Permissions

Rule Bases, SecurityURLHeaders, and SecurityURLBody

Version 1.2.2-SNAPSHOT, 2025-09-18T17:15:06Z

Table of Contents

This section explains how Quantum evaluates permissions for REST requests using rule bases that match on URL, HTTP method, headers, and request body content. It also covers how identities and roles (as found on userProfile or credentialUserIdPassword) are matched, how priority works, and how multiple matching rule bases are evaluated.

Note: The terms SecurityURLHeaders and SecurityURLBody in this document describe the matching dimensions for rules. Implementations may vary, but the semantics below are stable for authoring and reasoning about permissions.

Chapter 1. Introduction: Layered Enforcement Overview

Quantum evaluates "can this identity do X?" through three complementary layers. Understanding them in order helps you pick the right tool for the job and combine them safely:

1. REST API annotations (top layer, code-level)
    - What: JAX-RS/Jakarta Security annotations on resource methods, for example, `@RolesAllowed("ADMIN")`, `@PermitAll`, `@DenyAll`, `@Authenticated`.
    - Purpose: Coarse-grained, immediate gates right at the endpoint. Ideal for baseline protections (for example, only ADMIN may call `/admin/**`) and for non-dynamic constraints that rarely change.
    - Pros: Simple, fast, visible in code reviews.
    - Cons: Hard-coded; changing access requires a code change, build, and deploy. No data-aware scoping (for example, cannot express tenant/domain filters).
2. Feature flags (exposure and variants)
    - What: Turn capabilities on/off per environment or cohort and select variants (A/B, multivariate). See "Feature Flags, Variants, and Target Rules" below.
    - Purpose: Control who even sees or can reach a capability during rollout (by tenant, role, geography, plan), independently of authorization. Flags answer "is the feature ON and which variant?"; they do not by themselves prove the caller is authorized.
    - Pros: Reversible, environment-aware, safe rollout and experimentation.
    - Cons: Not a substitute for authorization; must be paired with roles/permission rules for enforcement.
3. Permission Rules with DataDomain filtering (fine-grained, dynamic)
    - What: Declarative rule bases that match URL, method, SecurityURLHeaders, and SecurityURLBody, and decide ALLOW/DENY. ALLOWs can contribute DataDomain constraints applied by repositories. See sections "Key Concepts", "Matching Algorithm", and examples below.
    - Purpose: Express least-privilege, data-aware policies that evolve without code changes (data-driven authoring).
    - Pros: Dynamic, auditable, supports role checks plus attribute predicates and domain scoping.
    - Cons: Requires governance of rulebases and careful priority management.

How roles, Functional Areas, and Functional Domains fit in

- Roles: Used by both layers (1) and (3). Annotations directly reference roles; rules can require rolesAny/rolesAll. Effective roles are resolved by merging IdP roles with roles on the user record; see "How roles are defined for an identity" below.
- Functional Area/Domain: Derived from the URL convention

`/{area}/{functionalDomain}/{action}` as parsed by `SecurityFilter.determineResourceContext`. Author policies using these fields to target business capabilities rather than raw URLs or ad-hoc headers.

- DataDomain: When rules ALLOW an action, they can attach data-scope filters (tenant/org/owner) so downstream reads/writes are constrained to the caller's domain.

Choosing the right approach

- Use annotations for stable, coarse gates you want visible in code (for example, admin-only endpoints, health endpoints with `@PermitAll`).
- Use feature flags to manage rollout/exposure and variants across environments and cohorts.
- Use permission rules to encode fine-grained, data-aware authorization and to evolve policy without redeploying.

Compare and contrast

- Annotation-based controls are compile-time and hard-code policy into the service; changing them requires code changes.
- Permission Rules and the Rule Language are data-driven and user-changeable (with proper governance), enabling rapid, auditable policy changes and DataDomain scoping.
- In practice: apply annotations as the first gate, evaluate feature flags to determine exposure/variant, then evaluate permission rules to decide ALLOW/DENY and attach scopes. This layered approach yields both safety and agility.

Chapter 2. Key Concepts

- Identity: The authenticated principal, typically originating from JWT or another provider. It includes:
  - userId (or credentialUserIdPassword username)
  - roles (authorities/groups)
  - tenantId, orgRefName, optional realm, and other claims that contribute to DomainContext
- userProfile: A domain representation of the user that adds human information such as first name, last name, email address, phone number and provides a linkage back to identity, roles, and policy decorations (feature flags, plans, expiration, etc.).
- Rule Base (Permission Rule): A declarative rule with matching criteria and an effect (ALLOW or DENY). Criteria may include:
  - HTTP method and URL pattern
  - SecurityURLHeaders: predicates over selected HTTP headers (for example, `X-Tenant-Id`). Note: functional area/domain/action are derived from the URL, not headers.
  - SecurityURLBody: predicates over request body fields (JSON paths) or query parameters
  - Required roles/attributes on identity or userProfile
  - Functional area/domain/action
  - Priority: integer used to sort rule evaluation
  - Effect: ALLOW or DENY; an ALLOW may also contribute scope filters (e.g., DataDomain constraints) to be applied downstream by repositories.

