Permissions

Rule Bases, SecurityURLHeaders, and SecurityURLBody

Version 1.2.2-SNAPSHOT, 2025-09-13T02:57:15Z

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. 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 aggregates 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 (e.g., x-functional-area, x-functional-domain, x-tenant-id)
- 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 2. Rule Structure (Illustrative)

```
- name: allow-public-reads
  priority: 100
  match:
    method: [GET]
    url: /api/catalog/**
    headers:
      x-functional-area: [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.
- Body predicates (SecurityURLBody) can be expressed similarly as JSONPath-like constraints:

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

Chapter 3. 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 4. 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 5. 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 6. 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.

Chapter 7. Example Scenarios

1) Public catalog browsing - Request: GET /api/catalog/products?search=widgets - Headers: x-functional-area=Catalog - 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 /api/shipments/ABC123 - Headers: x-functional-area=Collaboration, 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 8. 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 9. 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 (e.g., 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 10. 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 11. 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.

11.1. 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
    }
  ]
}
```

11.2. 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

11.3. 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"
```

11.4. 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 (e.g., 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