Table of Contents

1. REST: Find, Get, List, Save, Update, Delete

Quantum provides consistent REST resources backed by repositories. Extend BaseResource to expose CRUD quickly and consistently.

1.1. Base Concepts

- BaseResource<T, R extends Repo<T>> provides endpoints for:
- find: query by criteria (filters, pagination)
- get: fetch by id or refName
- list: list all within scope with paging
- save: create
- update: modify existing
- delete: delete or soft-delete/archival depending on model
- UIActionList: derive available actions based on current model state.
- DataDomain filtering is applied across all operations to enforce multi-tenancy.

1.2. Example Resource

```java
import com.e2eq.framework.rest.resources.BaseResource;
import jakarta.ws.rs.Path;

@Path("/products")
public class ProductResource extends BaseResource<Product, ProductRepo> {
}
```

1.3. Authorization Layers in REST CRUD

Quantum combines static, identity-based checks with dynamic, domain-aware policy evaluation. In practice you will often use both:

1) Hard-coded permissions via annotations

- Use standard Jakarta annotations like @RolesAllowed (or the framework's @RoleAllow if present) on resource classes or methods to declare role-based checks that must pass before executing an endpoint.
- These checks are fast and decisive. They rely on the caller's roles as established by the current SecurityIdentity.

Example:

```java
import jakarta.annotation.security.RolesAllowed;

@RolesAllowed({"ADMIN", "CATALOG_EDITOR"})
```

```
@Path("/products")
public class ProductResource extends BaseResource<Product, ProductRepo> {
    // Only ADMIN or CATALOG_EDITOR can access all inherited CRUD endpoints
}
```

2) JWT groups and role mapping

- When using the JWT provider, the token's groups/roles claims are mapped into the Quarkus SecurityIdentity (see the Authentication guide).
- Groups in JWT typically become roles on SecurityIdentity; these roles are what @RolesAllowed/@RoleAllow checks evaluate.
- You can augment or transform roles using a SecurityIdentityAugmentor (see RolesAugmentor in the framework) to add derived roles based on claims or external lookups.

3) RuleContext layered authorization (dynamic policies)

- After annotation checks pass, RuleContext evaluates domain-aware permissions. This layer can:
- Enforce DataDomain scoping (tenant/org/owner)
- Allow cross-tenant reads for specific functional areas when policy permits
- Contribute query predicates and projections to repositories
- Think of @RolesAllowed/@RoleAllow as the coarse-grained gate, and RuleContext as the fine-grained, context-sensitive policy engine.

4) Quarkus SecurityIdentity and SecurityFilter

- Quarkus produces a SecurityIdentity for each request containing principal name and roles.
- The framework's SecurityFilter inspects the incoming request (e.g., JWT) and populates/augments the SecurityIdentity and the derived DomainContext used by RuleContext and repositories.
- BaseResource and underlying repos (e.g., MorphiaRepo) consume SecurityIdentity/DomainContext to apply permissions and filters consistently.

For detailed rule-base matching (URL, headers, body predicates, priorities), see the Permissions section.

1.4. Querying

- Use query parameters or a request body (depending on your API convention) to express filters.
- RuleContext contributes tenant-aware filters and projections automatically.

1.5. Responses and Schemas

- Models are returned with calculated fields (e.g., actionList) when appropriate.
- OpenAPI annotations in your models/resources integrate with MicroProfile OpenAPI for schema docs.

1.6. Error Handling

- Validation errors (e.g., ImportRequiredField, Size) return helpful messages.
- Rule-based denials return appropriate HTTP statuses (403/404) without leaking cross-tenant metadata.

1.7. Query Language (ANTLR-based)

The find/list endpoints accept a filter string parsed by an ANTLR grammar (BIAPIQuery.g4). Use the filter query parameter to express predicates; combine them with logical operators and grouping. Sorting and projection are separate query parameters.

- Operators:
- Equals: ':'
- Not equals: ':!'
- Less than/Greater than: ':<' / ':>'
- Less-than-or-equal/Greater-than-or-equal: ':⇐' / ':>='
- Exists (field present): ':~' (no value)
- In list: ':^' followed by [v1,v2,…]
- Boolean literals: true/false
- Null literal: null
- Logical:
- AND: '&&'
- OR: '||'
- NOT: '!!' (applies to a single allowed expression)
- Grouping: parentheses '(' and ')'
- Values by type:
- Strings: unquoted or quoted with "…"; quotes allow spaces and punctuation
- Whole numbers: prefix with '#' (e.g., #10)
- Decimals: prefix with '' (e.g., 19.99)
- Date: yyyy-MM-dd (e.g., 2025-09-10)
- DateTime (ISO-8601): 2025-09-10T12:30:00Z (timezone supported)
- ObjectId (Mongo 24-hex): 5f1e9b9c8a0b0c0d1e2f3a4b
- Reference by ObjectId: @@5f1e9b9c8a0b0c0d1e2f3a4b
- Variables: ${ownerId|principalId|resourceId|action|functionalDomain|pTenantId|pAccountId|rTenantId|rAccountId|realm|area}