Chapter 3. Rule Structure (Illustrative)

```
- name: allow-public-reads
  priority: 100
  match:
    method: [GET]
    url: /Catalog/**
    rolesAny: [USER, ADMIN]
  effect: ALLOW
  filters:
    # Optional: contribute additional DataDomain filters
    readScope: { orgRefName: PUBLIC }

- name: deny-non-admin-delete
  priority: 10
  match:
    method: [DELETE]
    url: /api/**
  requireRolesAll: [ADMIN]
  effect: ALLOW

- name: default-deny
  priority: 10000
  match: {}
  effect: DENY
```

- headers under match are the SecurityURLHeaders predicates. Functional area/domain/action are parsed from the URL convention `/{area}/{functionalDomain}/{action}` (see `SecurityFilter.determineResourceContext`).
- Body predicates (SecurityURLBody) can be expressed similarly as JSONPath-like constraints:

```
  body:
    $.dataDomain.tenantId: ${identity.tenantId}
    $.action: [CREATE, UPDATE]
```

Chapter 4. Matching Algorithm

Given a request R and identity I, evaluate a set of rule bases RB as follows:

1. Candidate selection
    - From RB, select all rules whose URL pattern and HTTP method match R.
2. Attribute and header/body checks
    - For each candidate, check:
    - SecurityURLHeaders: header predicates must all match (case-insensitive header names; values support exact string, regex, or one-of lists depending on rule authoring capability).
    - SecurityURLBody: if present, evaluate body predicates against parsed JSON body (or query params when body is absent). Predicates must all match.
    - Identity/UserProfile: role requirements and attribute requirements must be satisfied.
3. Priority sort
    - Sort matching candidates by ascending priority (lower numbers indicate higher precedence). If not specified, default priority is 1000.
4. Evaluation order and decision
    - Iterate in sorted order; the first rule that yields a decisive effect (ALLOW or DENY) becomes the decision.
    - If the rule is ALLOW and contributes filters (e.g., DataDomain read/write scope), attach those to the request context for downstream repositories.
5. Multi-match aggregation (optional advanced mode)
    - In advanced configurations, if multiple ALLOW rules match at the same priority, their filters may be merged (intersection for restrictive scope, union for permissive scope) according to a configured merge strategy. If not configured, the default is first-match-wins.
6. Fallback
    - If no rules match decisively, apply a default policy (typically DENY).

Chapter 5. Priorities

- Lower integer = higher priority. Example: priority 1 overrides priority 10.
- Use tight scopes with low priority for critical protections (e.g., denies), and broader ALLOWs with higher numeric priority.
- Recommended ranges:
- 1–99: global deny rules and emergency blocks
- 100–499: domain/area-specific critical rules
- 500–999: standard ALLOW policies
- 1000+: defaults and catch-alls

Chapter 6. Grant-based vs Deny-based Rule Sets

Grant-based rule sets start with a default decision of DENY and then incrementally add ALLOW scenarios through explicit rules. This model is fail-safe by default: any URL, action, or functional area that does not have a matching ALLOW rule remains inaccessible. As new endpoints or capabilities are added to the system, users will not gain access until an explicit ALLOW is authored. This is the recommended posture for security-sensitive systems and multi-tenant platforms.

Deny-based rule sets start with a default decision of ALLOW and then add DENY scenarios to carve away disallowed cases. In this model, new functionality is exposed by default unless a DENY is added. While convenient during rapid prototyping, this posture risks accidental exposure as the surface area grows.

Practical implications:

- Change management: Grant-based requires adding ALLOWs when shipping new features; Deny-based requires remembering to add new DENYs.
- Auditability: Grant-based policies make it easy to enumerate what is permitted; Deny-based requires proving the absence of permissive gaps.
- Safety: In merge conflicts or partial deployments, Grant-based tends to fail closed (DENY), which is usually safer.

Example defaults:

- Grant-based (recommended):

```
- name: default-deny
  priority: 10000
  match: {}
  effect: DENY
```

- Deny-based (use with caution):

```
- name: default-allow
  priority: 10000
  match: {}
  effect: ALLOW
```

Tip: Even in a deny-based set, author low-number DENY rules for critical protections. In most production systems, prefer the grant-based model and layer specific ALLOWs for each capability.

Chapter 7. Feature Flags, Variants, and Target Rules

Feature flags complement permission rules by controlling whether a capability is active for a given principal, cohort, or environment. Permissions answer “may this identity perform this action?”; feature flags answer “is this capability turned on, and which variant applies?” Use them together to achieve safe rollouts and fine-grained authorization.

Model reference: com.e2eq.framework.model.general.FeatureFlag with key fields:

- enabled: master on/off
- type: BOOLEAN or MULTIVARIATE
- variants: list of variant keys for multivariate experiments
- targetRules: cohort targeting rules
- environment: e.g., dev, staging, prod
- jsonConfiguration: arbitrary configuration for the feature (e.g., rollout %, UI copy, limits)

Example: Boolean flag for a new export API with environment-specific targeting

```
{
  "refName": "EXPORT_API",
  "description": "Enable CSV export endpoint",
  "enabled": true,
  "type": "BOOLEAN",
  "environment": "prod",
  "targetRules": [
    { "attribute": "role",     "operator": "equals",  "values": ["BETA"] },
    { "attribute": "tenantId", "operator": "in",      "values": ["T100", "T200"] }
  ],
  "jsonConfiguration": { "rateLimitPerMin": 60 }
}
```

Example: Multivariate flag to roll out Search v2 to 10% of users and all members of a beta role

```
{
  "refName": "SEARCH_V2",
  "description": "New search implementation",
  "enabled": true,
  "type": "MULTIVARIATE",
  "variants": ["control", "v2"],
  "environment": "prod",
  "targetRules": [
    { "attribute": "role", "operator": "equals", "values": ["BETA"], "variant": "v2"
},
    { "attribute": "userId", "operator": "hashMod", "values": ["10"], "variant": "v2"
```

```
}
  ],
  "jsonConfiguration": { "defaultVariant": "control" }
}
```

Notes on TargetRules:

- attribute: a property from identity/userProfile (e.g., userId, role, tenantId, location, plan).
- operator: equals, in, contains, startsWith, regex, or domain-specific operators like hashMod for percentage rollouts.
- values: comparison values; semantics depend on operator.
- variant: when type is MULTIVARIATE, selects which variant applies when the rule matches.

