Table of Contents

Chapter 1. Query Language Reference

Quantum uses an ANTLR-based query language (BIAPIQuery.g4) for filtering, searching, and constraining data across all REST endpoints. This single, consistent syntax works everywhere: list APIs, permission rules, and access resolvers.

1.1. Basic Syntax

1.1.1. Operators

Operator	Symbol	Example
Equals	`:`	`name:"Widget"`
Not equals	`:!`	`status:!"DELETED"`
Less than	`:<`	`price:<##100`
Greater than	`:>`	`quantity:>#50`
Less than or equal	`:⇐`	`createdDate:⇐2024-12-31`
Greater than or equal	`:>=`	`updatedAt:>=2024-01-01T00:00:00Z`
Field exists	`:~`	`description:~`
In list	`:^`	`status:^["ACTIVE","PENDING"]`
Not in list	`:!^`	`status:!^["DELETED","ARCHIVED"]`

1.1.2. Value Types

Type	Prefix	Example
String	none or `"…"`	`name:widget` or `name:"Super Widget"`
Number (integer)	`#`	`quantity:#42`
Number (decimal)	`##`	`price:##19.99`
Date	none	`shipDate:2024-12-25`
DateTime	none	`createdAt:2024-12-25T10:30:00Z`
Boolean	none	`active:true`
Null	none	`description:null`
ObjectId	none	`id:507f1f77bcf86cd799439011`
Reference	`@@`	`parentId:@@507f1f77bcf86cd799439011`
Variable	`${…}`	`ownerId:${principalId}`

1.1.3. Logical Operators

Operator	Symbol	Example
AND	`&&`	`active:true && price:>##10`
OR	`||`	`status:"ACTIVE" || status:"PENDING"`
NOT	`!!`	`!!(price:<##5)`
Grouping	`()`	`(active:true || featured:true) && price:>##0`

1.2. Common Patterns

1.2.1. String Matching

```
# Exact match
name:"Super Widget"

# Wildcard matching
name:*widget*        # contains "widget"
name:widget*         # starts with "widget"
name:*widget         # ends with "widget"
name:w?dget          # single character wildcard

# Case sensitivity (depends on database collation)
name:"WIDGET"        # may or may not match "widget"
```

1.2.2. Text Search

```
# Full-text search (requires MongoDB text index)
text("priority escalation")

# Combine text search with other filters
text("priority escalation") && status:"OPEN"
```

Notes:

- `text(…)` maps to MongoDB `$text` queries and is case-insensitive by default.
- MongoDB allows only one `text(…)` clause per query.
- Text search must be backed by a text index on the target collection.

> MongoDB requires `$text` to be a top-level query operator. The following usages are **not allowed** and will be rejected at parse time:

Invalid Usage	Example	Reason
Inside NOT	`!!text("foo")`	MongoDB does not support `$text` inside `$not` or `$nor`
Inside OR	`text("foo") || status:"OPEN"`	MongoDB does not allow `$text` inside `$or`
Inside elemMatch	`tags:{text("foo")}`	MongoDB does not support `$text` inside `$elemMatch`
Multiple text clauses	`text("foo") && text("bar")`	Only one `$text` operator is allowed per query

Valid usage: `text(…)` combined with `&&` (AND) at the top level is supported because MongoDB allows `$text` alongside other top-level conditions.

1.2.3. Numeric Ranges

```
# Price between 10 and 100
price:>=##10 && price:<=##100

# Quantity greater than 0
quantity:>#0

# Exact count
itemCount:#5
```

1.2.4. Date and Time Queries

```
# Orders from today
createdDate:>=2024-12-25

# Orders from last week
createdDate:>=2024-12-18 && createdDate:<2024-12-25

# Specific timestamp
updatedAt:2024-12-25T14:30:00Z

# Orders modified this year
updatedAt:>=2024-01-01T00:00:00Z
```

