<https://gemini.google.com/u/1/app/21b8e85b3e611787>

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

* **Chosen Auth Provider**: **Supabase Auth**
  + **Rationale**: Supabase Auth is tightly integrated with the Supabase ecosystem (PostgreSQL, PostgREST, Storage), providing JWT-based authentication, RLS integration, OAuth providers, and secure user management out-of-the-box. This aligns perfectly with the project's backend choice.
* **Authentication Flows**:
* Code snippet

sequenceDiagram

actor ClientApp

participant SupabaseAuth as Supabase Auth UI / SDK

participant SupabaseAPI as Supabase Auth Backend

participant Database

ClientApp->>SupabaseAuth: 1. Signup (Email/Pass or OAuth)

SupabaseAuth->>SupabaseAPI: Forwards credentials/token

SupabaseAPI->>SupabaseAPI: Creates user in auth.users

SupabaseAPI-->>Database: auth.users new entry

Database-->>Database: Trigger: public.handle\_new\_user()

Note right of Database: Populates public.profiles,<br>assigns default role(s) (e.g., ['pilgrim']) [cite: 2023]

SupabaseAPI-->>ClientApp: Returns JWT (Access & Refresh Tokens) & User Session

ClientApp->>SupabaseAuth: 2. Login (Email/Pass or OAuth)

SupabaseAuth->>SupabaseAPI: Validates credentials/token

SupabaseAPI-->>ClientApp: Returns JWT (Access & Refresh Tokens) & User Session

ClientApp->>SupabaseAPI: 3. API Request with Access Token

Note right of SupabaseAPI: (e.g., to PostgREST)

SupabaseAPI->>Database: Executes query under user's role/permissions (RLS)

ClientApp->>SupabaseAPI: 4. Token Refresh (using Refresh Token)

Note left of ClientApp: When Access Token nears expiry

SupabaseAPI-->>ClientApp: Returns new JWT (Access & Refresh Tokens)

ClientApp->>SupabaseAuth: 5. Logout

SupabaseAuth->>SupabaseAPI: Invalidates session/tokens

SupabaseAPI-->>ClientApp: Confirmation

Note left of ClientApp: Client discards tokens

* **JWT Roles/Claims Injection**:  
  + Supabase Auth JWTs automatically include standard claims: sub (user UUID from auth.users.id), role (PostgreSQL role, typically authenticated for logged-in users by default), iat, exp, aud.
  + **Application-Specific Roles**: The public.profiles.roles (TEXT ARRAY) field stores granular application roles (e.g., pilgrim, host, moderator). To make these available for RLS and API logic:
    - **Option A (Preferred - Custom Claims)**: Use a Supabase Edge Function (triggered on token generation or manually) or a database trigger to add custom claims to the JWT. This claim (e.g., app\_roles: ['pilgrim', 'host']) can then be accessed by RLS helper functions.
    - **Option B (Simpler RLS, more DB lookups)**: RLS helper functions like public.has\_role\_on\_profile(auth.uid(), 'some\_role') will query public.profiles using auth.uid() at query time. This is already the pattern seen in some existing RLS policies.
  + PostgREST uses the role claim in the JWT to switch the PostgreSQL transaction to that role, enabling RLS. If application roles are more granular than what auth.users.role can store (it's a single TEXT field), custom claims or direct profile lookups in RLS are essential.