How feature flags complement Permission Rule Context:

- The evaluation of a request can enrich the Rule Context (SecurityURLHeaders/Body or userProfile) with resolved feature flags and variants (e.g., userProfile.features["SEARCH_V2"] = "v2").
- Permission rules can then require a feature to be present before ALLOWing an action:

```
- name: allow-export-when-flag-on
  priority: 300
  match:
    method: [GET]
    url: /Reports/Export/**
    # Example predicate that assumes features are surfaced in userProfile
    userProfile.features.EXPORT_API: [true]
  rolesAny: [ADMIN, REPORTER]
  effect: ALLOW
```

Alternatively, systems may surface feature decisions via headers (e.g., X-Feature-SEARCH_V2: v2) so that SecurityURLHeaders can match directly.

Business usage examples for TargetRules and their correlation to Permission Rules:

- Progressive rollout by tenant TargetRule tenantId in [T100, T200] → Permission adds ALLOW for endpoints guarded by that flag so only those tenants can call them during rollout.
  - Role-based beta access: TargetRule role equals BETA → Permission requires both the BETA feature flag and standard role checks (e.g., USER/ADMIN) to ALLOW sensitive actions.
  - Plan/entitlement tiers: TargetRule plan in [Pro, Enterprise] → Permission rules enforce additional data-domain constraints (e.g., export size limits) while the flag simply turns the feature on for eligible plans.

Guidance: Feature Flags vs Permission Rules

- Put into Feature Flags:

- Gradual, reversible rollouts; A/B or multivariate experiments; UI/behavior switches.
- Environment gates (dev/staging/prod) and cohort targeting (tenants, beta users, geography).
- Non-security configuration values in jsonConfiguration (limits, thresholds, copy) that do not change who is authorized.

- Put into Permission Rules:
  - Durable authorization logic: roles, identities, functional area/domain/action, and DataDomain constraints.
  - Compliance and least-privilege decisions where fail-closed behavior is required.
  - Enforcement that remains valid after a feature is fully launched (even when the flag is removed).

    Recommendation

    Use a grant-based permission posture (default DENY) and let feature flags decide which cohorts even see or can reach new capabilities. Then author explicit ALLOW rules for those capabilities, conditioned on both role and feature presence.

Chapter 8. Multiple Matching RuleBases

- First-match-wins (default): after sorting by priority, the first decisive rule determines the result; subsequent matches are ignored.
- Merge strategy (optional):
- When enabled and multiple ALLOW rules share the same priority, scopes/filters are merged.
- Conflicts between ALLOW and DENY at the same priority resolve to DENY unless explicitly configured otherwise (fail-safe).

Chapter 9. Identity and Role Matching

- RolesAny: request is allowed if identity has at least one of the specified roles.
- RolesAll: request requires all listed roles.
- Attribute predicates can compare identity/userProfile attributes (e.g., identity.tenantId == header.x-tenant-id).
- Time or plan-based conditions: userProfile can embed plan and expiration; rules may check that trials are active or features are enabled.

9.1. How roles are defined for an identity (role sources and resolution)

Quantum composes the effective roles for a request by merging:

- Roles from the identity provider (JWT/`SecurityIdentity`)
- Roles configured on the user record (`CredentialUserIdPassword.roles`)

Source details:

- Identity Provider (JWT): roles commonly arrive via standard claims (for example, `groups`, `roles`, or provider-specific fields). Quarkus maps these into `SecurityIdentity.getRoles()`. In multi-realm setups, the realm in `X-Realm` can scope lookups but does not alter what the JWT asserts.
- Quantum user record: `com.e2eq.framework.model.security.CredentialUserIdPassword` has a `String[] roles` field stored per realm. This can be administered by Quantum to grant platform- or tenant-level roles.

Merge semantics (current implementation):

- Union: the effective role set is the union of JWT roles and `CredentialUserIdPassword.roles`. If either source is empty, the other source defines the set.
- Fallback: when neither source yields roles, the framework defaults to `ANONYMOUS`.
- Where implemented: `SecurityFilter.determinePrincipalContext` builds `PrincipalContext` with the merged roles.

Realm considerations:

- The user record is looked up by subject or userId in the active realm (default or `X-Realm`). If a realm override is provided, it is validated with `CredentialUserIdPassword.realmRegEx`.
- Roles stored in a user record are realm-specific; JWT roles are whatever the IdP asserts for the token.

Operating models:

- Quantum-managed roles:
  - IdP authenticates the user (subject, username). Authorization is primarily driven by roles

stored in `CredentialUserIdPassword.roles`.

    - Use when you want central, auditable role assignment within Quantum, independent of IdP groups.

