https://gemini.google.com/u/1/app/440838222ac514d7

Okay, I'm switching hats to Platform Security & Auth Architect. Let's define the security and authentication strategy for the Via di Francesco Pilgrimage Platform.

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

Given the project goal of using Supabase, **Supabase Auth** is the most suitable auth provider. It integrates seamlessly with Supabase's database, PostgREST, and Realtime services, and handles JWT generation and management effectively.

**Authentication Flow:**

Code snippet

sequenceDiagram

participant UserApp as User Application (Frontend)

participant SupabaseAuth as Supabase Auth

participant ProfilesDB as Profiles Table (Database)

participant PostgREST as API (PostgREST)

%% Signup Flow

UserApp->>+SupabaseAuth: 1. Signup (email, password)

SupabaseAuth-->>-UserApp: 2. Confirmation Email Sent (if enabled)

UserApp->>+SupabaseAuth: 3. (User Clicks Confirmation Link)

SupabaseAuth-->>ProfilesDB: 4. (auth.users created) -> `handle\_new\_user()` trigger fires

ProfilesDB-->>ProfilesDB: 5. `profiles` record created with default role(s)

SupabaseAuth-->>-UserApp: 6. Signup Confirmed / Initial Session (JWT)

%% Login Flow

UserApp->>+SupabaseAuth: 7. Login (email, password)

SupabaseAuth-->>-UserApp: 8. Session (Access Token JWT, Refresh Token)

Note over SupabaseAuth,ProfilesDB: JWT `roles` claim injected based on `profiles.roles`

%% Authenticated API Request

UserApp->>+PostgREST: 9. API Request with `Authorization: Bearer <JWT>`

PostgREST->>SupabaseAuth: 10. Validate JWT

SupabaseAuth-->>PostgREST: 11. JWT Valid (includes user\_id, roles)

PostgREST->>ProfilesDB: 12. Apply RLS based on user\_id and roles

ProfilesDB-->>PostgREST: 13. Data according to RLS

PostgREST-->>-UserApp: 14. API Response

%% Token Refresh Flow

UserApp->>+SupabaseAuth: 15. Refresh Token (when Access Token expires)

SupabaseAuth-->>-UserApp: 16. New Session (New Access Token JWT, New Refresh Token)

%% Logout Flow

UserApp->>+SupabaseAuth: 17. Logout

SupabaseAuth-->>-UserApp: 18. Session invalidated / Tokens cleared

**Key JWT Claims and Injection:**

* sub: User's UUID from auth.users.id.
* role: Supabase Auth sets this to authenticated by default after login.
* aud: Typically authenticated.
* exp: Expiration timestamp.
* **Custom Claims (e.g.,** app\_roles **or** user\_roles**)**: To inject the user's specific application roles (from public.profiles.roles), we would typically use a PostgreSQL function hooked into the Supabase Auth token generation process (e.g., via auth.role\_claim\_hook if available, or by customizing the JWT through database triggers or edge functions that enrich the token upon login/refresh, though Supabase's default JWT handling with PostgREST RLS often relies on querying profiles table using auth.uid() within RLS policies rather than directly injecting all roles into the JWT to keep it lean).
  + For PostgREST RLS, the primary mechanism is auth.uid() to identify the user and auth.role() (which is 'authenticated'). RLS policies then typically query the public.profiles table using auth.uid() to get the specific application roles (e.g., SELECT roles FROM public.profiles WHERE id = auth.uid()) and apply permissions accordingly. This is efficient and secure. The public.has\_role() and public.has\_role\_on\_profile() helper functions are designed for this.

