ChatGPT

Arcade AssetGen Auth & Authorization Strategy

Firebase Authentication Setup

Use **Firebase Authentication** as the identity provider for Arcade AssetGen, enabling common sign-in methods like **Google Sign-In** and **email/password**. This ensures each user has a unique Firebase `uid` to identify them. All users must sign in via Firebase Auth (with email verification for email/password accounts) before accessing Firestore. You can allow multiple providers per user (e.g. linking Google and email) so that invites sent via email can be accepted with Google sign-in as long as the email matches.

Sign-In Methods: Enable OAuth providers (Google, etc.) and email/password in Firebase Auth. Leverage Firebase's built-in UI or SDK to handle the sign-in flows. Firebase Auth will handle password hashing, Google OAuth tokens, etc., simplifying user management. For added security, consider enabling features like email verification and multi-factor auth if appropriate, but primarily focus on obtaining a verified user identity (`request.auth.uid` in security rules).

User Identity: Once signed in, the Firebase Auth ID token carries the user's identity and any custom claims. We will use the Firebase UID as the key to link users to their organization memberships in Firestore. The **Auth UID** will map to membership documents (and possibly custom claims) that determine what orgs/projects a user can access.

Multi-Tenant Data Model and Roles

Arcade AssetGen uses a **multi-tenant Firestore data model** with a hierarchy of organizations → projects → assets (and related sub-resources like generations and concept images). Each organization (tenant) contains projects, and projects contain assets/generations. We define role-based access at both the organization level and project level:

- **Organization Roles:** e.g. **Admin**, **Editor**, **Viewer** (and possibly an Owner/Creator role which can be treated as Admin). Org Admins have full control over the organization (all projects and assets within), including managing members. Org Editors can modify content (projects, assets) but maybe not manage members. Org Viewers have read-only access to that org's data.
- **Project Roles:** It's possible to invite a user to a specific project without giving org-wide access. Project-level roles could mirror the same hierarchy (Project Editor, Project Viewer). A project role grants permissions only for that project's data. An **Org Admin** implicitly has access to all projects in the org, so org-level roles override project roles.

Data Model Structure: To reflect this multi-tenancy, structure Firestore with organization as a top-level grouping. For example:

- A top-level collection `organizations` with documents for each org. Each org document may have metadata (name, creator, etc).

- Under each `organizations/{orgId}` document, have a subcollection `projects` for that org's projects. Each project doc contains project info and a reference to its parent org (or the orgId in a field).
    - Under each project (e.g. `organizations/{orgId}/projects/{projectId}`), you can have subcollections for `assets`, `generations`, `conceptImages`, etc. Each asset/generation document includes a field for `orgId` and `projectId` for security checks and queries.
- Also under each organization, include a `members` (or `memberships`) subcollection listing users in that org and their org-level role.
- Alternatively, you could keep `projects` as a separate top-level collection with an `orgId` field, and similarly top-level `assets` with `projectId` and `orgId` fields. However, using subcollections under the org keeps data naturally partitioned by tenant, simplifying security rules. It's fine to use either approach as long as every document contains a tenant identifier field (`orgId`) for rule enforcement [1].

For this design, we'll assume **nested collections by org** (it aligns with easier rule patterns). Projects are scoped to one org, so they live under that org's document. Assets, generations, concept images live under their project. This hierarchy ensures that the path of any asset includes the org and project it belongs to, which helps in writing concise security rules.

Representing User Membership & Roles

Each user's membership in organizations and projects will be tracked in Firestore. We will use **Firestore documents to represent memberships** (with roles), and optionally use **Firebase custom claims** as a cache for these roles where it makes sense. The primary source of truth will be Firestore, to allow flexible role management from the client app or admin panel.

Organization Membership Documents: Each org has a subcollection like `organizations/{orgId}/members` (or `memberships`). Each document in this subcollection represents one user's membership in that org. A recommended practice is to use the **user's UID as the document ID** in the members subcollection [2]. For example: `organizations/ORG123/members/UID456 { role: "editor", addedOn: ..., ... }`. Using the UID as the key ensures a user can only have one membership per org and makes security rules simpler (we can easily check `exists(.../members/$(request.auth.uid))`).

Each membership doc contains at least a `role` field (like "admin"/"editor"/"viewer"). You might also include a reference to the user's profile document or duplicate some user info (name, email) for convenience in queries or display [2]. Duplicating minimal info (like user name or email) on the membership can save extra reads when listing members, and can be kept in sync via Cloud Functions.

