DomainContext, RuleContext, and DataDomain

Version 1.2.2-SNAPSHOT, 2025-10-22T01:39:23Z

Table of Contents

Quantum enforces multi-tenant isolation and sharing through contextual data carried on models and evaluated at runtime.

Chapter 1. DataDomain

Every persisted model includes a DataDomain that describes ownership and scope, commonly including fields such as:

- tenantId: Identifies the tenant
- orgRefName: Organization unit reference within a tenant
- ownerId: Owning user or system entity
- realm: Optional runtime override for partitioning

These fields enable filtering, authorization, and controlled sharing of data between tenants or org units.

Chapter 2. DomainContext

DomainContext represents the current execution context for a request or operation, typically capturing:

- current tenant/org/user identity
- functional area / functional domain
- the action being executed (e.g., CREATE, UPDATE, VIEW, DELETE, ARCHIVE)

It feeds downstream components (repositories, resources) to consistently apply filtering and policy decisions.

Chapter 3. RuleContext

RuleContext encapsulates policy evaluation. It can:

- Enforce whether an action is allowed for a given model and DataDomain
- Produce additional filters and projections used by repositories
- Grant cross-tenant read access for specific functional areas (e.g., shared catalogs) while keeping others strictly isolated

Chapter 4. End-to-End Flow

1. A REST request enters a BaseResource-derived endpoint.
2. The resource builds a DomainContext from the security principal and request parameters.
3. RuleContext evaluates permissions and returns effective filters.
4. Repository applies filters (DataDomain-aware) to find/get/list/update/delete.
5. The model's UIActionList can be computed to reflect what the caller can do next.

This pattern ensures consistent enforcement across all CRUD operations, independent of the specific model or repository.

Chapter 5. Resolvers and Variables in Rule Filters

RuleContext can attach FILTERs (not only ALLOW/DENY) to repository queries using rule fields and filter strings. Variables inside those filter strings are populated from:

- PrincipalContext and ResourceContext standard variables: principalId, pAccountId, pTenantId, ownerId, orgRefName, resourceId, action, functionalDomain, area
- AccessListResolver SPI implementations: per-request computed Collections (e.g., customer IDs the caller can access)

Implementation highlights: - AccessListResolver has methods key(), supports(...), resolve(...). Resolvers are injected and invoked for each request; results are published as variables by key. - MorphiaUtils.VariableBundle carries both string variables and object variables (including collections) to the query listener. - The QueryToFilterListener supports IN clauses using a single ${var} inside brackets, expanding Collections/arrays and coercing types (ObjectId, numbers, booleans, dates).

Authoring examples: - Constrain by principal domain:

+

```
orgRefName:${orgRefName} && dataDomain.tenantId:${pTenantId}
```

- Access list resolver for customer visibility:

  ```
  customerId:^[${accessibleCustomerIds}]
  ```

For the complete query language reference, see Query Language.

Chapter 6. Concrete example: building and using a resolver

This section shows how to implement a resolver that restricts access to orders by the set of customerIds the current user is allowed to see.

6.1. 1) Implement the SPI

Create a CDI bean that implements AccessListResolver. It decides when it applies and returns a Collection of values. The collection can be ObjectId, String, numbers, etc.

```
import com.e2eq.framework.securityrules.AccessListResolver;
import com.e2eq.framework.model.persistent.base.UnversionedBaseModel;
import com.e2eq.framework.model.securityrules.PrincipalContext;
import com.e2eq.framework.model.securityrules.ResourceContext;
import jakarta.enterprise.context.ApplicationScoped;
import jakarta.inject.Inject;
import org.bson.types.ObjectId;
import java.util.*;

@ApplicationScoped
public class CustomerAccessResolver implements AccessListResolver {

    @Inject CustomerAccessService service; // your app-specific service

    @Override
    public String key() {
        // This becomes the variable name available to rules: ${accessibleCustomerIds}
        return "accessibleCustomerIds";
    }

    @Override
    public boolean supports(PrincipalContext pctx, ResourceContext rctx,
                            Class<? extends UnversionedBaseModel> modelClass) {
        // Optionally narrow by area/domain/action/model
        return rctx != null && "sales".equalsIgnoreCase(rctx.getArea())
               && "order".equalsIgnoreCase(rctx.getFunctionalDomain());
    }

    @Override
    public Collection<?> resolve(PrincipalContext pctx, ResourceContext rctx,
                                 Class<? extends UnversionedBaseModel> modelClass) {
        // Return the set of customer ids for this user; could be ObjectId or String.
        // Example returns strings; the query listener will coerce 24-hex to ObjectId.
        return service.findCustomerIdsForUser(pctx.getUserId());
    }
}
```

Notes: - You can return List<ObjectId> directly if you prefer; no coercion needed then. - The resolver runs per request. Cache internally if the computation is expensive.

6.2. 2) How RuleContext uses resolvers

At query time, RuleContext discovers all AccessListResolver beans and calls supports(…). For those that apply, it invokes resolve(…) and publishes the result into the variable bundle under the provided key(). Variables are available to the BIAPI query via ${…}.

Internally this uses MorphiaUtils.VariableBundle and QueryToFilterListener to carry both strings and typed objects/collections.

6.3. 3) Author a rule that consumes the variable

Given the resolver above, a rule can attach an IN filter to constrain queries:

```
// andFilterString (example)
customerId:^[${accessibleCustomerIds}]
```

When executed: - If accessibleCustomerIds is a Collection/array, each element is type-coerced (ObjectId, number, date, boolean, or string) and used in $in. - If accessibleCustomerIds is a comma-separated string, it is split and each token is coerced similarly. - An empty collection results in an empty $in (matches none), effectively denying access via filtering, not via ALLOW/DENY.