- IdP-managed roles:
    - IdP carries authoritative roles/groups in the JWT. Keep `CredentialUserIdPassword.roles` minimal or empty.
    - Use when enterprises require IdP as the source of truth for access groups.
- Hybrid (recommended in many deployments):
    - Effective roles = JWT roles `CredentialUserIdPassword.roles`.
    - Use JWT for enterprise groups (for example, `DEPT_SALES`, `ORG_ADMIN`) and Quantum roles for app-specific grants (for example, `REPORT_EXPORTER`, `BETA`).
    - This avoids IdP churn for application-local concerns while respecting org policies.

Examples:

- JWT-only:
    - JWT.groups = [USER, REPORTER]; user record roles = []
    - Effective roles = [USER, REPORTER]
- Quantum-only:
    - JWT.groups = []; user record roles = [USER, ADMIN]
    - Effective roles = [USER, ADMIN]
- Hybrid union:
    - JWT.groups = [USER]; user record roles = [BETA, REPORT_EXPORTER]
    - Effective roles = [USER, BETA, REPORT_EXPORTER]

Guidance and best practices:

- Keep role names stable and environment-agnostic; use realms/permissions to scope where needed.
- Avoid overloading roles for feature rollout; use Feature Flags for rollout and variants, and roles for durable authorization.
- When IdP is authoritative, ensure consistent claim mapping so `SecurityIdentity.getRoles()` contains the expected values; commonly via `groups` claim in JWT.
- Use grant-based permission rules and require the minimal set of roles (`rolesAny`/`rolesAll`) needed for each capability.

Cross-references:

- User model: `com.e2eq.framework.model.security.CredentialUserIdPassword.roles`
- Context: `com.e2eq.framework.model.securityrules.PrincipalContext.getRoles()`
- Filter logic: `com.e2eq.framework.rest.filters.SecurityFilter.determinePrincipalContext`

Chapter 10. Example Scenarios

1. Public catalog browsing
    - Request: GET /Catalog/Products/VIEW?search=widgets
    - Identity: anonymous or role USER
    - Rules:
        - allow-public-reads (priority 100) ALLOW + readScope orgRefName=PUBLIC
    - Outcome: ALLOW; repository applies DataDomain filter orgRefName=PUBLIC
2. Tenant-scoped shipment update
    - Request: PUT /Collaboration/Shipments/UPDATE
    - Headers: x-tenant-id=T1
    - Body: { dataDomain: { tenantId: "T1" }, … }
    - Identity: user in tenant T1 with roles [USER]
    - Rules:
        - allow-collab-update (priority 300) requires body.dataDomain.tenantId == identity.tenantId and rolesAny USER, ADMIN ⇒ ALLOW
    - Outcome: ALLOW; Rule contributes writeScope tenantId=T1
3. Cross-tenant admin read with higher priority
    - Request: GET /api/partners
    - Identity: role ADMIN (super-admin)
    - Rules:
        - admin-override (priority 50) ALLOW
        - default-tenant-read (priority 600) ALLOW with tenant filter
    - Outcome: admin-override wins due to higher precedence (lower number), allowing broader read
4. Conflicting ALLOW and DENY at same priority
    - Two rules match with priority 200: one ALLOW, one DENY
    - Resolution: DENY wins unless merge strategy configured to handle explicitly; recommended to avoid same-priority conflicts by policy.

Chapter 11. Operational Tips

- Author specific DENY rules with low numbers to prevent accidental exposure.
- Keep URL patterns narrowly tailored for sensitive domains.
- Prefer header/body predicates to refine matches without exploding URL patterns.
- Log matched rule names and applied scopes for auditability.

Chapter 12. How UIActions and DefaultUIActions are calculated

When the server returns a collection of entities (for example, userProfiles), each entity may expose two action lists:

- DefaultUIActions: the full set of actions that conceptually apply to this type of entity (e.g., CREATE, UPDATE, VIEW, DELETE, ARCHIVE). Think of this as the “menu template” for the type.
- UIActions: the subset of actions the current user is actually permitted to perform on that specific entity instance right now.

Why they can differ per entity:

- Entity attributes: state or flags (e.g., archived, soft-deleted, immutable) can remove or alter available actions at instance level.
- Permission rule base: evaluated against the current request, identity, and context to allow or deny actions.
- DataDomain membership: tenant/org/owner scoping can further restrict actions if the identity is outside the entity’s domain.

How the server computes them:

1. Start with a default action template for the entity type (DefaultUIActions).
2. Apply simple state-based adjustments (for example, suppress CREATE on already-persisted instances).
3. Evaluate the permission rules with the current identity and context:
    - Consider roles, functional area/domain, action intent, headers/body, and any rule-contributed scopes.
    - Resolve DataDomain constraints to ensure the identity is permitted to act within the entity’s domain.
4. Produce UIActions as the allowed subset for that entity instance.
5. Return both lists with each entity in collection responses.

How the client should use the two lists:

- Render the full DefaultUIActions as the visible set of possible actions (icons, buttons, menus) so the UI stays consistent.
- Enable only those actions present in UIActions; gray out or disable the remainder to signal capability but lack of current permission.
- This approach avoids flicker and keeps affordances discoverable while remaining truthful to the user’s current authorization.

Example:

- You fetch 25 userProfiles.
- DefaultUIActions for the type = [CREATE, VIEW, UPDATE, DELETE, ARCHIVE].
- For a specific profile A (owned by your tenant), UIActions may be [VIEW, UPDATE] based on your roles and domain.
- For another profile B (in a different tenant), UIActions may be [VIEW] only.
- The UI renders the same controls for both A and B, but only enables the actions present in each item's UIActions list.