Project Membership: For project-specific access, have a similar subcollection on the project (e.g. `organizations/{orgId}/projects/{projectId}/members` for users specifically added to that project). Again, use the UID as the doc ID and include a `role` field (e.g. "editor" or "viewer" at the project scope). If a user is an **Org Admin**, you typically would not need to add them to each project's members – the security logic will grant org admins access to all sub-resources. Project-level membership is mainly for users who are *not* org-wide members. For example, an external collaborator could be given Editor access to a single project but no access to other projects in the org.

User Profile Collection: It's often useful to maintain a top-level `users` collection with basic user profiles (display name, etc) and perhaps a reference to their "current active org/project" for UI convenience [3] [4]. This isn't strictly required for security, but having a `users/{uid}` doc can help in invitations (store invite tokens or track which org a user last used). The membership docs might store a reference to the `users/{uid}` doc as well [2].

Custom Claims (Optional): Firebase Custom Claims can be used to mirror some of this membership info on the Auth token for performance. For example, you could set a custom claim with a list or map of orgs the user belongs to and their role (e.g. `orgRoles: { ORG123: "admin", ORG999: "viewer" }`). This allows security rules to quickly check `request.auth.token.orgRoles[orgId]` without reading Firestore for basic decisions. However, using custom claims for complex role structures has trade-offs: - **Token size limits:** Custom claims are included in the ID token (max ~1000 bytes). Listing many org/project roles could hit limits. - **Staleness:** Claims are set via Admin SDK and only refresh when the client obtains a new token. If roles change frequently or invites are accepted, there's a delay until the claim updates (the user would need to reauthenticate or the client to refresh the token after the admin update). - **Maintenance overhead:** You need Cloud Functions or server logic to update claims whenever membership changes (e.g., on user added/removed or role change) [5] [6].

Given these factors, a **hybrid approach** is wise: use Firestore as the source of truth and primary means of checking roles, and leverage custom claims *for static or frequently-needed info*. For instance, you might put an `isPlatformAdmin` flag in claims for global admins, or an array of org IDs the user is a member of (if that list is small and changes rarely). But for per-request fine-grained access (especially project-level roles), Firestore lookups are often simpler.

> **Why Firestore for roles?** In multi-tenant systems with dynamic membership, modeling authorization in the database is often more convenient for management [7]. It allows building a UI to invite/remove users and change roles by simply updating documents, rather than calling admin APIs to adjust claims each time. Firebase's own guidance notes that while you *can* use JWT claims for roles, a database-driven approach is usually easier to maintain for complex multi-tenant scenarios [7].

That said, custom claims can improve read performance by avoiding frequent document reads in security rules. A common pattern is to use Firestore to manage roles and have a Cloud Function sync certain info to custom claims [8] [6]. For example, when a user's memberships change, update a claim like `orgs: ["ORG123","ORG999"]` so you can do simple membership checks via the token. **Frank van Puffelen** (Firebase engineer) summarizes the trade-off: *"custom claims are more convenient and readable, while using the database is usually faster. It's common to use the database lookup for volatile information, and claims for information that is more static."* [9]. In our case, org/project roles can be considered semi-static (they change occasionally, not every minute), so you could go either way. We'll primarily rely on Firestore in security rules, but will design the system such that adding custom claims later (or in specific cases) is possible for optimization.

Firestore Security Rules Structure

Security rules will enforce that users can only access documents of orgs/projects where they have membership, and only perform allowed actions based on their role. We will use Firebase **Firestore Security**

Rules (rules_version = '2'), taking advantage of functions and recursive wildcards for collection group queries.

Key aspects of the security rules design:

- **Authenticated Access Only:** All accesses require `request.auth != null` (logged-in users). No anonymous or public reads/writes to the protected collections.
- **Organization Membership Check:** We create a helper function, e.g. `hasOrgAccess(orgId)` that returns true if the current user is a member of org `orgId` (in any role). This can be implemented by checking the existence of the membership doc for that user in the org's members subcollection [10] [11]. For example:

```
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {
    function hasOrgAccess(orgId) {
      return exists(/databases/$(database)/documents/organizations/$(orgId)/
members/$(request.auth.uid));
    }
    function hasOrgRole(orgId, role) {
      return get(/databases/$(database)/documents/organizations/$(orgId)/
members/$(request.auth.uid)).data.role == role;
    }
    ...
  }
}
```

Here, using `exists()` is a light check (just sees if a membership doc exists) suitable for read access control [10]. Using `get()` to retrieve the `role` allows us to compare roles for finer permissions (we'll use that for write access) [12] [13]. Keep in mind each `get()` or `exists()` counts as a document read in rules.