1.2.5. List Membership

```
# Status in specific values (IN)
status:^["ACTIVE","PENDING","PROCESSING"]

# Exclude statuses (NOT IN)
status:!^["DELETED","ARCHIVED"]
```

```
# User IDs from a list (IN)
ownerId:^["user1","user2","user3"]

# Exclude specific users (NOT IN)
ownerId:!^["user1","user2"]

# ObjectId list (IN)
categoryId:^[@@507f1f77bcf86cd799439011, @@507f1f77bcf86cd799439012]

# ObjectId list (NOT IN)
categoryId:!^[@@507f1f77bcf86cd799439011, @@507f1f77bcf86cd799439012]

# Mixed types (coerced automatically)
priority:^[#1,#2,#3]

# Using variables (CSV expansion supported by access resolvers)
customerId:!^[${accessibleCustomerIds}]
```

Null and Existence Checks

```
# Field has any value (not null)
description:~

# Field is null
description:null

# Field is not null
description:!null

# Field exists and is not empty string
description:~ && description:!""
```

1.3. Querying Dynamic Attributes

Models that implement `DynamicAttributeSupport` can store flexible, user-defined attributes organized into named sets. These can be queried using the elemMatch syntax for array fields.

1.3.1. Dynamic Attribute Structure

Dynamic attributes are stored as:

```
{
  "dynamicAttributeSets": [
    {
      "name": "logistics",
      "attributes": [
        { "id": "uuid-1", "name": "weight", "value": 25.5, "type": "NUMBER" },
        { "id": "uuid-2", "name": "hazmat", "value": false, "type": "BOOLEAN" }
```

```
      ]
    },
    {
      "name": "compliance",
      "attributes": [
        { "id": "uuid-3", "name": "certNumber", "value": "CERT-123", "type": "STRING"
}
      ]
    }
  ]
}
```

1.3.2. Querying by Attribute ID and Value

Use elemMatch syntax `{…}` to query attributes within the array:

```
# Find records where a specific attribute (by ID) has a specific value
dynamicAttributeSets.attributes:{id:21f63b90-08b4-4280-a28d-f003f9c114b3 &&
value:"Conference hall"}

# Find records with a numeric attribute value
dynamicAttributeSets.attributes:{id:94138c39-2115-4681-8c18-aa0c0596b065 && value:#99}

# Find records with a decimal attribute value
dynamicAttributeSets.attributes:{id:uuid-1 && value:##25.5}
```

1.3.3. Querying by Attribute Name and Value

```
# Find records where an attribute named "weight" has value > 10
dynamicAttributeSets.attributes:{name:"weight" && value:>##10}

# Find records with a specific attribute name and string value
dynamicAttributeSets.attributes:{name:"certNumber" && value:"CERT-123"}

# Boolean attribute query
dynamicAttributeSets.attributes:{name:"hazmat" && value:true}
```

1.3.4. Combining Multiple Dynamic Attribute Conditions

```
# Multiple attributes must match (separate elemMatch conditions)
dynamicAttributeSets.attributes:{name:"weight" && value:>##10} &&
dynamicAttributeSets.attributes:{name:"hazmat" && value:false}

# OR within a single attribute query
dynamicAttributeSets.attributes:{value:"A" || value:"B"}
```

```
# Combined with regular field queries
status:"ACTIVE" && dynamicAttributeSets.attributes:{name:"priority" && value:"HIGH"}
```

1.3.5. Querying by Set Name

To filter by the set name itself:

```
# Records that have a "logistics" attribute set
dynamicAttributeSets:{name:"logistics"}

# Combine set name with attribute value
dynamicAttributeSets:{name:"logistics"} &&
dynamicAttributeSets.attributes:{name:"weight" && value:>##50}
```

1.3.6. Advanced Tags (Simple Key-Value)

For simpler tag-based filtering, use `advancedTags`:

```
# Find by tag name and value
advancedTags:{name:"priority" && value:"urgent"}

# Multiple tag conditions
advancedTags:{name:"region" && value:"US"} && advancedTags:{name:"tier" &&
value:"premium"}
```

1.3.7. Advanced Examples

Complex Business Logic

```
# Active products under $50 OR featured products at any price
(active:true && price:<##50) || featured:true

# Orders needing attention: overdue OR high-value pending
(dueDate:<2024-12-25 && status:!"COMPLETED") ||
(status:"PENDING" && totalAmount:>##1000)

# Products with inventory issues
(quantity:<=#5 && reorderPoint:>#5) || stockStatus:"OUT_OF_STOCK"
```