Operational considerations:

- Keep action names stable and documented so front-ends can map to icons and tooltips consistently.
- Prefer small, composable rules that evaluate action permissions explicitly by functional area/domain to avoid surprises.
- Consider server-side caching of action evaluations for list views to reduce latency, respecting identity and scope.

Chapter 13. How This Integrates End-to-End

- BaseResource extracts identity and headers to construct DomainContext.
- Rule evaluation uses URL/method + SecurityURLHeaders + SecurityURLBody + identity/userProfile to reach a decision and derive scope filters.
- Repositories (e.g., MorphiaRepo) apply the filters to queries and updates, ensuring DataDomain-respecting access.

Chapter 14. Administering Policies via REST (PolicyResource)

The PolicyResource exposes CRUD-style REST APIs for creating and managing policies (rule bases) that drive authorization decisions. Each Policy targets a principalId (either a specific userId or a role name) and contains an ordered list of Rule objects. Rules match requests using SecurityURIHeader and SecurityURIBody and then contribute an effect (ALLOW/DENY) and optional repository filters.

- Base path: /security/permission/policies
- Auth: Bearer JWT (see Authentication); resource methods are guarded by @RolesAllowed("user", "admin") at the BaseResource level and your own realm/role policies.
- Multi-realm: pass X-Realm header to operate within a specific realm; otherwise the default realm is used.

14.1. Model shape (Policy)

A Policy extends FullBaseModel and includes: - id, refName, displayName, dataDomain, archived/expired flags (inherited) - principalId: userId or role name that this policy attaches to - description: human-readable summary - rules: array of Rule entries

Rule fields (key ones): - name, description - securityURI.header: identity, area, functionalDomain, action (supports wildcard "") - **securityURI.body: realm, orgRefName, accountNumber, tenantId, ownerId, dataSegment, resourceId (supports wildcard ""**) - effect: ALLOW or DENY - priority: integer; lower numbers evaluated first - finalRule: boolean; stop evaluating when this rule applies - andFilterString / orFilterString: ANTLR filter DSL snippets injected into repository queries (see Query Language section)

Example payload:

```
{
  "refName": "defaultUserPolicy",
  "displayName": "Default user policy",
  "principalId": "user",
  "description": "Users can act on their own data; deny dangerous ops in security
area",
  "rules": [
    {
      "name": "view-own-resources",
      "description": "Limit reads to owner and default data segment",
      "securityURI": {
        "header": { "identity": "user", "area": "*", "functionalDomain": "*", "action
": "*" },
        "body":   { "realm": "*", "orgRefName": "*", "accountNumber": "*", "tenantId":
"*", "ownerId": "*", "dataSegment": "*", "resourceId": "*" }
      },
```

```
      "andFilterString": "
dataDomain.ownerId:${principalId}&&dataDomain.dataSegment:#0",
      "effect": "ALLOW",
      "priority": 300,
      "finalRule": false
    },
    {
      "name": "deny-delete-in-security",
      "securityURI": {
        "header": { "identity": "user", "area": "security", "functionalDomain": "*",
"action": "delete" },
        "body":   { "realm": "*", "orgRefName": "*", "accountNumber": "*", "tenantId":
"*", "ownerId": "*", "dataSegment": "*", "resourceId": "*" }
      },
      "effect": "DENY",
      "priority": 100,
      "finalRule": true
    }
  ]
}
```

14.2. Endpoints

All endpoints are relative to /security/permission/policies. These are inherited from BaseResource and are consistent across entity resources.

- GET /list
  - Query params: skip, limit, filter, sort, projection
  - Returns a Collection<Policy> with paging metadata; respects X-Realm.
- GET /id/{id} and GET /id?id=...
  - Fetch a single Policy by id.
- GET /refName/{refName} and GET /refName?refName=...
  - Fetch a single Policy by refName.
- GET /count?filter=...
  - Returns a CounterResponse with total matching entities.
- GET /schema
  - Returns JSON Schema for Policy.
- POST /
  - Create or upsert a Policy (if id is present and matches an existing entity in the selected realm, it is updated).
- PUT /set?id=...&pairs=field:value
  - Targeted field updates by id. pairs is a repeated query parameter specifying field/value pairs.

- PUT /bulk/setByQuery?filter=...&pairs=...
  - Bulk updates by query. Note: ignoreRules=true is not supported on this endpoint.
- PUT /bulk/setByIds
  - Bulk updates by list of ids posted in the request body.
- PUT /bulk/setByRefAndDomain
  - Bulk updates by a list of (refName, dataDomain) pairs in the request body.
- DELETE /id/{id} (or /id?id=...)
  - Delete by id.
- DELETE /refName/{refName} (or /refName?refName=...)
  - Delete by refName.
- CSV import/export endpoints for bulk operations:
  - GET /csv – export as CSV (field selection, encoding, etc.)
  - POST /csv – import CSV into Policies
  - POST /csv/session – analyze CSV and create an import session (preview)
  - POST /csv/session/{sessionId}/commit – commit a previously analyzed session
  - DELETE /csv/session/{sessionId} – cancel a session
  - GET /csv/session/{sessionId}/rows – page through analyzed rows
- Index management (admin only):
  - POST /indexes/ensureIndexes/{realm}?collectionName=policy

