Access Control Decision Guide

Companion to the Authorization and Access Control docs.

Section 1: Use Case & Participants

1.1 Who are the users?

- Patients only
- Providers / staff / partners (clients or outside vendors) only
- Both

1.2 What is your organizational structure?

- Single organization
- Multiple locations of one organization
- Multiple independent organizations (MSO-style)
- Departments or service lines within one organization
- Combination

Why: biggest driver of downstream design. If single org with no internal isolation, skip Section 2.

Section 2: Multi-tenancy

Skip if single-organization with no internal isolation.

Isolation uses **compartments** (`meta.compartment`) matched by parameterized access policies.

Design principle (from Medplum docs): Push complexity into enrollment Bots, not policies. Each patient is tagged with their tenant(s); each practitioner's `ProjectMembership.access` lists every tenant they work in. See the MSO demo `enrollment.ts` for reference.

2a Tenant Modeling

Questions:

- Internal term for tenants? (clinics, programs, sites, accounts)
- Grouping is organizational, functional, or per-patient?
- Is there an organizational hierarchy (e.g., region → clinic)?

Situation	Approach
Clinics, practices, locations (MSO)	`Organization`
Service lines, departments	`HealthcareService`
Dedicated team per patient	`CareTeam` per patient

Hierarchy pattern (if tenants nest, e.g., region → clinic):

- Patients are tagged only at the **lowest level** they belong to.
- Practitioners enrolled at an intermediary level get one `ProjectMembership.access` entry per lowest-level tenant under it (iteratively expanded).
- Adding a new lowest-level tenant requires a workflow to update every practitioner membership at or above that intermediary.

2b Patient and Practitioner Sharing

Questions:

- Can a patient be enrolled in more than one tenant at a time?
- Are practitioners isolated to their tenant, or visible across all tenants (global directory)?

Situation	Approach
One tenant per patient	Single compartment on the Patient
Patient shared across tenants	Multiple compartments, one per tenant
Practitioners shared (global directory)	Leave unassigned; policy reads without `_compartment` filter
Practitioners isolated per tenant	Tag with tenant compartment; policy filters by `_compartment`

Section 3: Roles & Permissions

For each role: name it, scope it (which tenants, if multi-tenant), say what resources it touches, and narrow as needed.

3a Role Inventory & Scope

Questions:

- What are the role names? (physician, RN, MA, front desk, billing, care coordinator, medical director)
- *If multi-tenant*: for each role, which tenants does it cover? (one, several, all)
- Any oversight roles that need visibility across all tenants without per-tenant enrollment?

Role scope	Modeling approach
One tenant	Single `ProjectMembership.access` entry parameterized with that tenant
Multiple tenants, combined view	One `access` entry per tenant, same policy

Role scope	Modeling approach
Org-wide / cross-tenant oversight	Unparameterized policy, no `_compartment` filter
Tenants contractually can't share data (e.g., locum across competitor clinics)	Separate logins per tenant — see callout below

⚠️ **One-way door: stacking is additive only.** Policies compose by union — you can't narrow by composition. For genuinely narrower API access in one context (not just a filtered UI view), use **separate logins**, not stacked policies.

When separate logins are warranted: contractual or regulatory obligations requiring the API to never return both scopes' data in the same session. Example: a provider works at two independent clinics under separate BAAs — stacking would let one token read both, violating the BAA. Two memberships (each parameterized to one clinic) bound each session to one tenant.

Test: *if this token leaked, should the attacker reach both scopes?* No → separate logins. Yes or don't care → stacked; any "switcher" is purely a UI concern.

3b Permission Mechanics

Questions:

- For each role: which resource types does it read / write / search / delete?
- Any partial FHIR interactions? (create-only, read-without-search, etc.)

Situation	Approach
Full CRUD on a resource	Standard `resource` entry, no `interaction` restriction
Read-only	`readonly: true`
Partial interactions	Explicit `interaction` array

3c Narrowing What a Role Can Do

Cover only the knobs that apply.

Field-level — partial field access (e.g., billing sees Patient minus diagnoses):

Situation	Approach
Readable but not writable	`readonlyFields`
Not visible	`hiddenFields` (masks output only — for decluttering, do it in the UI)

Write constraints — enforce state transitions or required fields at the policy layer:

Situation	Approach
Prevent field change in terminal state	`writeConstraint` on `%before`
Require field at transition	`writeConstraint` on `%after` + field presence
Update-only	Prefix with `%before.exists() implies ...`
Create-only	Prefix with `%before.exists().not() implies ...`

Keep FHIRPath simple; complex logic belongs in a Bot.

Criteria-based filtering — filter by resource attribute (geography, code, status):

Situation	Approach
State license	`Patient?address-state=%licensed_state`, parameterized

Situation	Approach
Multi-state	One `access` entry per state
Specialty or status filter	`criteria` with code/status filter
Criteria + tenant	Single `criteria` string combining both

`criteria` supports only `:not` / `:missing` — no chained searches. Denormalize or tag via an enrollment Bot.

