Okay, I'm ready to define the security and authentication architecture.

### [**https://gemini.google.com/u/1/app/0e3ccfac5f6b71bb**](https://gemini.google.com/u/1/app/0e3ccfac5f6b71bb)

### **Auth Flow Diagram**

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

The authentication flow is as follows:

Code snippet

sequenceDiagram

participant UserApp as User Application (Frontend)

participant SupabaseAuth as Supabase Auth

participant Database as Pilgrimage Platform DB (PostgreSQL)

%% Signup Flow

UserApp->>+SupabaseAuth: 1. Signup (email, password, any raw\_user\_meta\_data)

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

UserApp->>+SupabaseAuth: 3. (User Clicks Confirmation Link if step 2 occurs)

SupabaseAuth-->>Database: 4. auth.users record created

Database-->>Database: 5. `handle\_new\_user()` trigger fires: <br/> - Creates public.profiles record <br/> - Sets initial roles in `profiles.roles` <br/> - Synchronizes roles to `auth.users.raw\_app\_meta\_data.roles` <br/> - Sets initial `last\_activity\_at`

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

%% Login Flow

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

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

Note over SupabaseAuth,Database: JWT `roles` claim is populated from `auth.users.raw\_app\_meta\_data.roles`. [cite: 556, 723, 906]

%% Authenticated API Request (e.g., to PostgREST)

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

Database->>Database: 10. PostgREST validates JWT with Supabase Auth & extracts user\_id, roles

Database->>Database: 11. RLS policies applied based on user\_id and roles (e.g., using `auth.uid()`, `auth.jwt()->>'roles'`)

Database-->>-UserApp: 12. API Response (data according to RLS)

%% Token Refresh Flow

UserApp->>+SupabaseAuth: 13. Refresh Token (when Access Token expires)

SupabaseAuth-->>-UserApp: 14. New Session (New Access Token JWT, New Refresh Token)

%% Logout Flow

UserApp->>+SupabaseAuth: 15. Logout

SupabaseAuth-->>-UserApp: 16. Session invalidated / Tokens cleared locally

**Key JWT Claims and Role Injection**:

* **Standard Claims**: sub (User's UUID from auth.users.id), aud (authenticated), exp (expiration timestamp), iat, etc.
* **Custom Claims / Roles**: Application-specific roles stored in public.profiles.roles are synchronized to auth.users.raw\_app\_meta\_data.roles by the handle\_new\_user and sync\_profile\_roles\_to\_auth\_user database triggers. Supabase Auth can then include these roles directly within the JWT from the raw\_app\_meta\_data.roles field. This makes roles available to PostgREST for RLS policies without requiring an additional lookup to the profiles table just for roles, though policies will still use auth.uid() to get the current user's ID for ownership checks.

### **Roles & Permission Matrix**

The roles are defined in user\_roles\_master. The matrix below outlines key responsibilities and conceptual API scopes, largely based on the existing security specification.

| **Role Name (Code)** | **Inherits From (Conceptually)** | **Key Responsibilities & RLS Policy Focus** | **Example API Scopes (Conceptual)** |
| --- | --- | --- | --- |
| anonymous | N/A | Read public content (published waypoints, active languages, public profiles, etc.). RLS: USING (entity.status = 'published' AND entity.deleted\_at IS NULL). | read:public\_languages, read:public\_event\_types, read:public\_waypoints, read:public\_articles |
| pilgrim\_user | anonymous | Read/update own profile. Create user-generated content (reviews, tips, votes). Access controlled by auth.uid() = owner\_id. | profile:read\_own, profile:update\_own, reviews:create\_own, tips:create\_own, media:upload\_own |
| accommodation\_host | pilgrim\_user | Manage own accommodation listings (CRUD operations, if accommodations are separate entities linked to waypoints and owned by hosts). RLS: auth.uid() = accommodation.host\_profile\_id. | accommodations:create\_own, accommodations:read\_own, accommodations:update\_own, accommodations:delete\_own |
| content\_moderator | pilgrim\_user | Review and moderate user-generated content (tips, reviews). Update media status. RLS: Check for content\_moderator role. | tips:moderate, reviews:moderate, media:update\_status |
| regional\_content\_manager | content\_moderator | Manage content (trails, waypoints, events, articles) within assigned regions. RLS: Helper like is\_regional\_manager\_for\_entity(entity\_id). | waypoints:update\_regional, events:create\_regional, articles:manage\_regional |
| admin\_platform | regional\_content\_manager | Manage most platform data (users, roles, all content). RLS: public.has\_role\_on\_profile(auth.uid(), 'admin\_platform'). | users:manage\_all, roles:manage\_all, master\_data:manage\_all, content:manage\_all\_regions |
| admin\_super | admin\_platform | Full system access, schema changes, critical configurations. RLS: public.has\_role\_on\_profile(auth.uid(), 'admin\_super'). Usually direct DB/migrations. | system:manage\_all (Primarily DB level, API access similar to admin\_platform but could have bypasses for specific maintenance tasks if designed). |

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

### **Security Headers & Cookies**

For the Supabase and PostgREST setup, security relies heavily on JWTs.