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

* **Defined Roles**:  
  + anon (Anonymous): Unauthenticated users. Implicit PostgreSQL role used by PostgREST for unauthenticated requests.
  + authenticated: Standard Supabase/PostgreSQL role for any logged-in user. All users in auth.users typically have this role.
  + **Application Roles** (managed in public.profiles.roles TEXT ARRAY ):
    - pilgrim: Default role for registered users. Can manage their profile, create reviews, tips, votes.
    - host: Accommodation provider. Can manage their own accommodation listings (CRUD on accommodations and linked tables where host\_profile\_id matches).
    - moderator: Can manage user-generated content (reviews, tips), update moderation statuses.
    - regional\_content\_manager (or regional\_manager): Can manage content (trails, waypoints, articles) within assigned regions.
    - admin\_platform (or platform\_admin): Broader content and user management capabilities across the platform, excluding direct database/infrastructure changes.
    - admin\_super: (Conceptual Supabase super admin) Full access, typically not directly mapped to application users but used for migrations, service\_role keys.
* **Permission Mapping (Conceptual - to be detailed per table in RLS):** | Role | RLS Policy Group (Examples) | API Scope Examples (Conceptual) |  
  | :------------------------- | :---------------------------------------------------------------- | :----------------------------------------------------------------------------------------------------------------------------- |  
  | anon | Read public, active data (e.g., waypoints where published, master data where is\_active) | GET /accommodations, GET /waypoints, GET /articles (published) |  
  | pilgrim (authenticated) | Read all public data, Create own profile data, Create own UGC (reviews, tips, votes), Update/Delete own UGC (pre-moderation) | All anon scopes + POST /reviews, PUT /profiles/me, DELETE /reviews/{id} (own) |  
  | host (authenticated) | pilgrim permissions + CRUD on own accommodations & linked entities (amenities, rooms) where host\_profile\_id = auth.uid() | pilgrim scopes + PUT /accommodations/{id} (own), POST /accommodations/{id}/rooms (own) |  
  | moderator (authenticated)| pilgrim permissions + Update moderation\_status on UGC (reviews, tips), Soft-delete any UGC | pilgrim scopes + PATCH /reviews/{id}/status, DELETE /tips/{id} (any) |  
  | regional\_content\_manager | pilgrim permissions + CRUD on content within assigned regions (e.g., trails, waypoints, articles via FK checks or helper fn) | pilgrim scopes + POST /trails, PUT /waypoints/{id} (region-scoped) |  
  | admin\_platform | Broad CRUD on most platform content, manage users (excluding auth.users directly), manage master data definitions. | All regional\_content\_manager scopes (unscoped) + PUT /users/{id}/roles, POST /amenities\_master |  
  | admin\_super | Full DB access (via service\_role key or direct DB connection). Not a typical API user role. | N/A (Operates at DB/infrastructure level) |  
    
   *RLS policies for each table will use helper functions like public.has\_role\_on\_profile(auth.uid(), 'role\_code') or check custom JWT claims to enforce these permissions.*

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

* **Required Headers per Request Type**:  
  + **All API Requests (to PostgREST/GraphQL endpoints)**:
    - apikey: <SUPABASE\_ANON\_KEY>: Supabase anonymous key, allows the request to reach the Supabase backend.
  + **Authenticated API Requests**:
    - Authorization: Bearer <JWT\_ACCESS\_TOKEN>: The JWT access token obtained after login.
  + **Data Modification Requests (POST, PUT, PATCH)**:
    - Content-Type: application/json (or application/graphql+json for GraphQL).
    - (Supabase may also require Prefer: return=representation if the client expects the modified resource back).
  + **General Security Headers (configured at Supabase/CDN/Webserver level)**:
    - Content-Security-Policy
    - Strict-Transport-Security
    - X-Content-Type-Options1
    - X-Frame-Options
    - X-XSS-Protection
    - Referrer-Policy
* **Cookies**:  
  + Supabase Auth primarily uses cookies for server-side auth flows or to store refresh tokens securely (HttpOnly).
  + For SPA/mobile clients interacting with the API, the JWT access token is typically sent in the Authorization header. The client is responsible for securely storing and managing tokens.
* **Sample** curl **with Authorization Header**:
* Bash

curl -X GET \

"https://<YOUR\_PROJECT\_REF>.supabase.co/rest/v1/accommodations?limit=1" \

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

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

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

* Standard Error Object Structure:  
  When an error occurs, API responses (especially for 4xx/5xx statuses) should ideally conform to a consistent JSON structure. PostgREST provides a default structure, which includes message, code, details, hint.
* JSON

{

"code": "PGRST\_ERROR\_CODE\_OR\_APP\_CODE", // e.g., "22P02" for invalid input, or custom app code

"message": "A user-friendly error message explaining what went wrong.",

"details": "Optional: More specific technical details or validation errors for particular fields.",

"hint": "Optional: A suggestion on how to fix the error."

