Provider Choice:** **Supabase Auth** is selected. It's tightly integrated with the Supabase ecosystem (including PostgreSQL and PostgREST), provides robust JWT-based authentication, supports various login methods (email/password, OAuth), and facilitates RLS integration through custom claims.

**Authentication Flows:**

Code snippet

sequenceDiagram

participant ClientApp as Client Application

participant SupabaseAuth as Supabase Auth (GoTrue)

participant UserDB as User Database (auth.users, public.profiles)

participant JWT

%% Signup Flow

ClientApp->>SupabaseAuth: 1. Signup Request (email, password)

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

SupabaseAuth-->>ClientApp: 3. Confirmation Email Sent (if enabled) / Session (JWT)

Note over SupabaseAuth,UserDB: Optional: Trigger creates entry in public.profiles, sets default app\_metadata.roles in auth.users

%% Login Flow (Email/Password)

ClientApp->>SupabaseAuth: 4. Login Request (email, password)

SupabaseAuth->>UserDB: 5. Verify credentials

SupabaseAuth-->>ClientApp: 6. Session (Access Token JWT, Refresh Token)

Note over SupabaseAuth,JWT: Access Token JWT contains user\_id, role, app\_metadata (incl. custom roles)

%% Login Flow (OAuth - e.g., Google)

ClientApp->>SupabaseAuth: 7. Initiate OAuth (provider='google')

SupabaseAuth->>ClientApp: 8. Redirect to Google

ClientApp->>Google: 9. User authenticates with Google

Google->>SupabaseAuth: 10. Authorization Code (via redirect)

SupabaseAuth->>Google: 11. Exchange code for Google tokens

SupabaseAuth->>UserDB: 12. Create/update user in auth.users, potentially update public.profiles

SupabaseAuth-->>ClientApp: 13. Session (Access Token JWT, Refresh Token)

%% Token Refresh Flow (Simplified - handled by Supabase client libs)

ClientApp->>SupabaseAuth: 14. Request with Expired Access Token (or proactive refresh)

SupabaseAuth-->>ClientApp: 15. New Access Token JWT (using Refresh Token)

%% Logout Flow

ClientApp->>SupabaseAuth: 16. Logout Request

SupabaseAuth->>SupabaseAuth: 17. Invalidate Session/Tokens

SupabaseAuth-->>ClientApp: 18. Logout Successful

**Roles/Claims in JWT:**

* The JWT issued by Supabase Auth will contain standard claims (sub for user ID, exp, iat, aud, iss, role for Supabase's basic auth role like authenticated or anon).
* Crucially, application-specific roles (e.g., platform\_admin, regional\_content\_manager) stored in public.profiles.roles (as a TEXT[]) will be injected into the JWT as a custom claim.
* This is achieved by populating the auth.users.raw\_app\_meta\_data field with a JSON object like {"roles": ["regional\_content\_manager", "pilgrim"]}. This data is then accessible in the JWT under the app\_metadata.roles claim.
* This synchronization between public.profiles.roles and auth.users.raw\_app\_meta\_data.roles must be maintained, typically via a database trigger on public.profiles or application logic managing role assignments.

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

**Application Roles** (stored in public.profiles.roles, derived from user\_roles\_master.role\_code):

* pilgrim: Basic authenticated user with general access to view published content and manage their own contributions (e.g., reviews, tips).
* accommodation\_host: Manages their own accommodation listings.
* regional\_content\_manager: Manages content (POIs, trail segments, articles, events) specific to their assigned region(s).
* moderator: Reviews and moderates user-generated content (e.g., reviews, tips, warnings).
* admin: Manages broader platform content, users (excluding super admin functions), and potentially some configurations.
* platform\_admin: Superuser with full control over the platform, including system settings, all content, and user roles.

**Supabase Built-in Roles:**

* anon: Unauthenticated users. Can typically read published, non-restricted content.
* authenticated: Any signed-in user. Forms the baseline for users who then have more specific application roles.

