

Table of Contents

1. Security Annotations: FunctionalMapping and FunctionalAction	1
1.1. @FunctionalMapping	1
1.2. @FunctionalAction	1
1.3. Default Action Mapping	2
1.4. How the Framework Uses These Annotations	2
1.5. Migration from Legacy Methods	2
1.6. Best Practices	3
1.7. Bypassing Data Scoping for System Operations	4
1.8. Integration with Permission Rules	5
1.9. See Also	6

1. Security Annotations: FunctionalMapping and FunctionalAction

Quantum uses annotations to declare a model or resource's functional area, domain, and actions for security evaluation. This replaces the legacy `bmFunctionalArea()` and `bmFunctionalDomain()` methods.

1.1. @FunctionalMapping

Use `@FunctionalMapping` on model classes or resource classes to declare their business placement:

```
import com.e2eq.framework.annotations.FunctionalMapping;

@Entity
@FunctionalMapping(area = "catalog", domain = "product")
public class Product extends BaseModel {
    // No need to override bmFunctionalArea/bmFunctionalDomain
}
```

```
@Path("/products")
@FunctionalMapping(area = "catalog", domain = "product")
public class ProductResource extends BaseResource<Product, ProductRepo> {
    // All methods inherit area/domain from class annotation
}
```

1.2. @FunctionalAction

Use `@FunctionalAction` on JAX-RS resource methods when the action differs from the HTTP verb default:

```
@Path("/products")
public class ProductResource {

    @POST
    @FunctionalAction("CREATE") // Explicit, though POST implies CREATE
    public Product create(Product payload) {
        return productRepo.save(payload);
    }

    @GET
    @Path("/{id}")
    // No annotation needed - GET implies VIEW
    public Product get(@PathParam("id") String id) {
        return productRepo.findById(id);
    }
}
```

```

@PUT
@Path("/{id}/approve")
@FunctionalAction("APPROVE") // Custom action beyond standard CRUD
public Product approve(@PathParam("id") String id) {
    Product p = productRepo.findById(id);
    p.setStatus("APPROVED");
    return productRepo.save(p);
}

```

1.3. Default Action Mapping

When `@FunctionalAction` is not present, actions are inferred from HTTP methods:

HTTP Method	Default Action
GET	VIEW
POST	CREATE
PUT	UPDATE
PATCH	UPDATE
DELETE	DELETE

1.4. How the Framework Uses These Annotations

SecurityFilter

- Reads `@FunctionalMapping` from the matched resource class for area/domain
- Reads `@FunctionalAction` from the method, or infers from HTTP method
- Falls back to path-based parsing if annotations are missing

MorphiaRepo.fillUIActions

- Uses `@FunctionalMapping` on model classes to resolve allowed UI actions
- Falls back to legacy `bmFunctionalArea()/bmFunctionalDomain()` methods

PermissionResource

- Prefers `@FunctionalMapping` when listing functional domains
- Falls back to legacy methods when annotation is missing

1.5. Migration from Legacy Methods

Current (Legacy) Approach

```

@Entity
public class Product extends BaseModel {

```

```

@Override
public String bmFunctionalArea() {
    return "Catalog";
}

@Override
public String bmFunctionalDomain() {
    return "Product";
}

```

New (Recommended) Approach

```

@Entity
@FunctionalMapping(area = "catalog", domain = "product")
public class Product extends BaseModel {
    // Clean - no method overrides needed
}

```

Transitional Support

You can use both during migration: - If `@FunctionalMapping` is present, it takes precedence - If missing, legacy methods are used as fallback - Plan to remove legacy methods in future releases

1.6. Best Practices

Consistent Naming

Use lowercase, kebab-case for areas and domains:

```

@FunctionalMapping(area = "supply-chain", domain = "purchase-order")
@FunctionalMapping(area = "catalog", domain = "product")
@FunctionalMapping(area = "identity", domain = "user-profile")

```

Resource vs Model Annotations

- **Prefer model annotations** for consistency across all usage
- Use resource annotations only when the resource handles multiple model types
- Avoid duplicating annotations on both model and resource for the same entity