- **Project Membership Check:** Similarly, we may have a function `hasProjectAccess(orgId, projectId)` that checks either:
- The user has org access (if they are in the org at any role, they implicitly can at least read projects? Depending on design, maybe even org viewers can see all projects, or you may require explicit project membership for non-admins).
- OR the user has a membership in that specific project's members subcollection.

We can combine these: e.g.

```
function hasProjectAccess(orgId, projectId) {
  return hasOrgAccess(orgId) || exists(/databases/$(database)/documents/
```

```
organizations/$(orgId)/projects/$(projectId)/members/$(request.auth.uid));
}
```

And similarly a `hasProjectRole(orgId, projectId, role)` to get the project membership doc's role (and also allow org admins regardless).

- **Rules for Org Documents:** Only members of an organization can read its data. For example:

```
match /organizations/{orgId} {
  allow get, list: if request.auth != null && hasOrgAccess(orgId);
  allow create: if request.auth != null;  // allow creating a new org (any
authenticated user can start a new org; you might restrict this if needed).
  allow update, delete: if request.auth != null && hasOrgRole(orgId,
'admin');
  ...
  // Subcollections rules (projects, members, etc) will be defined inside
this block.
}
```

  This ensures only org members can read org details, and only org admins can modify the org (e.g. change its settings or delete it). Creation of a new org is allowed to any signed-in user (alternatively, you could require some special claim or limit who can create orgs if needed).

- **Rules for Projects:** Within the org rule:

```
match /organizations/{orgId}/projects/{projectId} {
  allow get, list: if request.auth != null && hasProjectAccess(orgId,
projectId);
  allow create: if request.auth != null && hasOrgAccess(orgId) &&
hasOrgRole(orgId, 'editor');
  allow update, delete: if request.auth != null && (
          hasOrgRole(orgId, 'admin') ||    // org admin can modify any
project
          get(/databases/$(database)/documents/organizations/$(orgId)/
projects/$(projectId)/members/$(request.auth.uid)).data.role in
['admin','editor']
        );
}
```

  In reads, we allow any org member or project-specific member to fetch the project. For writes, we allow if the user is an **Org Admin** or has a project membership role that permits it (e.g. project-level admin/editor). If you want project-level roles like "Project Owner/Admin", you might designate a role value and check for that as well.

- **Rules for Assets/Generations/ConceptImages:** All these are subordinate to a project. We can use a **recursive rule** to cover any subcollections under a project:

```
match /organizations/{orgId}/projects/{projectId}/{subcoll}/{docId} {
  allow read: if request.auth != null && hasProjectAccess(orgId,
projectId);
  allow create: if request.auth != null && hasProjectAccess(orgId,
projectId) && (
          hasOrgRole(orgId, 'editor') || get(/databases/$(database)/
documents/organizations/$(orgId)/projects/$(projectId)/members/$
(request.auth.uid)).data.role in ['admin','editor']
        );
  allow update, delete: if request.auth != null && (
          hasOrgRole(orgId, 'editor') || get(/databases/$(database)/
documents/organizations/$(orgId)/projects/$(projectId)/members/$
(request.auth.uid)).data.role == 'editor'
        );
}
```

  This wildcard `{subcoll}` can match “assets”, “generations”, “conceptImages” (and any future subcollection) – granting the same access rules to all. Essentially, if you can access the project, you can read its sub-items. Writing (creating/updating) an asset or generation requires edit privilege at either org or project level. Org Editors/Admins can create assets in any project, or a project-specific Editor can do so for that project [14]. Viewers (no edit rights) would fail the write rule.

If certain subcollections need stricter rules (for example, perhaps only admins can delete a generation), you can also write separate matches for them. But the above gives a general pattern.