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

| **Role Name** | **Inherits From** | **Key Responsibilities & RLS Policy Groups** | **Example API Scopes (Conceptual)** |
| --- | --- | --- | --- |
| anonymous | N/A | Read public content (e.g., published media, active languages/roles, public profiles via view, published trail/waypoint data). Uses RLS policies with USING (true) for generally accessible data. | read:public\_languages, read:public\_roles, read:public\_media, read:public\_profiles\_view, read:public\_trails, read:public\_waypoints |
| pilgrim\_user | anonymous | Read own profile, update own profile, create user-generated content (tips, votes), read all published content. Access controlled by auth.uid() = id or auth.uid() = user\_id in RLS. | profile:read\_own, profile:update\_own, tips:create, votes:create, media:upload\_own |
| accommodation\_host | pilgrim\_user | Manage own accommodation listings (CRUD). | accommodations:create\_own, accommodations:read\_own, accommodations:update\_own, accommodations:delete\_own |
| content\_moderator | pilgrim\_user | Review and moderate user-generated content (tips, potentially media based on status), update media.media\_status. | tips:moderate, media:update\_status, media:update\_any |
| regional\_content\_manager | content\_moderator | Manage content (trails, POIs, segments, warnings) within assigned regions. Requires RLS helper like is\_regional\_manager\_for\_trail(trail\_id). | trails:update\_regional, waypoints:create\_regional, warnings:manage\_regional |
| admin\_platform | regional\_content\_manager | Manage most platform data (users, roles, all content), except super-admin functions. Uses public.has\_role('admin\_platform') in RLS. | users:manage\_all, roles:manage\_all, languages:manage\_all, content:manage\_all, media:delete\_any |
| admin\_super | admin\_platform | Full system access, manage core schema, critical configurations, is\_system\_role definitions. Often used for migrations or direct DB tasks rather than API scopes. Uses public.has\_role('admin\_super') in RLS. | system:manage\_all, (Potentially bypasses some application-level checks for critical maintenance via direct DB access if needed, though API access would be same as admin\_platform or more privileged). |

*Note: API scopes are conceptual and would be enforced by a combination of RLS (for data access) and potentially at an API gateway level or backend logic if specific business actions beyond CRUD are needed.*

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

For a Supabase/PostgREST setup, the primary mechanism is the JWT Bearer token.

* **Required Headers (for authenticated requests)**:
  + Authorization: Bearer <YOUR\_SUPABASE\_JWT>: Essential for authenticating the user and enabling RLS.
  + apikey: <YOUR\_SUPABASE\_ANON\_KEY>: Supabase's public/anonymous key, required for all requests to PostgREST.
  + Content-Type: application/json (For POST, PUT, PATCH requests).
  + Accept: application/json (Typically).
* **Security Best Practice Headers (set by the server/PostgREST/Supabase Edge Functions delivering HTML or API responses)**:
  + Strict-Transport-Security: max-age=31536000; includeSubDomains; preload
  + X-Content-Type-Options: nosniff
  + X-Frame-Options: DENY (or SAMEORIGIN depending on needs)
  + Content-Security-Policy: default-src 'self'; script-src 'self' <trusted\_cdn\_if\_any>; style-src 'self' 'unsafe-inline' <trusted\_cdn\_if\_any>; img-src 'self' data: <your\_supabase\_storage\_url>; connect-src 'self' <your\_supabase\_api\_url>; frame-ancestors 'none'; (This is an example, needs to be tailored).
  + Referrer-Policy: strict-origin-when-cross-origin
  + Permissions-Policy: accelerometer=(), camera=(), geolocation=(), gyroscope=(), magnetometer=(), microphone=(), payment=(), usb=() (Restrict features not used).
* **Cookies**:
  + Supabase Auth uses cookies (e.g., sb-access-token, sb-refresh-token) to manage sessions, primarily for server-side rendering or when its client library is used in a browser context. These are typically HttpOnly and Secure.
  + For SPA frontends making direct API calls, the JWT is usually retrieved from local storage/session storage (or secure in-memory store) by the Supabase client library and sent in the Authorization header. PostgREST itself is stateless and relies on the JWT.

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

Bash

curl -X GET \

'https://<your-project-ref>.supabase.co/rest/v1/profiles?select=\*&id=eq.<user\_uuid>' \

-H 'apikey: <YOUR\_SUPABASE\_ANON\_KEY>' \

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

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

**Standard Error Object:**

JSON

{

"code": "ERROR\_CODE\_SLUG", // e.g., "validation\_failed", "resource\_not\_found", "unauthenticated"

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

"detail": "More specific details or field-specific errors (optional, can be an object or string)."