6.4. 4) End-to-end behavior

- SecurityFilter sets ResourceContext (area/domain/action) per request.
- RuleContext evaluates rules for the principal and resource and gathers resolvers.
- The repository composes filters including the rule-provided IN clause with the access list.
- Only documents whose customerId is in the caller's resolved set are returned.

6.5. String literals vs. typed values in resolver variables

When an `AccessListResolver` returns a list of values that will be used in an `IN` clause (for example, `field:^[${var}]`), the engine attempts to coerce each element to an appropriate type so Mongo/Morphia filters are typed correctly:

- 24-hex string → ObjectId
- `true`/`false` → Boolean
- integer → Long
- decimal → Double
- ISO-8601 datetime → `java.util.Date`

- `yyyy-MM-dd` → `java.time.LocalDate`
- otherwise → String

This works well when your target field is an ObjectId, number, or date. However, string fields can contain values that look like other types (for example, a 24-hex string that resembles an ObjectId). In those cases you must force "treat as plain string" so no coercion occurs.

To do this, the framework provides a small wrapper type `StringLiteral`. If a resolver returns `StringLiteral` instances, the listener unwraps them to plain `String` values and skips coercion entirely.

6.5.1. Example A: Resolver returns ObjectIds (typed)

```
@ApplicationScoped
public class CustomerAccessResolver implements AccessListResolver {
    public static final ObjectId ID1 = new ObjectId("5f1e1a5e5e5e5e5e5e5e5e51");
    public static final ObjectId ID2 = new ObjectId("5f1e1a5e5e5e5e5e5e5e5e52");

    @Override public String key() { return "accessibleCustomerIds"; }
    @Override public boolean supports(PrincipalContext p, ResourceContext r, Class<?
extends UnversionedBaseModel> m) {
        return r != null && "sales".equalsIgnoreCase(r.getArea()) && "order"
.equalsIgnoreCase(r.getFunctionalDomain()) && "view".equalsIgnoreCase(r.getAction());
    }
    @Override public Collection<?> resolve(PrincipalContext p, ResourceContext r,
Class<? extends UnversionedBaseModel> m) {
        return java.util.List.of(ID1, ID2); // typed values pass through as-is
    }
}
```

Rule:

```
customerId:^[${accessibleCustomerIds}]
```

Result: `$in` with `List<ObjectId>` on `customerId`.

6.5.2. Example B: Resolver returns String literals (force raw strings)

```
@ApplicationScoped
public class CustomerCodeResolver implements AccessListResolver {
    @Override public String key() { return "accessibleCustomerCodes"; }
    @Override public boolean supports(PrincipalContext p, ResourceContext r, Class<?
extends UnversionedBaseModel> m) {
        return r != null && "sales".equalsIgnoreCase(r.getArea()) && "order"
.equalsIgnoreCase(r.getFunctionalDomain()) && "view".equalsIgnoreCase(r.getAction());
    }
    @Override public Collection<?> resolve(PrincipalContext p, ResourceContext r,
```

```
Class<? extends UnversionedBaseModel> m) {
        return java.util.List.of(
            com.e2eq.framework.model.persistent.morphia.StringLiteral.of(
"5f1e1a5e5e5e5e5e5e5e51"),
            com.e2eq.framework.model.persistent.morphia.StringLiteral.of("CUST-42")
        );
    }
}
```

Rule:

```
customerCode:^[${accessibleCustomerCodes}]
```

Result: `$in` with `List<String>` on `customerCode` (even for hex-like strings).

6.5.3. Other types supported

Resolvers can also return numbers, booleans, and dates/datetimes. Already-typed elements (Number, Boolean, java.util.Date, java.time.LocalDate, ObjectId) are preserved. String elements are heuristically parsed into those types unless wrapped with `StringLiteral`.

Authoring tips:

- Prefer returning already-typed values when you know the target field type.
- Use `StringLiteral` when a value might be misinterpreted (for example, 24-hex or numeric-looking strings).
- For CSV strings published under a variable, the engine splits by comma and applies the same per-element coercion.

6.6. Using AccessListResolver with Ontology (optional)

When ontology is enabled, an AccessListResolver can compute ID lists from semantic edges (materialized in Mongo) and publish them as variables for use in rule filters.

Example resolver (conceptual)

```
@ApplicationScoped
public class OrdersByOrgResolver implements AccessListResolver {
  @Inject EdgeDao edgeDao; // from quantum-ontology-mongo
  @Override public String key() { return "idsByPlacedInOrg"; }
  @Override public boolean supports(PrincipalContext p, ResourceContext r, Class<?
extends UnversionedBaseModel> model) {
    return model.getSimpleName().equals("Order");
  }
  @Override public Collection<?> resolve(PrincipalContext p, ResourceContext r, Class
<? extends UnversionedBaseModel> model) {
    String tenantId = p.getDataDomain().getTenantId();
```

```
    String org = p.getDataDomain().getOrgRefName();
    return edgeDao.srcIdsByDst(tenantId, "placedInOrg", org);
  }
}
```

Rule filter usage

```
id:^${idsByPlacedInOrg}
```

Notes

- Always scope by tenantId from RuleContext/PrincipalContext.
- This is optional and only active if you wire ontology components. For a deeper integration path, see Integrating Ontology.