- **Membership Management Rules:** We must protect the membership documents themselves. Only authorized users should be able to read or modify the org’s member list:
- **Read members:** Likely only org members can read the list of members (or perhaps only admins can list members – depends on your app requirements). We could allow org **Admins** (and possibly Editors) to list all members. Or even all org members can see who else is a member. For privacy, you might restrict this to at least org Editor or above.

```
match /organizations/{orgId}/members/{userId} {
  allow get: if request.auth != null && hasOrgAccess(orgId);
  allow list: if request.auth != null && hasOrgAccess(orgId);
  // Only members can see membership entries (could tighten to admin only
for list).
}
```

- **Add member (Invite acceptance):** We will handle invites via a secure method (described later), but when it comes to writing a new membership doc in Firestore, enforce that only the proper flow can do so. For instance, an Org Admin can **invite** a user by creating an invite record (not directly the

membership), and a Cloud Function will create the membership when accepted. Or if directly adding (without invite acceptance flow), only an Admin should create a `members/{uid}` doc:

```
allow create: if request.auth != null
              && hasOrgRole(orgId, 'admin')
              && request.resource.data.role in ['admin','editor','viewer'];
```

This says an admin can add a new member with an assigned role. You'd also ensure the `userId` (doc ID) matches the intended user's UID – but since an admin is doing it, they provide it. Alternatively, if using invite self-service (user creates their own membership with an invite code), you'd have a different rule (see the Invitation section below).

- **Update member (role changes):** Only Org Admins should change another user's role (e.g. promote/demote) or remove a member. So:

```
allow update, delete: if request.auth != null && hasOrgRole(orgId,
'admin');
```

Possibly with a condition that they aren't demoting themselves out of admin (edge-case to consider in client logic).

- **Project members subcollection:** Similarly, only certain people can add/remove project-specific members. Likely Org Admin or Project Admin can manage project members. You'd write analogous rules under `projects/{projectId}/members/{uid}`.

- **Collection Group Queries:** We plan to allow some collection group queries (for instance, querying all projects a user has access to, or all assets in an org). With rules_version 2, we can write a rule to cover a collection group. One common use-case: get all memberships of the current user across teams/orgs. In our model, that could be "find all orgs (or projects) where my UID appears in the members subcollection". This can be done via a collection group query on `members`. We need a rule to allow it safely:

For example, to allow a user to query all their org memberships:

```
match /{path=**}/members/{memberId} {
  allow list: if request.auth != null && memberId == request.auth.uid;
  allow get: if request.auth != null && memberId == request.auth.uid;
}
```

This rule (with `{path=**}` covering any `members` subcollection in any org) ensures that a user can read membership docs *only* where the doc ID matches their own UID [15]. That way, a query like `firestore().collectionGroup('members').where('__name__', '==', 'organizations/ORG123/members/UID456')` or more practically `.where('someField', ... )` that results in only their memberships will be allowed. It prevents listing all members in all orgs. In practice, you would query

by a field (like `userRef` or similar), but this rule is a safety net that the only members docs they can ever see have to be theirs. We can similarly allow querying project membership docs for the user by a similar rule on `/{path=**}/projects/{projectId}/members/{memberId}`.

If you want to allow users to list all their projects across orgs, one approach is to store a reference to the user in each project membership (or a separate index). However, it may be simpler to just query the `memberships` collection group as above, then get the parent project/org info.

In summary, the Firestore rules will leverage **membership documents as the gatekeeper** for all data access. If a user's UID is present in the appropriate members subcollection (with sufficient role), the rule will allow access; if not, access is denied [16]. This approach ensures **zero trust**: even if a malicious client tries to read or write an org they're not part of, the rules will block it.

Protecting Collection Group Queries

Special care is needed for **collection group queries** to avoid leaking data across tenants. Firestore security rules evaluate queries against *all potential matches*, not just actual returned docs [17] [18]. This means if a query could retrieve documents a user isn't allowed to see, the entire query will be rejected. We must design rules and queries so that queries are always narrow enough to only include authorized data.

Tenant Isolation in Queries: Always include the tenant (org or project) filter in any collection group query. For example: - To load all assets in a given org, query `collectionGroup('assets').where('orgId','==',currentOrgId)`. The security rule for assets will require that `resource.data.orgId == orgId` that the user has access to. Because the query includes that filter, it aligns with the rule and will succeed [19] [20]. If the query tried to fetch *all* assets without specifying org, it would fail (since it could include unauthorized orgs) [21]. - If querying all projects the user can access, you might not know org upfront if they have many. In that case, it's better to do multiple queries (one per org) or have a specific index. For instance, if using the membership collection group approach: query `collectionGroup('members').where('memberId','==', myUid)` to get all memberships for that user, then derive the projects/orgs. This query is allowed by our rules because of the check `memberId == request.auth.uid` which guarantees the query only returns membership docs of the current user [15] [22]. It will **fail** if someone attempted `collectionGroup('members').get()` with no filter, because that would include other users' docs which is not permitted.

