Detailed API Flow Design

Here is the step-by-step flow of how the API will function, from authentication to data manipulation, using FastAPI and Firestore.

Flow 1: Authentication & Token Generation (`POST /auth/token`)

This is the entry point for any user interacting with the protected parts of the API.

Objective: A client provides user credentials and receives a signed JWT containing their identity and permissions.

Sequence of Events:

1. **Client Request:** The client application sends a `POST` request to `/auth/token`.
    - **Request Body:**
    - Generated json

```json
{
  "userId": "user_001",
  "role": "zone_admin",
  "zone": "Gabon"
}
```

2. **FastAPI Backend - Validation:**
    - The endpoint receives the request body and validates it.
    - It checks if `userId`, `role`, and `zone` are present. If not, it returns a `400 Bad Request` with the `MISSING_PARAMETERS` error.
    - It checks if the `role` is one of `super_admin`, `zone_admin`, or `normal_user`. If not, it returns a `400 Bad Request` with the `INVALID_ROLE` error.
3. **FastAPI Backend - Permission Assignment:**
    - The backend contains a simple, hardcoded mapping of roles to permissions, exactly as specified in section 1.3 of your document.
    - **Example Logic:**

- Generated python

```python
permissions_map = {

    "super_admin": {"read": ["plots", "zones"], "write": ["plots", "zones"]},

    "zone_admin":  {"read": ["plots", "zones"], "write": ["plots", "zones"]},

    "normal_user": {"read": ["plots", "zones"], "write": []}

}
```

4. **FastAPI Backend - JWT Creation:**
    - The backend constructs the JWT payload dictionary.
    - It includes `iat` (issued at time), `exp` (expiration, e.g., 24 hours from now), the `userId`, `role`, `zone` from the request, and the `permissions` object derived in the previous step.
    - It signs this payload using a secret key (stored securely as an environment variable) to create the final token string.
5. **Backend Response:** The server sends a `200 OK` response back to the client.
    - **Response Body:**
    - Generated json

```json
{

  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."

}
```

Visual Flow Diagram (Authentication):

```
Client App                                    FastAPI Backend

   |                                          |

   |  POST /auth/token                                     |

   |  Body: {userId, role, zone}  --------------> |

   |                                          |

   |                                          | 1. Validate Input

   |                                          | 2. Assign Permissions based on Role

   |                                          | 3. Create & Sign JWT Payload

   |                                          |

   |  <----------------------------------------- |

   |  200 OK                                          |

   |  Body: {token: "..."}                          |

   |
```

Flow 2: Authorized API Call (The General Pattern for All Protected Endpoints)

This flow describes how the generated token is used to access protected data.

1. **Client Request:** The client stores the JWT from Flow 1. For any subsequent request to a protected endpoint (e.g., `GET /plots/available`), it includes the token in the `Authorization` header.
    - **Header:** `Authorization: Bearer eyJhbGciOiJIUzI1Ni…`

2. **FastAPI Backend - Token Verification (Dependency):**
    - A reusable "dependency" function automatically runs before the main endpoint logic.
    - It extracts the token from the `Authorization` header.
    - It decodes the JWT using the same secret key. If the signature is invalid or the token is malformed, it immediately returns a `401 Unauthorized` error.
    - It checks the `exp` (expiration) claim. If the token is expired, it returns a `401 Unauthorized` error.
    - If everything is valid, the dependency passes the decoded payload (containing `userId`, `role`, `zone`, `permissions`) to the endpoint function.
3. **FastAPI Backend - Authorization (Endpoint Logic):**
    - The endpoint function now has the user's identity and a clear list of their permissions. This makes authorization checks simple `if/else` statements.
    - **Example:** `if "plots" not in user_payload["permissions"]["write"]:` -> `raise HTTPException(status_code=403, detail="Forbidden")`

Flow 3: Retrieving Available Plots (

This flow demonstrates a protected "read" operation with role-based filtering.

1. **Client Request:** Sends a `GET` request to `/plots/available?country=Gabon&zoneCode=GSEZ...` with the `Authorization` header.
2. **FastAPI Backend:**
    - The token verification dependency runs successfully, providing the user's payload to the endpoint.
    - The endpoint checks if the user has `read` permission for `plots` (which all roles do).

- **Firestore Query Logic:** The backend builds a query for the `plots` collection in Firestore.
    - It applies `.where()` clauses for all query parameters provided by the client (`country`, `zoneCode`, `category`, `phase`).
    - **RBAC ENFORCEMENT:** It checks the `role` from the token payload.
        - If `role` is **`zone_admin`**, it adds an *additional, non-negotiable* filter to the query: `.where("zoneCode", "==", user_payload["zone"])`. This guarantees the admin can *only* see plots from their assigned zone, regardless of what they request in the query parameters.
        - If `role` is `super_admin` or `normal_user`, this extra zone filter is not applied.
- The backend executes the query against Firestore.
- It loops through the returned documents, formats them into the required JSON structure, and sends the `200 OK` response.

Visual Flow Diagram (Plot Retrieval):

```
Client App                        FastAPI Backend                     Firestore DB

   |                               |                             |

   |  GET /plots/available?zoneCode=GSEZ        |                              |

   |  Header: Authorization: Bearer <token> ----> |                              |

   |                               |                             |

   |                               | 1. Verify Token (Dependency)             |

   |                               |    - Get user_payload {role, zone, etc.}   |

   |                               |                             |

   |                               | 2. Authorize: Check for "read" permission  |

   |                               |                             |

   |                               | 3. Build Firestore Query                 |

   |                               |    - Add .where("zoneCode", "==", "GSEZ")  |

   |                               |    - IF role == 'zone_admin':             |

   |                               |      - Add .where("zoneCode", "==",
user_payload["zone"]) |

   |                               |----------------------------------------> | Query plots

   |                               |                             |

   |  <------------------------------------------| <--------------------------------------- | Return
documents

   |  200 OK                            | 4. Format documents to JSON             |

   |  Body: {plots: [...]}                   |                              |
```

Flow 4: Updating a Plot (`PUT /update-plot`)

This flow demonstrates a protected "write" operation.

1. **Client Request:** Sends a `PUT` request to `/update-plot` with the plot data in the request body and the `Authorization` header.
2. **FastAPI Backend:**

   The token verification dependency runs, providing the user's payload.

   - **Authorization Check 1 (Permission):** The endpoint first checks if `"plots"` is in the `user_payload["permissions"]["write"]` list.
     - For a `normal_user`, this list is empty. The backend immediately returns a `403 Forbidden` error. The process stops here.
   - **Authorization Check 2 (Scope):**
     - If the user's role is `zone_admin`, the backend compares the `zoneCode` from the *request body* with the `zone` from the *user's token payload*. If they do not match, it returns a `403 Forbidden` error.
   - **Firestore Update Logic:** If all checks pass, the backend proceeds.
     - It finds the specific plot document in Firestore to update (e.g., by querying for the `plotNumber` and `zoneCode`).
     - It uses the `.update()` method on the document reference, passing in the data from the request body.
   - The backend returns a `200 OK` with the success message.