<https://gemini.google.com/u/1/app/b90970622259e225>

Okay, I am ready to outline the security and authentication architecture.

### 1\. Auth Flow Diagram

- Auth Provider: Supabase Auth is the most suitable auth provider for this project. It integrates seamlessly with Supabase's database, PostgREST, and Realtime services, handling JWT generation and management effectively.

- Authentication Flow:

Code snippet

```

sequenceDiagram

participant UserApp as User Application (Frontend)

participant SupabaseAuth as Supabase Auth

participant Database as Platform Database (inc. Profiles Table)

participant API as API (PostgREST / GraphQL)

%% Signup Flow

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

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

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

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

Database-->>Database: 5. `handle\_new\_user()` trigger fires (creates public.profiles record with default role(s); syncs roles to auth.users.raw\_app\_meta\_data.roles)

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,Database: JWT `roles` claim (e.g., in raw\_app\_meta\_data) populated based on `public.profiles.roles` [cite: 671]

%% Authenticated API Request

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

API->>SupabaseAuth: 10. Validate JWT (delegated to Supabase PostgREST/Auth layer)

SupabaseAuth-->>API: 11. JWT Valid (provides user\_id, roles claims to API execution context)

API->>Database: 12. Query Database (RLS policies applied based on user\_id and roles from JWT/session)

Database-->>API: 13. Data according to RLS

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

%% Token Refresh Flow

UserApp->>+SupabaseAuth: 15. Request with 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 locally

```

- JWT Claims and Injection:

- Standard claims include `sub` (User's UUID from `auth.users.id`), `aud` (typically 'authenticated'), `exp` (expiration timestamp), and `role` (Supabase Auth sets this to 'authenticated' by default after login).

- Custom application roles (from `public.profiles.roles`) are synchronized to `auth.users.raw\_app\_meta\_data.roles` via database triggers (`handle\_new\_user` and `sync\_profile\_roles\_to\_auth\_user`). Supabase can then be configured to include these `raw\_app\_meta\_data.roles` in the JWT.

- RLS policies primarily use `auth.uid()` to identify the user and can access these roles from the JWT or by querying the `public.profiles` table using `auth.uid()`, for which helper functions like `public.has\_role(TEXT)` are designed.

\* \* \* \* \*

### 2\. Roles & Permission Matrix

The roles are based on the detailed list from the security specification document.

| 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, reviews), 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`, `reviews: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, reviews, potentially media based on status), update `media.media\_status`. | `tips:moderate`, `reviews: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\_entity(entity\_id)`. | `trails:update\_regional`, `waypoints:create\_regional`, `warnings:manage\_regional`, `articles:create\_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 for migrations/direct DB tasks. 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). |

\*Note: API scopes are conceptual and enforced by RLS for data access, and potentially at an API gateway or backend logic for specific business actions beyond CRUD.\*

\* \* \* \* \*

### 3\. Security Headers & Cookies

- Required Headers (for authenticated requests to PostgREST):

- `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 its PostgREST API.

- `Content-Type: application/json` (For `POST`, `PUT`, `PATCH` requests).

- `Accept: application/json` (Typically requested by clients).

- Security Best Practice Headers (to be set by the server/hosting provider):

- `Strict-Transport-Security: max-age=31536000; includeSubDomains; preload`

- `X-Content-Type-Options: nosniff`

- `X-Frame-Options: DENY` (or `SAMEORIGIN` as needed)

- `Content-Security-Policy`: (Example, needs tailoring) `default-src 'self'; script-src 'self'; style-src 'self' 'unsafe-inline'; img-src 'self' data: <your\_supabase\_storage\_url>; connect-src 'self' <your\_supabase\_api\_url>; frame-ancestors 'none';`

- `Referrer-Policy: strict-origin-when-cross-origin`

- `Permissions-Policy`: (Restrict features not used) `accelerometer=(), camera=(), geolocation=(), gyroscope=(), magnetometer=(), microphone=(), payment=(), usb=()`

- Cookies:

- Supabase Auth uses cookies (e.g., `sb-access-token`, `sb-refresh-token`) to manage sessions, typically `HttpOnly` and `Secure`, primarily for server-side rendering or when its client library is used in a browser context.

- For SPAs making direct API calls, the JWT is usually retrieved from local storage/session storage (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=\*&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: PostgREST provides a detailed JSON error response structure, which we should align with or extend if using a custom API gateway. A typical structure includes:

JSON

```

{

"message": "A human-readable summary of the error.", // PostgREST provides this

"code": "ERROR\_CODE\_SLUG", // PostgREST provides this (e.g., PGRST116 for RLS violation)

"details": "More specific details or field-specific errors (PostgREST often uses this).", // [cite: 690, 691]

"hint": "Optional hint on how to resolve the error (PostgREST provides this)." // [cite: 690]

}

```

- HTTP Status Codes:

- `200 OK`: Successful `GET`, `PUT`, `PATCH`.

- `201 Created`: Successful `POST`.

- `204 No Content`: Successful `DELETE`, or `PUT`/`PATCH` with no content returned.

- `400 Bad Request`: Client-side error (e.g., malformed JSON, invalid parameters).

- `401 Unauthorized`: Authentication required or failed (missing/invalid JWT).

- `403 Forbidden`: Authenticated user lacks permission (RLS denial).

- `404 Not Found`: Resource does not exist.

- `409 Conflict`: Request conflicts with resource state (e.g., unique constraint violation).

- `422 Unprocessable Entity`: Semantic errors in payload (e.g., validation failed).

- `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 (for anonymous requests) or User ID (for authenticated requests).

- General Limit: e.g., 100-200 requests per minute per user/IP (conceptual).

- Stricter Limits for Sensitive/Expensive Operations: e.g., Login attempts (5-10 per minute), new user registration.

- Response for Exceeded Limit: `429 Too Many Requests`, potentially with a `Retry-After` header.

- More granular control can be implemented via a dedicated API gateway or custom logic in Supabase Edge Functions if needed.

\* \* \* \* \*

### 5\. Impact Checklist

- Any new DB columns, indexes, or RLS helpers required?

- 🔴 None critical that aren't already addressed by V2.1/V2.2/V2.3 table specs. The existing database schema specifications, which include standard audit columns, `profiles.roles` for application roles, and robust RLS policies supported by helper functions like `public.has\_role(TEXT)` and `public.has\_role\_on\_profile(UUID, TEXT)` (defined in User & Content Infrastructure Module ), are well-aligned with this security architecture.

- The RLS helper functions (`public.has\_role(TEXT)`, `public.has\_role\_on\_profile(UUID, TEXT)`, `is\_platform\_admin()`, etc.) must be robustly implemented and secured (e.g., `SECURITY DEFINER` functions need careful `search\_path` hardening and minimal necessary privileges). These are already planned.

- 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 reviewed versions (V2.1/V2.2/V2.3) of the table specs are the baseline.

- 🟠 Clarification on Audit Field Population: It's a development consideration (rather than a schema change) that `created\_by\_profile\_id` and `updated\_by\_profile\_id` are primarily set by application-layer logic passed to the database, especially for admin/moderator actions. Triggers can assist, but the originating user ID for an action often comes from the application context handling the API request.

- 🟢 Refinement of JWT Claims Strategy (Implementation Detail): The strategy of synchronizing `public.profiles.roles` to `auth.users.raw\_app\_meta\_data.roles` for inclusion in the JWT, and then using `auth.uid()` and these JWT roles (or direct profile lookups) within RLS policies, is robust. No schema change, but an important implementation detail to ensure is correctly configured in Supabase.