Security Rule Constraints on Queries: Firestore rules can distinguish between single-document gets and queries (list). We used `allow get, list` appropriately. We also ensure that for collection group rules, the **same conditions apply to each result**. For example, our asset rules require `hasProjectAccess(orgId, projectId)` for each asset. If a query spans multiple projects/orgs, the user must have access to *all* those projects or the query will be denied. This is usually fine because the user will include filters for org or project. In some cases, if a user belongs to many orgs and wants to query data from all of them at once, they might hit the scenario of mixing authorized and unauthorized. We should avoid queries that span tenants arbitrarily. It's safer to query one org at a time, or use an `in` query for a limited set of orgs.

Rules Version 2: We use `rules_version = '2';` at the top of our rules. This is required to support recursive wildcards for collection group queries [23]. It also gives access to `request.query` variables if

needed (we mostly rely on implicit matching of constraints rather than explicit `request.query`, but version 2 is essential for multi-tenant queries).

Example – User's assets across projects: Suppose we want to let a user search their name across all assets they own in any of their orgs. One could query `collectionGroup('assets').where('ownerId','==', myUid)`. For this to succeed, our rules for assets must not only check membership but also allow that specific query. If the asset rule is `allow list: if hasProjectAccess(orgId, projectId)` (which is per-doc check), Firestore will see that the query lacks an explicit orgId/projectId equality. However, because `ownerId == myUid` doesn't guarantee the user has access to those projects (they might own assets in a project they left – unlikely if membership required to create – but anyway), the safer approach is to structure queries with tenant filter. Alternatively, we could enforce that an asset document includes the orgId and that the user is member of that org. The rule already does that check per doc (`hasProjectAccess` -> `hasOrgAccess or project membership`). Firestore will evaluate each possible asset where owner == myUid; if any asset's org fails `hasOrgAccess`, the query fails entirely [17]. Thus, to avoid failure, the user should only own assets in orgs they belong to (which should hold true).

In summary, to **protect against unauthorized access in collection group queries**, ensure: - **Rules** require tenant fields match the user's permissions. - **Queries** from the client always include the necessary filters (orgId, etc.) so that the query cannot retrieve another tenant's data. If a developer mistakenly queries without filtering by org when rules demand it, the query will be rejected (which is good). We will document for developers that any collectionGroup query must include an equality condition on `orgId` or a similar membership constraint.

Invitation & Onboarding Workflow

Supporting user invitations is crucial for onboarding new members to an organization in a secure manner. The invite system will allow an org admin to add a user by email, even if that user doesn't yet have an account.

Invite Object: We introduce an `invitations` collection (or subcollection under the org). For example, `organizations/{orgId}/invitations/{inviteId}` documents that contain an `email` field and a `role` field, plus perhaps an expiration timestamp or status. Only Org Admins can create an invitation in this collection, specifying the invitee's email and intended role:

```
match /organizations/{orgId}/invitations/{inviteId} {
   allow create: if request.auth != null && hasOrgRole(orgId, 'admin')
                 && request.resource.data.email is string
                 && request.resource.data.role in ['admin','editor','viewer'];
   allow read: if request.auth != null && hasOrgRole(orgId, 'admin');
   allow delete: if request.auth != null && hasOrgRole(orgId, 'admin');
}
```

(The read/delete could also allow the invited user to fetch their own invite if you want that, but often the invite is processed via backend, so only admins need read access to see pending invites.)

Sending the Invite: When an admin creates an invite (via your application), you'd typically trigger a Cloud Function or some server process to send an email to that address. The email should contain a secure link or code. One approach is to use Firebase's Email Link Authentication (passwordless sign-in) combined with some query parameters for org and invite ID. Alternatively, generate a random invite code and include it in the link.

Accepting the Invite: The user clicks the invite link, which brings them to your app. You may have them sign in (or sign up) with Firebase Auth at this point. **Important:** The email they use to sign in should match the invite email (you can enforce this by pre-filling the email or checking after sign-up). After a successful sign-in, your client app (or a Cloud Function) will verify the invite: - The app could call a Cloud Function like `acceptInvite(orgId, inviteCode)` which: - Checks that an invite with that code (or ID) exists and is still valid. - Verifies that the `request.auth.uid` corresponds to an Auth user whose email matches the invite's email [24] [25]. - If everything checks out, it creates the Firestore membership doc for that user in `organizations/{orgId}/members/{uid}` with the role from the invite, and deletes/marks the invite as used. - Possibly also sends a notification to org admins that the user joined.