// "hint": "Optional hint on how to resolve the error." // PostgREST uses this

// "details": "PostgREST often uses this for more structured error info."

}

PostgREST already provides a detailed JSON error response structure which includes message, code, details, and hint. We should align with or extend this.

**HTTP Status Codes:**

* 200 OK: Successful GET, PUT, PATCH.
* 201 Created: Successful POST.
* 204 No Content: Successful DELETE, or successful PUT/PATCH if no content is returned.
* 400 Bad Request: General client-side error (e.g., malformed request, invalid JSON).
* 401 Unauthorized: Authentication required or failed (e.g., missing/invalid JWT).
* 403 Forbidden: Authenticated user does not have permission for the action (RLS denial).
* 404 Not Found: Resource does not exist.
* 409 Conflict: Request conflicts with the current state of the resource (e.g., unique constraint violation not caught by pre-flight checks).
* 422 Unprocessable Entity: Semantic errors in the request payload (e.g., validation failed for specific fields, even if syntactically correct).
* 429 Too Many Requests: Rate limit exceeded.
* 500 Internal Server Error: Unexpected server-side error.

**Basic Global Rate-Limit Policy (Conceptual - Supabase has built-in limits, custom via Edge Functions/API Gateway if needed):**

* **Per IP Address or User ID**: Apply limits based on the source IP for anonymous requests and user ID for authenticated requests.
* **General Limit**: e.g., 100-200 requests per minute per user/IP.
* **Stricter Limits for Sensitive/Expensive Operations**: e.g., Login attempts (5-10 per minute), new user registration, resource-intensive queries.
* **Response**: 429 Too Many Requests with Retry-After header if applicable.
* Supabase's built-in abuse protection for Auth endpoints and general platform limits should be leveraged first. More granular control can be implemented with an API gateway or custom logic in Edge Functions if specific endpoints become targets.

### **5. Impact Checklist**

Based on this security and auth architecture using Supabase Auth and PostgREST RLS:

* **New DB Columns, Indexes, or RLS Helpers Required?**
  + 🔴 **None critical that aren't already addressed.** The existing table specs (as of Versions 2.1/2.2 after re-evaluation) which include standard audit columns (created\_by\_profile\_id, updated\_by\_profile\_id), icon\_identifier fields, and robust RLS policies supported by helper functions like public.has\_role() and public.has\_role\_on\_profile(), are well-aligned.
  + 🟠 **Optional**: If a more granular role/permission system beyond the profiles.roles text array were ever needed (e.g., specific permissions assigned per user, not just roles), that would require new tables. However, for the current role-based model, the existing structure is sound.
  + The helper functions public.has\_role(TEXT) and public.has\_role\_on\_profile(UUID, TEXT) must be robustly implemented as they are central to many RLS policies. Their definitions were included in the Module Overview.
* **Any Changes to Existing Specs?**
  + 🔴 **No "Must-add" changes** to the *database schema specifications* themselves are identified solely from this auth architecture phase, assuming the latest versions (2.1/2.2) of the table specs are the baseline.
  + 🟠 **Clarification in RLS Policy Population of Audit Fields**: While the RLS policies define *who can update rows*, the specs (or dev guidelines) should note that the updated\_by\_profile\_id (and created\_by\_profile\_id where not implicit) columns in tables like languages\_master, user\_roles\_master, media, translations, and profiles (for admin updates) will primarily be set by **application-layer logic** when an authenticated user (especially an admin or moderator) performs an action. The generic extensions.moddatetime trigger only handles updated\_at. If direct DB manipulation by admins is common and needs to capture their ID automatically via triggers, a more complex BEFORE UPDATE trigger strategy would be needed for these audit fields, potentially replacing the simple extensions.moddatetime for updated\_at with a custom one that also sets updated\_by\_profile\_id = auth.uid(). This is a development consideration for how these fields get populated. The current schema supports their existence.
  + 🟢 **Refinement of JWT Claims**: While not a schema change, the precise mechanism and extent of custom claims (like app\_roles) injected into the JWT versus relying purely on DB lookups within RLS policies is an implementation detail. The current approach (lean JWT, RLS queries profiles table) is robust and scalable.