}

* Application-level errors (e.g., from custom Edge Functions if used as an API layer) should strive for a similar structure.
* **HTTP Status Codes**:  
  + 200 OK: Successful GET, PUT, PATCH.
  + 201 Created: Successful POST.
  + 204 No Content: Successful DELETE, or PUT/PATCH where no content is returned.
  + 400 Bad Request: Invalid syntax in request (e.g., malformed JSON, invalid query parameter format).
  + 401 Unauthorized: Missing, invalid, or expired authentication token.
  + 403 Forbidden: Authenticated user lacks permission for the requested resource/action (RLS denial).
  + 404 Not Found: Requested resource does not exist.
  + 409 Conflict: Resource already exists (e.g., unique constraint violation during POST).
  + 422 Unprocessable Entity: Request was well-formed but semantically incorrect (e.g., business rule violation not caught by DB constraints).
  + 429 Too Many Requests: Rate limit exceeded.
  + 500 Internal Server Error: Unhandled server-side exception.
* **Basic Global Rate-Limit Policy (Conceptual - Supabase Config)**:  
  + **Anonymous Users**: e.g., 60 requests per minute per IP address.
  + **Authenticated Users (**pilgrim**,** host**)**: e.g., 120-180 requests per minute per user ID.
  + **Privileged Roles (**moderator**,** regional\_manager**,** admin\_platform**)**: Higher limits, or specific limits for bulk operations if any.
  + Supabase's built-in abuse detection and rate limiting features should be leveraged. Fine-grained limits per endpoint can be implemented via an API gateway or custom logic if needed beyond Supabase's default capabilities.

### **5. Impact Checklist**

* **New DB Columns, Indexes, or RLS Helpers Required?**:  
  + ⚪ auth.users.role **vs** public.profiles.roles **Synchronization**: If PostgREST's role impersonation based on auth.users.role is to be used extensively for very distinct PostgreSQL roles per application role, a robust mechanism (trigger or edge function) to keep auth.users.role (TEXT) in sync with the primary role from public.profiles.roles (TEXT ARRAY) might be needed.
    - **Alternatively (and often preferred for flexibility with multiple roles)**: Rely on custom JWT claims populated with app\_roles from public.profiles.roles. RLS policies would then check these custom claims (e.g., auth.jwt() ->> 'app\_roles' @> '{"moderator"}'::jsonb) instead of or in addition to auth.role(). This is generally more flexible than trying to map multiple application roles to a single auth.users.role field. The existing public.has\_role\_on\_profile(UUID, TEXT) function effectively handles this by querying public.profiles.
  + ⚪ **RLS Helper for Custom JWT Array Claims**: If custom JWT claims for roles are implemented as an array (e.g., app\_roles: ["host", "pilgrim"]), an RLS helper function like current\_user\_has\_app\_role(TEXT) or jwt\_has\_app\_role(TEXT) could be beneficial to easily check array membership within policies. Example:
  + SQL

CREATE OR REPLACE FUNCTION public.current\_user\_has\_app\_role(p\_role TEXT)

RETURNS BOOLEAN AS $$

DECLARE

app\_roles TEXT[];

BEGIN

-- Supabase stores custom claims within the 'raw\_user\_meta\_data' or 'app\_metadata' in the JWT by default

-- This might need adjustment based on how claims are added.

-- For this example, let's assume it's in app\_metadata.app\_roles

app\_roles := ARRAY(SELECT jsonb\_array\_elements\_text(auth.jwt() -> 'app\_metadata' -> 'app\_roles'));

RETURN p\_role = ANY(app\_roles);

EXCEPTION WHEN OTHERS THEN

RETURN FALSE;

END;

$$ LANGUAGE plpgsql STABLE SECURITY INVOKER;

* + This is 🟠 **Optional** if the existing public.has\_role\_on\_profile(UUID, TEXT) proves sufficient for all RLS needs (it performs a DB lookup but is clear).
* **Any Changes to Existing Specs?**:  
  + 🟠 **Review** public.handle\_new\_user() **Function**: The spec for this function (Module 1) should explicitly state that it assigns default application role(s) (e.g., {'pilgrim'}) to public.profiles.roles upon new user signup. This ensures new users have a defined application role from the outset.
  + 🟠 **Consistency in RLS Policies**: All existing table specification documents should be re-checked to ensure their RLS policy examples consistently use the agreed-upon helper functions (e.g., public.has\_role\_on\_profile(auth.uid(), 'role\_code') or the new JWT claim helper if adopted) for checking application roles, rather than relying solely on auth.role() = 'authenticated' for anything beyond basic logged-in status.