Table of Contents

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

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"]`

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}`

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

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"
```

Numeric Ranges

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

# Quantity greater than 0
quantity:>#0

# Exact count
itemCount:#5
```

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
```

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. 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
```

1.4. 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

Variable	Description
${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

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

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

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

1.5. 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
```

1.6. 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

```
- 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}]"
```

1.7. 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)
```

1.8. See Also

- REST CRUD Querying
- Permission Rules
- Access Resolvers

1.9. 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 &&/||/!!) - 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:

```
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*";
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:

```
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.