**Permission Mapping (Conceptual):**

| **Role** | **Key RLS Policy Groups / DB Permissions** | **Example API Scopes (PostgREST reflects DB permissions)** |
| --- | --- | --- |
| anon | Read access to published and non-restricted content (e.g., regions, towns, articles, waypoints). | GET /regions, GET /towns, GET /articles, GET /waypoints (publicly visible ones) |
| authenticated **(pilgrim)** | Own record in profiles, user\_waypoint\_votes, user\_waypoint\_short\_tips, accommodation\_reviews. Read access like anon. | GET /profiles/me, POST /user\_waypoint\_votes, POST /accommodation\_reviews, PUT /profiles/me (own record) |
| accommodation\_host | CRUD on own accommodations records (linked to their profile). Read access like pilgrim. | POST /accommodations, PUT /accommodations/{id} (own), GET /accommodations/{id} (own details for editing) |
| regional\_content\_manager | CRUD on content within their assigned region(s) (e.g., waypoints, segments, articles). | POST /waypoints, PUT /waypoints/{id} (in own region), DELETE /articles/{id} (in own region) |
| moderator | Update content\_moderation\_status\_enum on user\_waypoint\_short\_tips, accommodation\_reviews. Read most content. | PUT /user\_waypoint\_short\_tips/{id}/status, PUT /accommodation\_reviews/{id}/status |
| admin | Broader CRUD on content tables (waypoints, articles, curated\_itineraries). Manage some user aspects. | POST /curated\_itineraries, PUT /articles/{id} (any), GET /profiles (list for admin purposes) |
| platform\_admin | Full CRUD on almost all tables, including user\_roles\_master, languages\_master. Bypasses most restrictive RLS via helper function is\_platform\_admin(). | Full access to all API endpoints exposed by PostgREST, effectively. Manage platform settings. |

*RLS policies use helper functions like auth.uid(), auth.role(), public.has\_role(TEXT), and public.is\_platform\_admin() to enforce these permissions at the database row level.*

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

**Required Headers/Cookies Per Request Type:**

* **For All API Requests to Supabase (PostgREST):**
  + apikey: The Supabase public anon key. This key grants access to the API gateway but doesn't bypass RLS or authentication checks for protected resources.
* **For Authenticated API Requests:**
  + Authorization: Bearer <Supabase\_JWT>: The JSON Web Token obtained after successful login. This is the primary mechanism for identifying the user and their permissions.
* **Client-Side Token Storage:**
  + Supabase client libraries typically handle JWT storage (e.g., in localStorage). For web applications requiring higher security for tokens (to mitigate XSS), tokens (especially refresh tokens) can be stored in HttpOnly, Secure, SameSite=Strict (or Lax) cookies, managed by a backend-for-frontend (BFF) or server-side rendering setup. PostgREST itself is stateless and expects the JWT in the Authorization header.
* **Recommended HTTP Security Headers (set by your web server/CDN/API Gateway):**
  + Strict-Transport-Security: max-age=31536000; includeSubDomains; preload1
  + Content-Security-Policy: default-src 'self'; script-src 'self' <your\_cdn\_if\_any>; ... (configure strictly)
  + X-Content-Type-Options: nosniff
  + X-Frame-Options: DENY (or SAMEORIGIN)
  + Referrer-Policy: strict-origin-when-cross-origin
  + Permissions-Policy: microphone=(), camera=() (deny unneeded features)

**Sample** curl **with Authorization Header:**

Bash

# Replace YOUR\_SUPABASE\_URL, YOUR\_ANON\_KEY, and YOUR\_JWT\_TOKEN

# Example: Fetching the current user's profile

curl -X GET \

"https://YOUR\_SUPABASE\_URL/rest/v1/profiles?select=\*&id=eq.current\_user\_id" \

-H "apikey: YOUR\_ANON\_KEY" \

-H "Authorization: Bearer YOUR\_JWT\_TOKEN" \

-H "Content-Type: application/json"

(Note: id=eq.current\_user\_id is a placeholder; typically, you'd query for auth.uid() equivalent or a /me endpoint if custom-defined).

The 0. VDF Database Overview (1).md document mentions GET /profiles/me as a conceptual API endpoint.

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

Standard Error Object:

PostgREST provides detailed JSON error responses. We will adhere to this structure, which typically includes:

JSON

{

"message": "A human-readable error message", // e.g., "permission denied for table profiles"

"details": "More specific details about the error", // e.g., "Failing row contains..." (optional)

"hint": "Suggestions for how to resolve the error", // (optional)

"code": "PGRSTXXX" // PostgREST error code (e.g., "PGRST100", "22P02" for db errors)

}

Application-level errors (if an API gateway is used) should aim for a similar structure, possibly with an application-specific error code field.