2. Simple filters (equals)

```
# string equality
name:"Acme Widget"
# whole number
quantity:#10
# decimal number
price:##19.99
# date and datetime
shipDate:2025-09-12
updatedAt:2025-09-12T10:15:00Z
# boolean
active:true
# null checks
description:null
# field exists
lastLogin:~
# object id equality
id:5f1e9b9c8a0b0c0d1e2f3a4b
# variable usage (e.g., tenant scoping)
dataDomain.tenantId:${pTenantId}
```

3. Advanced filters: grouping and AND/OR/NOT

```
# Products that are active and (name contains widget OR gizmo), excluding discontinued
active:true && (name:*widget* || name:*gizmo*) && status:!"DISCONTINUED"

# Shipments updated after a date AND (destination NY OR CA)
updatedAt:>=2025-09-01 && (destination:"NY" || destination:"CA")

# NOT example: items where category is not null and not (price < 10)
category:!null && !!(price:<##10)
```

Notes: - Wildcard matching uses ''**: name:*widget** (prefix/suffix/contains). '?' matches a single character. - Use parentheses to enforce precedence; otherwise AND/OR follow standard left-to-right with explicit operators.

4. IN lists

```
status:^["OPEN","CLOSED","ON_HOLD"]
ownerId:^["u1","u2","u3"]
referenceId:^[@@5f1e9b9c8a0b0c0d1e2f3a4b, @@6a7b8c9d0e1f2a3b4c5d6e7f]
```

5. Sorting

Provide a sort query parameter (comma-separated fields): - '-' prefix = descending, '+' or no prefix = ascending.

Examples:

```
# single field descending
?sort=-createdAt

# multiple fields: createdAt desc, refName asc
?sort=-createdAt,refName
```

6. Projections

Limit returned fields with the projection parameter (comma-separated): - '+' prefix = include, '-' prefix = exclude.

Examples:

```
# include only id and refName, exclude heavy fields
?projection=+id,+refName,-auditInfo,-persistentEvents
```

7. End-to-end examples

- GET /products/list?skip=0&limit=50&filter=active:true&&name:*widget*&sort=-updatedAt&projection=+id,+name,-auditInfo
- GET /shipments/list?filter=(destination:"NY"||destination:"CA")&&updatedAt:>=2025-09-01&sort=origin

These features integrate with RuleContext and DataDomain: your filter runs within the tenant/org scope derived from the security context; RuleContext may add further predicates or projections automatically.

Chapter 1. CSV Export and Import

These endpoints are inherited by every resource that extends BaseResource. They are mounted under the resource's base path. For example, PolicyResource at /security/permission/policies exposes: - GET /security/permission/policies/csv - POST /security/permission/policies/csv - POST /security/permission/policies/csv/session - POST /security/permission/policies/csv/session/{sessionId}/commit - DELETE /security/permission/policies/csv/session/{sessionId} - GET /security/permission/policies/csv/session/{sessionId}/rows

Authorization and scoping: - All CSV endpoints are protected by the same @RolesAllowed("user", "admin") checks as other CRUD operations. - RuleContext filters and DataDomain scoping apply the same way as list/find; exports stream only what the caller may see, and imports are saved under the same permissions. - In multi-realm deployments, include your X-Realm header as you do for CRUD; underlying repos resolve realm and domain context consistently.

1.1. Export: GET /csv

Produces a streamed CSV download of the current resource collection.

Query parameters and behavior:

fieldSeparator (default `"`)

Single character used to separate fields. Typical values: `,`, `;`, `\t`.

requestedColumns (default refName)

Comma-separated list of model field names to include, in output order. If omitted, BaseResource defaults to refName. Nested list extraction is supported with the `[0]` notation on a single nested property across all requested columns (e.g., `addresses[0].city`, `addresses[0].zip`). Indices other than `[0]` are rejected. If the nested list has multiple items, multiple rows are emitted per record (one per list element), preserving other column values.

quotingStrategy (default QUOTE_WHERE_ESSENTIAL)

- QUOTE_WHERE_ESSENTIAL: quote only when needed (when a value contains the separator or quoteChar).
- QUOTE_ALL_COLUMNS: quote every column in every row.

quoteChar (default `"`)

The character used to surround quoted values.

decimalSeparator (default `.`)

Reserved for decimal formatting. Note: current implementation ignores this value; decimals are rendered using the locale-independent dot.

charsetEncoding (default UTF-8-without-BOM)

One of: `US-ASCII`, `UTF-8-without-BOM`, `UTF-8-with-BOM`, `UTF-16-with-BOM`, `UTF-16BE`, `UTF-16LE`. "with-BOM" values write a Byte Order Mark at the beginning of the file (UTF-8: `EF BB BF`; UTF-16:

`FE FF`).

filter (optional)
: ANTLR DSL filter applied server-side before streaming (see Query Language section). Reduces rows and can improve performance.

filename (default downloaded.csv)
: Suggested download filename returned via Content-Disposition header.

offset (default 0)
: Zero-based index of the first record to stream.

length (default 1000, use `-1` for all)
: Maximum number of records to stream from offset. Use `-1` to stream all (be mindful of client memory/time).

prependHeaderRow (optional boolean, default false)
: When true, the first row contains column headers. Requires requestedColumns to be set (the default refName satisfies this requirement).

preferredColumnNames (optional list)
: Overrides header names positionally when `prependHeaderRow=true`. The list length must be ≤ requestedColumns; an empty string entry means "use default field name" for that column.

Response: - 200 OK with Content-Type: text/csv and Content-Disposition: attachment; filename="...". - On validation/processing errors, the response status is 400/500 and the body contains a single text line describing the problem (e.g., "Incorrect information supplied: ..."). Unrecognized query parameters are rejected with 400.

Examples:

- Export selected fields with header, custom filename and filter

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

"https://host/api/products/csv?requestedColumns=id,refName,price&prependHeaderRow=true
&filename=products.csv&filter=active:true&sort=+refName"
```

- Export nested list's first element across columns

```
# emits one row per address entry when more than one is present
curl -H "Authorization: Bearer $JWT" \

"https://host/api/customers/csv?requestedColumns=refName,addresses[0].city,addresses[0
].zip&prependHeaderRow=true"
```

1.2. Import: POST /csv (multipart)

Consumes a CSV file (multipart/form-data) and imports records in batches. The form field name for the file is file.

Query parameters and behavior:

fieldSeparator (default ")
: Single character expected between fields.

quotingStrategy (default QUOTE_WHERE_ESSENTIAL)
: Same values as export; controls how embedded quotes are recognized.

quoteChar (default ")
: The expected quote character in the file.

skipHeaderRow (default true)
: When true, the first row is treated as a header and skipped. Mapping is positional, not by header names.

charsetEncoding (default UTF-8-without-BOM)
: The file encoding. “with-BOM” variants allow consuming a BOM at the start.

requestedColumns (required)
: Comma-separated list of model field names in the same order as the CSV columns. This positional mapping drives parsing and validation. Nested list syntax `[0]` is allowed with the same constraints as export.

Behavior: - Each row is parsed into a model instance using type-aware processors (ints, longs, decimals, enums, etc.). - Bean Validation is applied; rows with violations are collected as errors and not saved; valid rows are batched and saved. - For each saved batch, insert vs update is determined by refName presence in the repository. - Response entity includes counts (importedCount, failedCount) and per-row results when available. - Response headers: - X-Import-Success-Count: number of rows successfully imported. - X-Import-Failed-Count: number of rows that failed validation or DB write. - X-Import-Message: summary message.

Example (direct import):

```
curl -X POST \
  -H "Authorization: Bearer $JWT" \
  -H "X-Realm: system-com" \
  -F "file=@policies.csv" \

"https://host/api/security/permission/policies/csv?requestedColumns=refName,principalI
d,description&skipHeaderRow=true&fieldSeparator=,&quoteChar=\"&quotingStrategy=QUOTE_W
HERE_ESSENTIAL&charsetEncoding=UTF-8-without-BOM"
```

1.3. Import with preview sessions

Use a two-step flow to analyze first, then commit only valid rows.

- POST /csv/session (multipart): analyzes the file and creates a session
- Same parameters as POST /csv (fieldSeparator, quotingStrategy, quoteChar, skipHeaderRow, charsetEncoding, requestedColumns).
- Returns a preview ImportResult including sessionId, totals (totalRows, validRows, errorRows), and row-level findings. No data is saved yet.
- POST /csv/session/{sessionId}/commit: imports only error-free rows from the analyzed session
- Returns CommitResult with inserted/updated counts.
- DELETE /csv/session/{sessionId}: cancels and discards session state (idempotent; always returns 204).
- GET /csv/session/{sessionId}/rows: page through analyzed rows
- Query params:
- skip (default 0), limit (default 50)
- onlyErrors (default false): when true, returns only rows with errors
- intent (optional): filter rows by intended action: INSERT, UPDATE, or SKIP

Notes and constraints: - requestedColumns must reference actual model fields. Unknown fields or multiple different nested properties are rejected (only one nested property across requestedColumns is allowed when using [0]). - Unrecognized query parameters are rejected with HTTP 400 to prevent silent misconfiguration. - Very large exports should prefer streaming with sensible length settings or server-side filters to reduce memory and time. - Imports run under the same security rules as POST / (save). Ensure the caller has permission to create/update the target entities in the chosen realm.