Headers:

- Authorization: Bearer <token>
- X-Realm: realm identifier (optional but recommended in multi-tenant deployments)

Filtering and sorting:

- filter uses the ANTLR-based DSL (see REST CRUD > Query Language)
- sort uses comma-separated fields with optional +/- prefix; projection accepts a comma-separated field list

14.3. Examples

- Create or update a Policy

```
curl -X POST \
  -H "Authorization: Bearer $JWT" \
  -H "Content-Type: application/json" \
  -H "X-Realm: system-com" \
  https://host/api/security/permission/policies \
```

```
  -d @policy.json
```

- List policies for principalId=user

```
curl -H "Authorization: Bearer $JWT" \
     -H "X-Realm: system-com" \

"https://host/api/security/permission/policies/list?filter=principalId:'user'&sort=+re
fName&limit=50"
```

- Delete a policy by refName

```
curl -X DELETE \
  -H "Authorization: Bearer $JWT" \
  -H "X-Realm: system-com" \
  "https://host/api/security/permission/policies/refName/defaultUserPolicy"
```

14.4. How changes affect rule bases and enforcement

- Persistence vs. in-memory rules:
- PolicyResource updates the persistent store of policies (one policy per principalId or role with a list of rules).
- RuleContext is the in-memory evaluator used by repositories and resources to enforce permissions. It matches SecurityURIHeader/Body, orders rules by priority, and applies effects and filters.
- Making persisted policy changes effective:
- On startup, migrations (see InitializeDatabase and AddAnonymousSecurityRules) typically seed default policies and/or programmatically add rules to RuleContext.
- When you modify policies via REST, you have two options to apply them at runtime:
  1. Implement a reload step that reads policies from PolicyRepo and rehydrates RuleContext (for example, RuleContext.clear(); then add rules built from current policies).
  2. Restart the service or trigger whatever policy-loader your application uses at boot.
- Tip: If you maintain a background watcher or admin endpoint to refresh policies, keep it tenant/realm-aware and idempotent.
- Evaluation semantics (recap):
- Rules are sorted by ascending priority; the first decisive rule sets the outcome. finalRule=true stops further processing.
- andFilterString/orFilterString contribute repository filters through RuleContext.getFilters(), constraining result sets and write scopes.
- principalId can be a concrete userId or a role; RuleContext considers both the principal and all associated roles.

- Safe rollout:
- Create new policies with a higher numeric priority (lower precedence) first, test with GET /schema and dry-run queries.
- Use realm scoping via X-Realm to stage changes in a non-production realm.
- Prefer DENY with low priority numbers for critical protections.

See also: - Permissions: Matching Algorithm, Priorities, and Multiple Matching RuleBases (sections above) - REST CRUD: Query Language and generic endpoint behaviors

Chapter 15. Realm override (X-Realm) and Impersonation (X-Impersonate)

This section explains how to use the request headers X-Realm and X-Impersonate-* alongside permission rule bases. These headers influence which realm (database) a request operates against and, in the case of impersonation, which identity's roles are evaluated by the rule engine.

15.1. What they do (at a glance)

- X-Realm: Overrides the target realm (MongoDB database) used by repositories for this request. Your own identity and roles remain the same; only the data context (tenant/realm) changes for this call. This lets you "switch tenants" at the database level in deployments that use the one-tenant-per-database model.
- X-Impersonate-Subject or X-Impersonate-UserId: Causes the request to run as another identity. The effective permissions become those of the impersonated identity (potentially more or less than your own). This is analogous to sudo on Unix or to "simulate a user/role" for troubleshooting.

Only one of X-Impersonate-Subject or X-Impersonate-UserId may be supplied per request. Supplying both results in a 400/IllegalArgumentException.

15.2. How the headers integrate with permission evaluation

- Rule matching and effects (ALLOW/DENY) still follow the standard algorithm described earlier.
- With X-Realm (no impersonation):
- The PrincipalContext.defaultRealm is set to the header value (after validation), and repositories operate in that realm.
- Your own roles and identity remain intact; the rule base is evaluated for your identity and roles but in the specified realm's data context.
- With impersonation:
- The PrincipalContext is rebuilt from the impersonated user's credential. The effective roles used by the rule engine include the impersonated user's roles; the platform also merges in the caller's security roles from Quarkus SecurityIdentity. This means permissions can be a superset; design policy rules accordingly.
- The effective realm for the request is set to the impersonated user's default realm (not the X-Realm header). If you passed X-Realm, it is still validated (see below) but not used to override the impersonated default realm in the current implementation.

15.3. Required credential configuration (CredentialUserIdPassword)

Two fields on CredentialUserIdPassword govern whether a user may use these headers:

- realmRegEx (for X-Realm):
- A wildcard pattern ("*" matches any sequence; case-insensitive) listing the realms a user is allowed to target with X-Realm.
- If X-Realm is present but realmRegEx is null/blank or does not match the requested realm, the server returns 403 Forbidden.
- Examples:
- "*" → allow any realm
- "acme-*" → allow realms that start with acme-
- "dev|stage|prod" is not supported as-is; use wildcards like "dev*" and "stage*" or a combined pattern like "(dev|stage|prod)" only if you store a true regex. The current validator replaces "**with "."** and matches case-insensitively.
- impersonateFilterScript (for X-Impersonate-*):
- A JavaScript snippet executed by the server (GraalVM) that must return a boolean. It receives three variables: username (the caller's subject), userId (caller's userId), and realm (the requested realm or current DB name).
- If the script evaluates to false, the server returns 403 Forbidden for impersonation.
- If the script is missing (null) and you attempt impersonation, the server rejects the request with 400/IllegalArgumentException.