**HTTP Status Codes Usage:**

* 200 OK: Successful GET, PUT, PATCH requests.
* 201 Created: Successful POST request that creates a resource.
* 204 No Content: Successful DELETE request or POST/PUT/PATCH that doesn't return a body.
* 400 Bad Request: Client error, malformed syntax, invalid parameters, or failed data validation (e.g., incorrect data type, missing required field).
* 401 Unauthorized: Authentication failed or JWT is missing/invalid. Client should attempt to re-authenticate.
* 403 Forbidden: Authentication succeeded, but the user does not have permission to access the requested resource or perform the action (RLS denial).
* 404 Not Found: The requested resource does not exist.
* 405 Method Not Allowed: HTTP method used is not supported for the resource.
* 409 Conflict: The2 request could not be completed due to a conflict with the current state of the resource (e.g., creating a resource that already exists with a unique constraint).
* 422 Unprocessable Entity: The request was well-formed but was unable to be followed due to semantic errors (e.g., invalid data values even if types are correct). PostgREST might lean towards 400 for some of these.
* 429 Too Many Requests: The user has sent3 too many requests in a given amount of time (rate limiting).
* 500 Internal Server Error: An unexpected error occurred on the server. Client should retry later. Avoid exposing sensitive error details.

Basic Global Rate-Limit Policy:

Rate limiting will be implemented at the API gateway level (e.g., Supabase's built-in limits for specific services, or a custom gateway if used).

* **Per IP Address (for anonymous requests):** e.g., 60 requests per minute.
* **Per Authenticated User ID (for authenticated requests):** e.g., 120 requests per minute (higher limit for logged-in users).
* **Auth Endpoints:** Supabase Auth has its own rate limits for login, signup, password reset attempts to prevent abuse (e.g., max 30 email link requests per hour, 5 sign-in attempts before temporary lockout).
* Specific high-cost or sensitive endpoints might have stricter, custom rate limits.
* The Retry-After header should be sent with 429 responses where appropriate.

### **5. Impact Checklist**

* **New DB Columns, Indexes, or RLS Helpers Required?**
  + 🔴 public.profiles.roles **Synchronization to** auth.users.raw\_app\_meta\_data**:**
    - **Requirement:** A mechanism is essential to ensure the public.profiles.roles TEXT[] array is synchronized into the auth.users.raw\_app\_meta\_data field (e.g., as {"roles": [...]}). This allows Supabase Auth to inject these application-specific roles as custom claims into the JWT.
    - **Implementation:** This can be achieved via a PostgreSQL trigger on public.profiles that fires AFTER INSERT OR UPDATE of the roles column (or any relevant user data) and calls a SECURITY DEFINER function to update auth.users.raw\_app\_meta\_data for the corresponding user. Care must be taken with permissions for this function. The existing public.handle\_new\_user() function mentioned in 0. VDF Database Overview (1).md should also set initial raw\_app\_meta\_data.roles.
  + 🟠 **Standardized RLS Helper Functions:** While some helpers like public.is\_platform\_admin() were assumed during Module 3 review, ensure a comprehensive and consistently named set of RLS helper functions (e.g., public.has\_role(TEXT), public.is\_owner(UUID, record\_user\_id\_column UUID), public.is\_regional\_manager\_for\_content(content\_region\_id INT)) are defined, implemented securely (SECURITY INVOKER where possible, SECURITY DEFINER with caution), and used throughout all RLS policies. The project overview 0. VDF Database Overview (1).md mentions public.check\_profile\_roles(), public.has\_role(TEXT), and public.has\_role\_on\_profile(UUID, TEXT) which align with this.
  + No other *new* columns or indexes are immediately required for auth *beyond what's already specified in the detailed table designs* (which include profiles.roles and audit columns linking to profiles.id).
* **Any Changes to Existing Specs?**
  + 🔴 public.profiles **Table Specification:** Needs to explicitly mention the trigger and function mechanism for synchronizing its roles column with auth.users.raw\_app\_meta\_data.roles. The handle\_new\_user function (Module 1) should be reviewed to ensure it populates raw\_app\_meta\_data.roles with default roles upon profile creation.
  + 🟠 **RLS Policy Definitions (All Modules):** All RLS policies across all modules should be reviewed to ensure they consistently use the standardized helper functions and correctly reference JWT claims (e.g., auth.uid(), and roles extracted from app\_metadata.roles via a helper like current\_user\_has\_role(TEXT)).

This checklist focuses on ensuring the database schema and RLS policies fully support the described authentication and authorization model using Supabase Auth and JWT custom claims.