Custom Actions

Define custom actions for business operations beyond CRUD:

```

@PUT
@Path("/{id}/publish")

```

```

@FunctionalAction("PUBLISH")
public Product publish(@PathParam("id") String id) { ... }

@POST
@Path("/{id}/duplicate")
@FunctionalAction("DUPLICATE")
public Product duplicate(@PathParam("id") String id) { ... }

@DELETE
@Path("/{id}/archive")
@FunctionalAction("ARCHIVE") // Soft delete vs hard DELETE
public void archive(@PathParam("id") String id) { ... }

```

1.7. Bypassing Data Scoping for System Operations

Some endpoints perform system-level operations that don't operate on tenant-scoped data entities. For example:

- Database migration operations (index creation, schema changes)
- System initialization and setup endpoints
- Administrative operations that don't query/modify tenant-scoped records

For these endpoints, permission rules may return a **SCOPED** decision with data-level constraints like `dataDomain.tenantId:${pTenantId}`. However, since there's no data entity to scope on, these constraints can't be meaningfully applied.

Use `bypassDataScoping = true` to mark such endpoints:

```

@POST
@Path("/indexes/applyAllIndexes/{realm}")
@RolesAllowed("admin")
@FunctionalMapping(area = "migration", domain = "indexes")
@FunctionalAction(value = "APPLY_ALL_INDEXES", bypassDataScoping = true)
public void applyAllIndexes(@PathParam("realm") String realm) {
    migrationService.applyAllIndexes(realm);
}

```

How It Works

When `bypassDataScoping = true`:

- Permission checks still apply** - The user must still be authorized (ALLOW decision) based on their roles and the permission rules
- SCOPED constraints are ignored** - Data-level filters (like `dataDomain.tenantId:${pTenantId}`) are bypassed because they're not applicable
- Audit logging** - The framework logs when data scoping is bypassed for transparency

Security Considerations



Only use `bypassDataScoping = true` for endpoints that genuinely do not operate on tenant-scoped data. Misuse could expose cross-tenant data access vulnerabilities.

Use Case	<code>bypassDataScoping</code>	Why
Database migrations	<code>true</code>	Index/schema operations don't involve tenant data
System initialization	<code>true</code>	Setup operations are realm-wide, not tenant-scoped
CRUD on tenant data	<code>false</code> (default)	Must respect tenant isolation
Report generation	<code>false</code>	Reports should only include user's accessible data

Example: MigrationResource

The `MigrationResource` class uses `bypassDataScoping` for all index management operations:

```
@Path("/system/migration")
@RolesAllowed({"admin"})
public class MigrationResource {

    @POST
    @Path("/indexes/applyIndexes/{realm}")
    @FunctionalMapping(area = "MIGRATION", domain = "INDEXES")
    @FunctionalAction(value = "APPLY_INDEXES", bypassDataScoping = true)
    public void applyIndexes(@PathParam("realm") String realm) {
        migrationService.applyIndexes(realm);
    }

    @POST
    @Path("/indexes/dropAllIndexes/{realm}")
    @FunctionalMapping(area = "MIGRATION", domain = "INDEXES")
    @FunctionalAction(value = "DROP_ALL_INDEXES", bypassDataScoping = true)
    public void dropIndexes(@PathParam("realm") String realm) {
        migrationService.dropAllIndexes(realm);
    }
}
```

1.8. Integration with Permission Rules

Annotations feed into permission rule matching:

```
- name: allow-catalog-reads
  priority: 300
  match:
```

```
method: [GET]
# Matches area/domain from @FunctionalMapping
functionalArea: catalog
functionalDomain: product
rolesAny: [USER, ADMIN]
effect: ALLOW
```

```
- name: admin-only-approval
priority: 100
match:
  method: [PUT]
  functionalArea: catalog
  functionalDomain: product
  # Matches @FunctionalAction("APPROVE")
  action: APPROVE
  rolesAll: [ADMIN]
  effect: ALLOW
```

1.9. See Also

- [Modeling: Functional Mapping](#)
- [Permission Rules](#)
- [DomainContext and RuleContext](#)