Multi-tenant Filtering

```
# User's own records
dataDomain.ownerId:${principalId}

# Organization-wide access
```

```
dataDomain.orgRefName:${orgRefName}

# Tenant-scoped with public sharing
dataDomain.tenantId:${pTenantId} || dataDomain.orgRefName:"PUBLIC"
```

Audit and Compliance

```
# Records modified by specific user
auditInfo.lastUpdatedBy:"john.doe"

# Changes in date range
auditInfo.lastUpdatedDate:>=2024-12-01 &&
auditInfo.lastUpdatedDate:<2024-12-31

# Created vs modified
auditInfo.createdDate:auditInfo.lastUpdatedDate  # never modified
auditInfo.createdDate:!auditInfo.lastUpdatedDate # has been modified
```

1.3.8. Variables in Filters

Variables are resolved from the current security context and can be used in permission rules and access resolvers.

Standard Variables

Variable	Description
`${principalId}`	Current user’s ID
`${pTenantId}`	Principal’s tenant ID
`${pAccountId}`	Principal’s account ID
`${pOrgRefName}`	Principal’s organization
`${realm}`	Current realm/database
`${area}`	Current functional area
`${functionalDomain}`	Current functional domain
`${action}`	Current action (CREATE, UPDATE, etc.)

Custom Variables from Access Resolvers

```java
// In your AccessListResolver
@Override
public String key() {
    return "accessibleCustomerIds";  // becomes ${accessibleCustomerIds}
}

@Override
```

```java
public Collection<?> resolve(...) {
    return Arrays.asList("CUST001", "CUST002", "CUST003");
}
```

```
# Use in filter
customerId:^[${accessibleCustomerIds}]
```

1.3.9. Performance Tips

Efficient Queries

```
# Good: Use indexed fields first
status:"ACTIVE" && createdDate:>=2024-01-01

# Better: Combine with specific values
status:"ACTIVE" && ownerId:${principalId} && createdDate:>=2024-01-01

# Avoid: Leading wildcards on large collections
name:*widget  # can be slow on millions of records
```

Projection for Large Objects

```
# In REST calls, limit returned fields
GET /products/list?filter=active:true&projection=+id,+name,+price,-description
```

1.3.10. Integration with REST APIs

List Endpoints

```
# Basic filtering
GET /products/list?filter=active:true

# With sorting and pagination
GET /products/list?filter=price:>##10&sort=-createdDate&skip=20&limit=10

# Complex filter with projection
GET
/orders/list?filter=(status:"PENDING"||status:"PROCESSING")&&totalAmount:>##100&projec
tion=+id,+status,+totalAmount,+customerName
```

Permission Rules

```yaml
- name: user-own-records
  priority: 300
```

```
  match:
    method: [GET]
    url: /api/**
  effect: ALLOW
  andFilterString: "dataDomain.ownerId:${principalId}"
```

Access Resolvers

```
// Resolver returns customer IDs user can access
public Collection<?> resolve(...) {
    return customerService.getAccessibleIds(principalId);
}

// Used in permission rule
andFilterString: "customerId:^[${accessibleCustomerIds}]"
```

1.3.11. Error Handling

Common syntax errors and solutions:

```
# Wrong: Missing quotes for multi-word strings
name:Super Widget
# Right:
name:"Super Widget"

# Wrong: Incorrect number prefix
price:19.99
# Right:
price:##19.99

# Wrong: Invalid date format
createdDate:12/25/2024
# Right:
createdDate:2024-12-25

# Wrong: Unbalanced parentheses
(active:true && price:>##10
# Right:
(active:true && price:>##10)

# Wrong: text() inside NOT
!!text("search")
# Right: Use text() at top level only
text("search") && status:!"DELETED"

# Wrong: text() inside OR
text("search") || status:"ACTIVE"
# Right: Use text() with AND only
```

```
text("search") && (status:"ACTIVE" || status:"PENDING")

# Wrong: Multiple text() clauses
text("foo") && text("bar")
# Right: Combine search terms in a single text()
text("foo bar")
```

1.3.12. See Also

- REST CRUD Querying
- Permission Rules
- Access Resolvers

1.3.13. Execution engines and listeners

The BIAPI query syntax is parsed once (via ANTLR) and can be executed by different "listeners" depending on the use case. Quantum ships with two primary implementations that share the same grammar and semantics:

- Morphia listener: converts a query into Mongo/Morphia Filters for database-side execution
- In-memory listener: converts a query into a Java Predicate over JSON data for Quarkus/GraalVM-friendly in-memory execution

Morphia: QueryToFilterListener

Use this when you want the database to perform the filtering. The listener walks the parse tree and produces a dev.morphia.query.filters.Filter which you can apply to Morphia queries. This is ideal for repository APIs and any endpoint where you want to leverage MongoDB indexes and avoid loading large data sets into memory.