By doing this in a trusted environment (Cloud Function with admin privileges), we avoid exposing direct membership creation to the client. The security rules can back this up by only allowing that membership creation if an invite exists. For example, a rule could allow a user to create a membership for themselves *if an invite is present*:

```
match /organizations/{orgId}/members/{memberId} {
    allow create: if request.auth != null
                  && memberId == request.auth.uid  // user adding themselves
                  && request.resource.data.role != null
                  && // check invite exists for this email
                     exists(/databases/$(database)/documents/organizations/$
(orgId)/invitations/$(request.auth.token.email));
}
```

However, checking the invite via rules is tricky because the invite ID might not be the email (could be random). A simpler approach is to let the Cloud Function bypass security (it uses Admin SDK) to write the membership, as it can do its own checks. The rules would primarily ensure that normal users cannot create arbitrary memberships.

Adding Existing Users: If the invited email already corresponds to an existing Firebase user, the acceptance flow can be streamlined: - The admin invites the email. - A Cloud Function could detect that this email is already in Auth (by listing users or maintaining a mapping in Firestore). If so, it can immediately add the membership doc and perhaps send a notification to that user (if the app has a mechanism). - Alternatively, send the email anyway; when the user logs in next, you could show "You have been added to Org X" (because the membership doc will now be present).

You might prefer requiring explicit acceptance even for existing users (for compliance or consent), in which case treat them similarly to new users: send an invite, and maybe only add the membership when they click accept. This can be implemented by storing invites and not auto-adding membership until confirmed.

Invitation Security: The invite link should contain a **secure token** (e.g. a UUID or short random code stored in the invite doc). This prevents guessing invites. Also enforce one-time use: once used, mark the invite doc as redeemed (or delete it).

On the Firestore side, **never trust the client to add themselves to an org without verification**. That's why we restrict membership creation in rules and prefer an authenticated Cloud Function. The rules snippet above with `exists(invitation)` is a theoretical safeguard; in practice it's safer to not allow clients to directly write their membership at all (except perhaps in a very controlled rule when using the invite reference trick as described by Richard Keil [26]). His approach: the invite (with email) serves as proof in rules that a user with that email can create a membership doc [26]. Implementing that purely in rules is complex because rules would need to compare `request.auth.token.email` to the invite's email field. Firestore rules can read the invite doc via `get()`, and `auth.token.email` is available if the email is verified. So you *could* do:

```
allow create: if request.auth != null
              && memberId == request.auth.uid
              && request.auth.token.email != null
              && get(/organizations/$(orgId)/invitations/$
(inviteId)).data.email == request.auth.token.email;
```

This assumes the client provides the `inviteId` (or you derive it somehow). This is possible but slightly brittle. A safer pattern is offloading to server logic.

Invitation Expiration: Implement an expiration on invites (e.g. a field `expiresAt`). The Cloud Function or security rule should check this. For example, the rule could ensure `request.time < invite.expiresAt` when allowing membership creation. Or the Cloud Function refuses if expired.

In conclusion, the invite flow will involve: 1. Org Admin uses the app to invite a user by email (assigning a role and possibly project scope). 2. Firestore stores an invite doc (Org Admin authorized to do so). 3. Email sent to invitee with secure link. 4. Invitee signs in (or signs up) via Firebase Auth (we get their UID and verified email). 5. Server-side verifies the invite and creates the membership record in Firestore for that user. 6. The user now appears in the org's members and can access the data according to their role.

Enforcing Access in Client & Server Logic

While Firestore Security Rules provide the **ground truth enforcement**, a good practice is to also implement checks in the client and any server-side code: - **Client-Side Checks:** The frontend application should be aware of the user's roles (e.g. fetch the user's org memberships on login) and adjust the UI accordingly. For example, don't show an "Edit Project" button to a Viewer, and don't allow selecting an org or project the user isn't part of. This improves user experience by preventing forbidden operations upfront and helps catch mistakes (though the rules will catch any actual unauthorized attempts). You can store the user's org/project roles in a global state on the client after querying their memberships (since we allowed them to query their own membership docs). This also helps in constructing queries properly – e.g., the app knows which orgs the user has access to and will include the appropriate `where('orgId','==',...)` filters or