Example impersonation script (allow only company admins to impersonate in dev realms):

```
// username = caller's subject, userId = caller's userId, realm = requested realm (or
current)
(username.endsWith('@acme.com') && realm.startsWith('dev-'))
```

Tip: Manage these two fields via your auth provider's admin APIs or directly through CredentialRepo in controlled environments.

15.4. End-to-end behavior from SecurityFilter (reference)

The SecurityFilter constructs the PrincipalContext/ResourceContext before rule evaluation: - X-Realm is read and, if present, validated against the caller's credential.realmRegEx. - If impersonation headers are present: - The caller's credential.impersonateFilterScript is executed. If it returns true, the impersonated user's credential is loaded and used to build the PrincipalContext. - The final PrincipalContext carries the impersonated user's defaultRealm and roles (merged with the caller's SecurityIdentity roles), and may copy area2RealmOverrides from the impersonated

credential. - Without impersonation, the PrincipalContext is built from the caller's credential; X-Realm, when valid, sets the defaultRealm for this request.

15.5. Practical differences and use cases

- Realm override (X-Realm):
- Who you are does not change; only where you act changes. Your permissions (as determined by policies attached to your identity/roles) are applied against data in the specified realm.
- Use cases:
- Multi-tenant admin tooling that needs to inspect or repair data in customer realms.
- Reporting or backfills where the same service is pointed at different tenant databases per request.
- Impersonation (X-Impersonate-*):
- Who you are (for authorization purposes) changes. You act with the impersonated identity's permissions; depending on your configuration, additional caller roles may be merged.
- Use cases:
- Temporary elevation to an admin identity (sudo-like) for break-glass operations.
- Simulate what a given role/identity can see/do for troubleshooting or customer support.

Caveats: - Never set a permissive impersonateFilterScript in production. Keep it restrictive and auditable. - When using both X-Realm and impersonation in one call, be aware that the effective realm will be the impersonated user's default realm; X-Realm is not applied in the impersonation branch in the current implementation. - realmRegEx must be populated for any user who needs realm override; leaving it blank effectively disables X-Realm for that user.

15.6. Examples

- List policies in a different realm using your own identity

```
curl -H "Authorization: Bearer $JWT" \
     -H "X-Realm: acme-prod" \
     "https://host/api/security/permission/policies/list?limit=20&sort=+refName"
```

- Simulate another user by subject while staying in their default realm

```
curl -H "Authorization: Bearer $JWT" \
     -H "X-Impersonate-Subject: 3d8f4e7b-...-idp-subject" \
     "https://host/api/security/permission/policies/list?limit=20"
```

- Attempt impersonation with a realm hint (validated by script; effective realm = impersonated default)

```
curl -H "Authorization: Bearer $JWT" \
     -H "X-Realm: dev-acme" \
     -H "X-Impersonate-UserId: tenant-admin" \
     "https://host/api/security/permission/policies/list?limit=20"
```

Security outcomes in all cases continue to be driven by your rule bases (Policy rules) matched against the effective PrincipalContext and ResourceContext.

Chapter 16. Data domain assignment on create: DomainContext and DataDomainPolicy

This section explains how Quantum decides which dataDomain is stamped on newly created records, why this decision is necessary in a multi-tenant system, what the default behavior is, and how you can override it globally or per Functional Area / Functional Domain. It also describes the DataDomainResolver interface and the default implementation provided by the framework.

16.1. The problem this solves (and why it matters)

In a multi-tenant platform you must ensure each new record is written to the correct data partition so later reads/updates can be scoped safely. If the dataDomain is wrong or missing, you risk leaking data across tenants or making your own data inaccessible due to mis-scoping.

Historically, Quantum set the dataDomain of new entities to match the creator's credential (i.e., the principal's DomainContext → DataDomain). That default is sensible in many cases, but real systems often need more specific behavior per business area or type. For example: - You may centralize HR records in a single org-level domain regardless of who created them. - Sales invoices for EU customers must live under an EU data segment. - A specific product area might always write into a shared catalog domain separate from the author's tenant.

These needs require a simple, deterministic way to override the default per Functional Area and/or Functional Domain.

16.2. Key concepts recap: DomainContext and DataDomain

DomainContext (on credentials/realms)

captures the principal's scoping defaults (realm, org/account/tenant identifiers, data segment). At request time this is materialized into a DataDomain.

DataDomain

is what gets stamped onto persisted entities and later used by repositories to constrain queries and updates.

If you do nothing, new records inherit the principal's DataDomain.

16.3. The default policy (do nothing and it works)

Out of the box, Quantum preserves the existing behavior: if no policy is configured, the resolver falls back to the authenticated principal's DataDomain. This guarantees compatibility with existing applications.

Concretely: - ValidationInterceptor checks if an entity being persisted lacks a dataDomain. - If

missing, it calls DataDomainResolver.resolveForCreate(area, domain). - The DefaultDataDomainResolver first looks for overrides (credential-attached or global); if none match, it returns the principal's DataDomain from the current SecurityContext.