Key characteristics: - Output type: Morphia Filter - Execution: database-side (MongoDB) - Semantics: identical to grammar (comparisons, IN/NIN, exists, null, regex with wildcards, elemMatch, boolean &&/||/!!) - Text search: `text("…")` is supported when a collection has a text index; only one text clause is allowed per query, and it must be at the top level (cannot be inside OR, NOT, or elemMatch). - Variable expansion: supports ${vars} and single-variable IN list expansion (e.g., [${ids}] can expand to a collection/array or a comma-separated string)

Example:

```java
import com.e2eq.framework.grammar.*;
import com.e2eq.framework.model.persistent.morphia.QueryToFilterListener;
import dev.morphia.query.filters.Filter;
import dev.morphia.query.filters.Filters;
import org.antlr.v4.runtime.*;
import org.antlr.v4.runtime.tree.ParseTreeWalker;
import org.apache.commons.text.StringSubstitutor;

String query = "(status:Assigned||status:Pending)&&displayName:*Route*";
```

```java
var vars = java.util.Map.<String,String>of();

// Parse
CharStream cs = CharStreams.fromString(query);
BIAPIQueryLexer lexer = new BIAPIQueryLexer(cs);
CommonTokenStream tokens = new CommonTokenStream(lexer);
BIAPIQueryParser parser = new BIAPIQueryParser(tokens);
BIAPIQueryParser.QueryContext tree = parser.query();

// Build Morphia filter
QueryToFilterListener listener = new QueryToFilterListener(vars, new
StringSubstitutor(vars), /* modelClass */ null);
ParseTreeWalker.DEFAULT.walk(listener, tree);
Filter morphiaFilter = listener.getFilter();

// Use with Morphia query (example)
// datastore.find(MyEntity.class).filter(morphiaFilter).iterator().toList();
```

A few query examples (taken from testQueryStrings.txt):

- Equality: field:"quotedString"
- Comparisons: field:>##12.56, field:⇐#123
- IN/NIN: field:^[value1,value2], field:!^[value1,value2]
- Exists/Null: field:~, field:null
- elemMatch: arrayField:{(subField:<#12)||(subField:>#15)}

In-memory (JsonNode): QueryToPredicateJsonListener

Use this when you need to evaluate queries in memory without reflection on POJOs. This implementation compiles a query into a java.util.function.Predicate over a Jackson JsonNode. It is Quarkus/GraalVM friendly, useful for: - Unit tests where you want to validate query behavior without a database - Post-filtering or pre-filtering of already-fetched data - Evaluating access rules or business logic against transient objects

Key characteristics: - Output type: Predicate<JsonNode> - Execution: in-memory - No runtime reflection: operates on JsonNode - Semantics and variable expansion match the Morphia listener

Convenience helpers exist in QueryPredicates:

```java
import com.e2eq.framework.query.QueryPredicates;
import com.fasterxml.jackson.databind.JsonNode;
import java.util.function.Predicate;
import java.util.Map;

String query = "(status:Assigned||status:Pending)&&displayName:*Route*";
Predicate<JsonNode> p = QueryPredicates.compilePredicate(query, Map.of(), Map.of());

// Example data as a POJO or Map -> convert to JsonNode
```

```
record Ticket(String status, String displayName) {}
Ticket ticket = new Ticket("Assigned", "Route Exception in
Route:To[http://com.xxx/update]");
JsonNode node = QueryPredicates.toJsonNode(ticket);

boolean include = p.test(node); // true
```