avoid querying data from orgs they don't belong to. - **Server-Side Logic:** If you have any Cloud Functions or a custom backend that accesses Firestore on behalf of users (or performs privileged actions), enforce the same membership logic there. **Never assume the client did the check**. For instance, if there's a Cloud Function `generateAssetThumbnail(assetId)`, that function should verify that the calling user (from `context.auth.uid`) is allowed to access that asset's org/project. In a Cloud Function, you can either replicate the check by reading the membership doc or by using Firebase's `auth` context if the function was called with Firebase Auth. Essentially, treat the server as if it must also obey the org/project roles (unless it's performing an admin action). - **Custom Claims in Backend:** If you did implement custom claims for roles, you can use those in your backend authorization as well, since they're part of the decoded JWT. For example, check `authToken.orgRoles[orgId]` before proceeding with an operation. - **Error Handling:** Be prepared for `PERMISSION_DENIED` errors from Firestore in the client. This could happen if, say, a user's role was changed or an invite expired – the client might attempt something that fails rules. Handle these gracefully (e.g. refresh the user's role info, show a message "Access denied" etc.).

By enforcing in multiple layers (UI, server, and Firestore rules), you create a defense-in-depth that mitigates both mistakes and malicious attempts.

Scalability, Caching, and Rule Evaluation Limits

Scaling with Many Tenants/Users: Firestore can handle a large number of documents and users, but you need to be mindful of how security rules scale: - The **membership documents approach** is efficient. A membership doc is small, and a single existence check is O(1). Firestore security rules also cache document reads within a single request evaluation. So if a user is reading 50 assets from the same project, and our rule calls `exists(/orgs/OrgA/projects/Proj1/members/uid)` for each asset, it should only charge one read (the first time) and reuse it for subsequent docs in that query. However, if those 50 assets span 5 projects, that could be up to 5 membership lookups (still fine). The main limit is **10** `get/exists` calls per rules evaluation [27]. If a user somehow triggered a single read that would require more than 10 distinct membership checks, the rules evaluation would fail. In practice: - A single document read (get) will only do one check for the specific org/project it's in. - A query across multiple orgs/projects might hit the 10 limit if the user is in >10 projects returned. Usually, your queries will be constrained to one org or a small set of projects, so this isn't a problem. If the use-case arose (say a user in 15 projects wants to query all their assets at once), we might handle it by multiple queries or by structuring the data differently (sharding the query). - **Caching Roles on Client:** To reduce repeated overhead, the client app can cache the user's memberships/roles (e.g. in memory or local storage). Since roles don't change often, you might only fetch them on login or when switching orgs. This way, client can avoid unnecessary queries and also quickly answer "can the user do X?" in the UI without a round trip. Just ensure to refresh this cache if the user accepts a new invite or an admin role changes (you could listen to the membership doc in real-time if desired). - **Cloud Firestore Costs:** Each `get` / `exists` in rules counts as a read for billing. Our rules might do 1-2 extra reads per request (checking membership). This is usually negligible in cost, but if a client loads a list of 100 items, and each triggers a membership lookup, that's 100 reads. As noted, Firestore tries to coalesce identical lookups, but if those 100 items are in 100 different projects, you might hit the 10 reads limit and also be doing 100 reads (denied after 10). To avoid issues, design your queries to fetch data in sensible batches (e.g. one project at a time if the user has so many). In most applications, a user won't belong to extreme numbers of orgs/projects actively. - **Use of Custom Claims for Performance:** If you observe that the membership checks are becoming a bottleneck (e.g. lots of rule reads), you could pivot to use custom claims for the most common checks. For example, include an array of org IDs in the claim. Then the rule `hasOrgAccess` could be:

```
return request.auth.token.orgs != null && orgId in request.auth.token.orgs;
```

This avoids a Firestore read entirely (just checks the token). As long as the token is kept up-to-date, it's very fast. For role checks, you might include an `orgRoles` map in claims as discussed. Just be cautious to update on changes and handle token refresh. A hybrid approach is also possible: use the token for broad membership and use Firestore for granular (like project-level) checks.

- **Rule Evaluation Complexity:** Keep functions straightforward and avoid heavy computations. The rules we outlined mostly do simple existence checks and equality comparisons, which is fine. We should avoid any rule that needs to scan arrays of large size or similar. Storing roles as a field (string) is simple. If we had multiple roles per user, an array might be used and rules can use `array-contains` in a get, but in our case one role per membership is sufficient.

- **Firestore Indexes:** Ensure you create necessary indexes for your queries. Multi-tenant typically means adding composite indexes for things like `collectionGroup('members').where('user','==',...)`, or `collectionGroup('assets').where('orgId','==',...). Firestore will prompt for needed indexes. This is more about performance than security, but worth noting.