**Required Headers (for authenticated API requests to PostgREST)**:

* Authorization: Bearer <YOUR\_SUPABASE\_JWT>: Essential for authenticating the user.
* apikey: <YOUR\_SUPABASE\_ANON\_KEY>: Supabase's public/anonymous key, required for all requests.
* Content-Type: application/json (For POST, PUT, PATCH requests with a JSON body).
* Accept: application/json (Typically for clients expecting JSON responses).

**Security Best Practice Headers (to be set by the server/hosting platform/edge functions)**:

* Strict-Transport-Security: max-age=31536000; includeSubDomains; preload
* X-Content-Type-Options: nosniff
* X-Frame-Options: DENY (or SAMEORIGIN if embedding is needed)
* Content-Security-Policy: A restrictive policy tailored to the application, e.g., default-src 'self'; img-src 'self' <your\_supabase\_storage\_url>; script-src 'self'; style-src 'self' 'unsafe-inline'; connect-src 'self' <your\_supabase\_api\_url>; frame-ancestors 'none'; (This is an example and needs careful tailoring).
* Referrer-Policy: strict-origin-when-cross-origin
* Permissions-Policy: Restrict browser features not used, e.g., geolocation=(), microphone=().

**Cookies**:

* Supabase Auth client libraries may use cookies (e.g., sb-access-token, sb-refresh-token) to manage sessions, especially for server-side rendering or when its JavaScript library is used in a browser context where it handles token storage.
* These cookies are typically set as HttpOnly and Secure by Supabase.
* For SPAs making direct API calls, the JWT is usually retrieved from localStorage or sessionStorage (or a secure in-memory store) by the Supabase client library and sent in the Authorization header. PostgREST itself is stateless.

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

Bash

curl -X GET \

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

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

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

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

Standard Error Object:

PostgREST provides detailed JSON error responses. We should align with this structure:

JSON

{

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

"code": "PGRSTXXX", // PostgREST error code, or a custom application error code

"details": "More specific details or structural information about the error (e.g., constraint name).", // [cite: 695]

"hint": "Optional hint on how to resolve the error." // [cite: 694]

}

**HTTP Status Codes**:

* 200 OK: Successful GET, PUT, PATCH.
* 201 Created: Successful POST.
* 204 No Content: Successful DELETE, or PUT/PATCH if no content is returned.
* 400 Bad Request: 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).
* 422 Unprocessable Entity: Semantic errors in the request payload (e.g., validation failed for specific fields).
* 429 Too Many Requests: Rate limit exceeded.
* 500 Internal Server Error: Unexpected server-side error.

**Basic Global Rate-Limit Policy**:

* Leverage Supabase's built-in abuse protection for Auth endpoints and general platform request limits first.
* **Per IP Address / User ID**: Apply limits based on source IP for anonymous requests and User ID for authenticated requests.
* **General API Limit**: e.g., 100-300 requests per minute per user/IP.
* **Stricter Limits for Sensitive/Expensive Operations**:
  + Login/Signup attempts: e.g., 5-10 per minute.
  + Resource-intensive queries or mutations: Define specific, tighter limits.
* **Response**: 429 Too Many Requests HTTP status code, potentially with a Retry-After header.
* **Customization**: For more granular control beyond Supabase's defaults, custom rate limiting can be implemented using Supabase Edge Functions or an API gateway if necessary.

### **Impact Checklist**

* **Any new DB columns, indexes, or RLS helpers required?**
  + 🔴 **None critical that aren't already addressed by the V2.1 database specifications.** The existing design, which includes:
    - public.profiles.roles TEXT[] column synchronized with auth.users.raw\_app\_meta\_data.roles.
    - Database triggers public.handle\_new\_user() and public.sync\_profile\_roles\_to\_auth\_user() for profile creation and role synchronization.
    - RLS helper functions like public.has\_role(TEXT) and public.has\_role\_on\_profile(UUID, TEXT) (whose robust implementation is assumed as per Module 1 overview ).
    - Standard audit columns (created\_by\_profile\_id, updated\_by\_profile\_id) on relevant tables. These elements are fundamental to this authentication architecture and are presumed to be part of the finalized V2.1 database specs.
* **Any changes to existing specs?**
  + 🔴 **No "Must-add" changes** to the database schema specifications themselves (assuming the V2.1/V2.2/V2.3 versions of table specs incorporating audit columns, role sync, etc., are indeed the baseline).
  + 🟠 **Clarification on Audit Field Population**: While schemas define audit columns like updated\_by\_profile\_id, the existing documentation correctly notes that their population during operations by administrators or moderators is primarily an **application-layer or specific trigger logic responsibility**. The standard extensions.moddatetime trigger only handles updated\_at. This isn't a schema change but a development best practice to ensure these fields are correctly populated via Supabase client calls or custom functions that pass the acting user's ID.
  + 🟢 **Review RLS Helper Function Security**: Ensure all SECURITY DEFINER helper functions (like those for role checking or profile creation) have their search\_path explicitly set and follow security best practices to prevent privilege escalation. This is more of an implementation detail review than a schema change.
  + 🟢 **JWT Claim Strategy**: The current strategy is to include roles from auth.users.raw\_app\_meta\_data.roles in the JWT. If the JWT size becomes an issue with many roles, an alternative (leaner JWT, more DB lookups in RLS) could be considered later, but the current approach is robust for typical scenarios.