Additional examples

- Equality and comparisons

```
var vars = Map.<String,String>of();
var objVars = Map.<String,Object>of();
Predicate<JsonNode> eq = QueryPredicates.compilePredicate("quantity:#42", vars,
objVars);
Predicate<JsonNode> gt = QueryPredicates.compilePredicate("price:>##19.99", vars,
objVars);

JsonNode product = QueryPredicates.toJsonNode(Map.of("quantity", 42, "price", 25.00));
assert eq.test(product);
assert gt.test(product);
```

- IN / NIN with variable expansion

```
var vars = Map.of("principalId", "66d1f1ab452b94674bbd934a");
Predicate<JsonNode> in =
QueryPredicates.compilePredicate("ownerId:^[${principalId},value2]", vars, Map.of());
JsonNode doc = QueryPredicates.toJsonNode(Map.of("ownerId",
"66d1f1ab452b94674bbd934a"));
assert in.test(doc);
```

- elemMatch over arrays of objects

```
String q = "items:{(sku:abc||qty:>#10)&&price:<=##9.99}";
Predicate<JsonNode> em = QueryPredicates.compilePredicate(q, Map.of(), Map.of());
JsonNode order = QueryPredicates.toJsonNode(Map.of(
  "items", java.util.List.of(
     Map.of("sku","abc","qty", 5,  "price", 9.99),
     Map.of("sku","xyz","qty", 12, "price", 8.50)
)));
// Matches: first item by sku OR second item by qty with price cap
assert em.test(order);
```

- Regex with wildcards

```
Predicate<JsonNode> rx = QueryPredicates.compilePredicate("displayName:*Route*",
```

```
Map.of(), Map.of());
JsonNode ticket = QueryPredicates.toJsonNode(Map.of("displayName", "Route Exception in
Route:To[...]"));
assert rx.test(ticket);
```

Choosing the right listener

- Use Morphia (QueryToFilterListener) when:
  - You are filtering MongoDB collections and want the DB to do the work (indexing, pagination, scalability)
  - You need server-side performance and minimal memory footprint
- Use In-memory (QueryToPredicateJsonListener) when:
  - You run in Quarkus native image and want to avoid reflection on POJOs
  - You are writing unit tests or applying rules to transient/aggregated data
  - You need to evaluate a query over already materialized objects without another database round-trip

Both listeners aim to maintain parity with the grammar. If you observe mismatches, please file an issue with the query string, the evaluated data sample, and the expected vs actual results.

Query Field Validation

Quantum provides validation capabilities to detect references to non-existent fields before query execution, preventing runtime errors and providing early feedback.

Validating Listeners

Two validating listener implementations extend the standard listeners:

- ValidatingQueryToFilterListener: extends QueryToFilterListener for Morphia queries
- ValidatingQueryToPredicateJsonListener: extends QueryToPredicateJsonListener for in-memory predicates

Both validate field references against a model class during parsing and accumulate errors.

Example usage:

```
import com.e2eq.framework.model.persistent.morphia.ValidatingQueryToFilterListener;
import com.e2eq.framework.grammar.*;
import org.antlr.v4.runtime.*;
import org.antlr.v4.runtime.tree.ParseTreeWalker;

String query = "name:John AND invalidField:test";

BIAPIQueryLexer lexer = new BIAPIQueryLexer(CharStreams.fromString(query));
CommonTokenStream tokens = new CommonTokenStream(lexer);
BIAPIQueryParser parser = new BIAPIQueryParser(tokens);
```

```java
ValidatingQueryToFilterListener listener =
    new ValidatingQueryToFilterListener(UserProfile.class);
ParseTreeWalker.DEFAULT.walk(listener, parser.query());

if (listener.hasValidationErrors()) {
    List<String> errors = listener.getValidationErrors();
    throw new IllegalArgumentException("Invalid query: " + errors);
}

Filter filter = listener.getFilter();
```

@ValidQueryFilter Annotation

For automatic validation in DTOs and models, use the `@ValidQueryFilter` annotation:

```java
import com.e2eq.framework.annotations.ValidQueryFilter;
import jakarta.validation.constraints.NotNull;

public class SearchRequest {
    @NotNull
    @ValidQueryFilter(modelClass = UserProfile.class)
    private String filterQuery;

    // getters/setters
}
```

The annotation integrates with Jakarta Bean Validation and is automatically enforced by the ValidationInterceptor during entity persistence.