Section 4: Admin Structure

Questions:

- Who invites/removes users and manages memberships?
- Who manages Bots, ClientApplications, AccessPolicies?
- Any super admin needs? (project creation via API, overwriting protected fields)
- Should admins be blocked from clinical data?

Situation	Approach
Standard admin	`ProjectMembership.admin`
Admin with narrowed clinical access	Admin flag + dedicated AccessPolicy
Admin blocked from clinical content	Admin flag + `hiddenFields` on clinical resources
Project creation, overwriting protected fields	Super admin — server operators only
Separate “user manager” vs. “tech admin”	Two AccessPolicies stacked on admin flag (see 3b)

⚠️ **Note:** Admin flag covers admin resources only (`Project`, `ProjectMembership`, `User`) — admins still need a policy for clinical data.

Appendix: Additional Decisions

Narrower decisions that apply only in specific situations. Cover each only if relevant.

A. Open Registration & Caregivers

If the product has a patient portal.

Questions:

- Self-registration or staff-invite?
- Are caregivers / proxies needed? (parents, adult children, guardians)
- One caregiver with multiple patients? One patient with multiple caregivers?
- How is the caregiver relationship established and revoked?

Situation	Approach
Invite-only	Standard `ProjectMembership` with patient policy
Open self-registration	Set `Project.defaultPatientAccessPolicy`; enable open registration
Patient sees own data	Parameterized policy with `%patient` on clinical resources
Single caregiver, single patient	One `access` entry with `%patient`
One caregiver, multiple patients	Multiple `access` entries, different `%patient`
Multiple caregivers per patient	Each caregiver has their own ProjectMembership
Revoke caregiver	Remove the corresponding `access` entry

Situation	Approach
Time-bound caregiver access (e.g., expires at age 18)	Application-layer enforcement — not native

B. SMART on FHIR

If third-party apps (not your own frontend) request access.

Questions:

- Patient-context launch, provider-context launch, or standalone?
- Which scopes?
- Offline access needed?

Situation	Approach
Your own frontend	Not SMART — standard ClientApplication + AccessPolicy
Third-party, patient-context	SMART 2.0.0, `launch/patient`
Third-party, provider-context	SMART 2.0.0, `launch`
Standalone patient app	`patient/*.rs` or narrower
Offline access	Add `offline_access`

⚠️ **Note:** Effective access = SMART scopes ∩ AccessPolicy.

C. IP Access Rules

If access should be restricted by IP range.

Questions:

- Restrict by IP range? (VPN, on-prem devices)
- Any roles that bypass?

Situation	Approach
VPN-only	Allow VPN ranges + wildcard block
On-prem only	Allow clinic network + wildcard block
Mixed	Different AccessPolicies per role

⚠️ **Note:** IPv4 only. Rules evaluate sequentially — always terminate with a wildcard block.

D. Enrollment & Reassignment Workflows

If Section 2 multi-tenancy applies.

Questions:

- When does a patient get enrolled in a tenant? (registration, first appointment, referral, insurance, geography)
- Who is authorized to move a patient between tenants?
- On transfer: do records follow the patient, stay with the original tenant, or become shared?

Situation	Approach
Patient enrollment	Bot calls `$set-accounts` on the triggering event; use `propagate: true` to tag related resources
Transfer, records follow	Update compartment with `propagate: true`; old tenant loses access
Transfer, records stay / shared	Add new compartment (don't replace); both tenants retain access

Reference implementation: MSO demo `enrollment.ts` — canonical example patterns.

E. Shared vs. Tenant-Specific Resources

If Section 2 multi-tenancy applies.

Questions:

- Terminology (CodeSystems, ValueSets) shared or tenant-specific?
- Questionnaires shared or per-tenant?
- PlanDefinitions / ActivityDefinitions global or per-tenant?
- Any resources shared across projects (not just tenants)?

Situation	Approach
Shared standard terminology	Leave unassigned; unfiltered read in policy
Tenant-specific Questionnaires	`$set-accounts` tags with tenant; filter by `_compartment`
Mixed (some global, some per-tenant)	Two resource entries in policy: one filtered, one unfiltered
Global Bots	Leave unassigned; Bots run as ClientApplication
Reference data shared across multiple projects	Super admin creates a linked project (read-only view)

⚠️ **Note:** Forgetting unfiltered read on standard terminology causes empty dropdowns — a common multi-tenant misconfiguration.

F. Role-Aware UI

If your frontend needs to show / hide UI based on the user's role

Situation	Approach
UI changes based on role	`AccessPolicy.basedOn` + `/auth/me`

Using `AccessPolicy.basedOn` keeps the UI in sync with what the API enforces. The alternative (hardcoded role field, custom endpoint) tends to drift.