- **Membership Doc Sync:** If you duplicate user or org info in membership docs (like user name or team name), use Cloud Functions to keep those in sync on updates [28]. This doesn't directly affect security, but ensures data consistency at scale.

Scaling Number of Roles: The approach supports many orgs and users, but consider extremes: - If an organization has thousands of members, listing them means thousands of membership docs. This is fine; queries and rules can handle that (with proper indexes). Just ensure pagination/limits on reads. - If a user is in hundreds of orgs (rare in a typical setup like this), their ID token might overflow if you put all in custom claims. It might be better then to rely on Firestore queries for their memberships (which we already plan to allow). - The 1MB document size limit means you shouldn't try to store *all* members in a single org document field (we are not doing that – we use subcollection per member). That way, you can scale to many members per org. Each membership is an individual doc, avoiding any hard size limit per org.

Caching on Backend: If you have administrative server code frequently checking roles, you might implement an in-memory cache (for example, caching the membership lookup result for a given user-org pair for a short time) to reduce Firestore reads. Be careful with cache staleness though – if roles change, you might serve outdated info. Since role changes are infrequent, even no cache is acceptable; just mentioning as an option if needed.

Firestore Rule Limits: We already mentioned the 10 doc read rule limit [27]. Also note that rules have a size and complexity limit – our described rules are moderate and should be well within limits. Using functions helps reuse code without duplication. If the ruleset grows (covering many subcollections separately), keep an eye on overall size (~256 KB text limit). Our approach of generic `{subcollection}` catch-alls helps keep it succinct.

Summary

By combining **Firebase Authentication** for identity management and **Firestore Security Rules** for data authorization, we achieve a robust multi-tenant security model: - **Auth** provides verified user identities (with support for multiple sign-in methods and easy integration). - **Role memberships** are stored in Firestore (`members` subcollections under each org/project). This makes permission management and queries straightforward (e.g. listing team members or a user's teams) [28] [16]. - **Security rules** strictly enforce that only members can read/write data in the appropriate scope, checking the existence of a membership document or role field for permission [10] [12]. Org and project roles are respected for every database operation. Collection group queries are allowed only with constraints that match the user's permissions to prevent any cross-tenant data leakage. - **Invitations** are handled via a secure flow using invitation records and email verification, ensuring users can be onboarded without exposing the system to unauthorized sign-ups [26]. Only admins can send invites, and only the intended user can accept, at which point a controlled process adds them as a member. - **Application logic** (both client and any backend) complements these rules by checking roles and tailoring the experience, though the ultimate enforcement is at the database level. - **Performance** is kept in mind by using efficient rule checks (with possible augmentation via custom claims) and by structuring data to leverage Firestore's indexing and rules capabilities. We note the 10-read rule limit and design around it, ensuring our common queries stay within safe bounds [27].

This strategy should scale well as your user base and number of organizations grows, while keeping each tenant's data isolated and secure from unauthorized access. It provides a clear path for managing roles at both the org and project level, and a blueprint for adding new members through invites in a secure manner.

---

[1] Is using firestore for multi tenant best practices? : r/Firebase
https://www.reddit.com/r/Firebase/comments/1lpg4e4/is_using_firestore_for_multi_tenant_best_practices/

[2] [3] [4] [7] [10] [11] [12] [13] [14] [15] [16] [22] [24] [25] [26] [28] How to build a team-based user management system with Firebase | by Richard Keil | Firebase Developers | Medium
https://medium.com/firebase-developers/how-to-build-a-team-based-user-management-system-with-firebase-6a9a6e5c740d

[5] [6] [8] How do you keep your custom claims in sync with roles stored in Firebase database - Stack Overflow
https://stackoverflow.com/questions/54394316/how-do-you-keep-your-custom-claims-in-sync-with-roles-stored-in-firebase-databas

[9] firebase - Firestore Rules with multi-tenancy? - Stack Overflow
https://stackoverflow.com/questions/63291425/firestore-rules-with-multi-tenancy

[17] [18] [19] [20] [21] [23] Securely query data | Firestore | Firebase
https://firebase.google.com/docs/firestore/security/rules-query

[27] Patterns for security with Firebase: group-based permissions for Cloud Firestore | by Doug Stevenson | Firebase Developers | Medium
https://medium.com/firebase-developers/patterns-for-security-with-firebase-group-based-permissions-for-cloud-firestore-72859cdec8f6