REST endpoint example:

```java
@Path("/users")
public class UserResource {

    @GET
    public Response searchUsers(@Valid @BeanParam SearchParams params) {
        // filterQuery is validated before this method executes
        return Response.ok(userService.search(params.getFilterQuery())).build();
    }
}

public class SearchParams {
    @QueryParam("filter")
    @ValidQueryFilter(modelClass = UserProfile.class)
    private String filterQuery;
}
```

Persisted filter example:

```java
public class SavedSearch extends BaseModel {
    private String name;

    @ValidQueryFilter(modelClass = Order.class)
    private String filterExpression;

    // When saved, ValidationInterceptor validates the filter
}
```

QueryFieldValidator Utility

For programmatic validation without listeners:

```java
import com.e2eq.framework.query.QueryFieldValidator;

// Validate against a model class
QueryFieldValidator validator = QueryFieldValidator.forModelClass(UserProfile.class);

boolean isValid = validator.validateField("email");  // true
boolean isInvalid = validator.validateField("nonExistentField");  // false

if (validator.hasErrors()) {
    List<String> errors = validator.getErrors();
    // Handle validation errors
}

// Validate against aggregation pipeline schema
Map<String, Class<?>> schema = Map.of(
    "totalAmount", Double.class,
    "orderCount", Long.class,
    "customerName", String.class
);

QueryFieldValidator aggValidator = QueryFieldValidator.forAggregationSchema(schema);
aggValidator.validateField("totalAmount");  // true
aggValidator.validateField("invalidField");  // false
```

Benefits

- Early error detection: catch field reference errors before query execution
- Better error messages: specific feedback about which fields are invalid
- Development safety: prevent typos and refactoring issues
- API validation: validate user-provided query strings in REST endpoints
- Data integrity: prevent saving invalid filter expressions to the database

1.3.14. Expressing Ontology Constraints in Queries (optional)

The BIAPI query language primarily targets fields on documents. When using the ontology modules, there are two ways to incorporate ontology relationships into queries:

Option A: Variable from an AccessListResolver

- A resolver computes the set of IDs using EdgeDao (e.g., orders related by placedInOrg to the caller's org).
- Use an IN clause over id in your filter string:

```
# idsByPlacedInOrg is published by an AccessListResolver
id:^${idsByPlacedInOrg}
```

1.3.15. Ontology operators: hasEdge and hasIncomingEdge

When the ontology module is enabled and edges are materialized, you can constrain results by semantic relationships using `hasEdge` and `hasIncomingEdge`.

- Syntax: `hasEdge(predicate, dst)`
- Finds entities (sources) that have an edge with the specified `predicate` pointing TO the `dst`.
- Syntax: `hasIncomingEdge(predicate, src)`
- Finds entities (targets) that have an edge with the specified `predicate` coming FROM the `src`.

Arguments: - `predicate` can be an unquoted string, a quoted string, or a `${var}` placeholder. - `dst` / `src` can be a literal id, a variable like `${principalId}`, an ObjectId, or a reference literal `@@…`.

Examples:

```
# Orders placed in a specific organization (find sources)
hasEdge("placedInOrg", ${orgRefName})

# Tickets routed to a region by id
hasEdge(routedToRegion, @@507f1f77bcf86cd799439011)

# Find locations that an associate can access (find targets)
# assumes an edge exists: Associate --canAccessLocation--> Location
hasIncomingEdge(canAccessLocation, ${principalId})
```

Semantics:

- These operators narrow the result set based on the existence of edges in the ontology edges collection.
- They compose with other filters using `&&`, `||`, and `!!`.
- Tenancy: edge resolution is always scoped by tenant/realm.

1.3.16. Root projection recap: fields:[+…,-…]

To include or exclude fields from the root documents:

- Syntax: `fields:[+f1,+f2,-f3]`
- Include mode: if any `+` is present, only the listed fields are included (with explicit `-` carving out exceptions).
- Exclude mode: if only `-` entries exist, all fields are included except those.
- Default `_id`: preserved unless explicitly specified (`+_id`/`-_id`).

Examples

```
# Include a few fields from the root and drop one
fields:[+_id,+total,-internalNotes]

# Combine with expansion (planner chooses aggregation mode)
expand(customer) && fields:[+_id,+customer,+total]
```

For per-expansion projections and advanced traversal filters, see Query Expansion.