16.4. Policy scopes: principal-attached vs. global

You can define overrides at two levels: - Principal-attached (per credential): attach a DataDomainPolicy to a CredentialUserIdPassword. The SecurityFilter places this policy into the PrincipalContext, so it applies only to records created by that principal. This is useful for VIP service accounts or specific partners. - Global policy: an application-wide DataDomainPolicy provided by GlobalDataDomainPolicyProvider. If present, this applies when the principal has no specific override for the matching area/domain.

Precedence: principal-attached policy wins over global policy; if neither applies, fall back to the principal's credential domain.

16.5. The policy map and matching

A DataDomainPolicy is a small map of rules: Map<String, DataDomainPolicyEntry> policyEntries, keyed by "<FunctionalArea>:<FunctionalDomain>" with support for "*" wildcards. The resolver evaluates keys in this order:

1. area:domain (most specific)
2. area:*
3. *:domain
4. : (global catch-all)
5. Fallback to principal's domain if no entry yields a value

Each DataDomainPolicyEntry has a resolutionMode: - FROM_CREDENTIAL (default): use the principal's credential domain (i.e., the historical behavior). - FIXED: use the first DataDomain listed in dataDomains on the entry.

Example policy definitions (illustrative JSON):

```
{
  "policyEntries": {
    "Sales:Invoice": { "resolutionMode": "FIXED", "dataDomains": [ {"orgRefName":
"ACME", "tenantId": "eu-1", "dataSegment": "INVOICE"} ] },
    "Sales:*":      { "resolutionMode": "FROM_CREDENTIAL" },
    "*:HR":         { "resolutionMode": "FIXED", "dataDomains": [ {"orgRefName":
"GLOBAL", "tenantId": "hr", "dataSegment": "HR"} ] },
    "*:*":          { "resolutionMode": "FROM_CREDENTIAL" }
  }
}
```

Behavior of the above: - Sales:Invoice records always go to the fixed EU invoices domain. - Any

other Sales:* creation uses the creator's credential domain. - All HR records go to a central HR domain. - Otherwise, default to the creator's domain.

16.6. How the resolver works

Interfaces and default implementation:

```
public interface DataDomainResolver {
  DataDomain resolveForCreate(String functionalArea, String functionalDomain);
}

@ApplicationScoped
public class DefaultDataDomainResolver implements DataDomainResolver {
  @Inject GlobalDataDomainPolicyProvider globalPolicyProvider;
  public DataDomain resolveForCreate(String area, String domain) {
    DataDomain principalDD = SecurityContext.getPrincipalDataDomain()
      .orElseThrow(() -> new IllegalStateException("Principal context not providing a
data domain"));
    List<String> keys = List.of(areaOrStar(area)+":"+areaOrStar(domain), areaOrStar
(area)+":*", "*:"+areaOrStar(domain), "*:*");
    // 1) principal attached policy from PrincipalContext
    DataDomain fromPrincipal = resolveFrom(policyFromPrincipal(), keys, principalDD);
    if (fromPrincipal != null) return fromPrincipal;
    // 2) global policy
    DataDomain fromGlobal = resolveFrom(globalPolicyProvider.getPolicy().orElse(null),
keys, principalDD);
    if (fromGlobal != null) return fromGlobal;
    // 3) default fallback
    return principalDD;
  }
}
```

Integration point: - ValidationInterceptor injects DataDomainResolver and calls it in prePersist when an entity's dataDomain is null. - SecurityFilter propagates a principal's attached DataDomainPolicy (if any) into the PrincipalContext so the resolver can see it.

16.7. When would you want a non-global policy?

Here are a few concrete scenarios: - Centralized HR: All HR Employee records are written to a shared HR domain regardless of the team creating them. This supports a shared-service HR model without duplicating HR data per tenant. - Regulated invoices: In the Sales:Invoice domain for EU, you must write under a specific EU tenantId/dataSegment to satisfy data residency. Other Sales domains can keep default behavior. - Shared catalog: The Catalog:Item domain is a cross-tenant shared catalog maintained by a core team. Writes should go to a canonical catalog domain even when initiated by tenant-specific users. - VIP account override: A particular integration user should always write to a staging domain for testing purposes, while all others use defaults. Attach a small policy to just that credential.

16.8. Relation to tenancy models

The policy mechanism supports both siloed and pooled tenancy: - Siloed tenancy: Most domains default to FROM_CREDENTIAL (each tenant writes to its own partition). Only a few shared services (e.g., HR, catalog) use FIXED to centralize data. - Pooled tenancy: You may lean on FIXED policies more often to route writes into pooled/segment-specific domains (e.g., region, product line), while still enforcing read/write scoping via permissions.

Because the resolver always validates through the principal context and falls back safely, you can introduce overrides gradually without destabilizing existing flows.

16.9. Authoring tips

- Start with no policy and verify your default flows. Add entries only where necessary.
- Prefer specific keys (area:domain) for clarity; use wildcards sparingly.
- Keep FIXED DataDomain objects minimal and valid for your deployment (orgRefName, tenantId, and dataSegment as needed).
- Document any global policy so teams know which areas are centralized.

16.10. API pointers

- CredentialUserIdPassword.dataDomainPolicy: optional per-credential overrides (propagated to PrincipalContext).
- GlobalDataDomainPolicyProvider: holds an optional in-memory global policy (null by default).
- DataDomainPolicyEntry.resolutionMode: FROM_CREDENTIAL (default) or FIXED.
- DataDomainResolver / DefaultDataDomainResolver: the extension point and default behavior.