Table of Contents

Chapter 1. MCP Server and Client

The `quantum-mcp-server` module exposes the Query Gateway as a set of Model Context Protocol (MCP) tools and resources, and provides an MCP client for calling external MCP tool providers. This enables AI assistants (Claude Desktop, Cursor, ChatGPT, and others) to discover and invoke your application's CRUDL operations, browse entity schemas, and receive query-building hints — all through the standard MCP JSON-RPC protocol.

The module also includes a REST-based agent layer (`/api/agent/*`) that mirrors the same tool set for non-MCP integrations.

1.1. Module Overview

`quantum-mcp-server` is a standalone Maven module with three key dependencies:

Dependency	Version	Purpose
`quantum-framework`	`${quantum.version}`	Core framework (QueryGatewayResource, security, Morphia)
`quarkus-mcp-server-http`	1.9.1	Quarkiverse MCP Server — Streamable HTTP + legacy SSE transport
`quarkus-langchain4j-mcp`	1.6.0	Quarkiverse MCP Client via LangChain4j — connect to external MCP servers

Add the module to your application's POM:

```
<dependency>
    <groupId>com.end2endlogic</groupId>
    <artifactId>quantum-mcp-server</artifactId>
    <version>${quantum.version}</version>
</dependency>
```

1.2. MCP Server

The MCP server exposes the Query Gateway as six tools and three resources at the `/mcp` endpoint. MCP clients discover these automatically via the `tools/list` and `resources/list` JSON-RPC methods.

1.2.1. Tools

Tools are defined in `McpGatewayTools` using the Quarkiverse `@Tool` and `@ToolArg` annotations. Each tool delegates to `AgentExecuteHandler`, which routes to the `QueryGatewayResource` — reusing the same security, realm resolution, and query execution as the REST API.

Tool Name	Description
`query_rootTypes`	List available entity types (root types) that can be queried, saved, or deleted
`query_plan`	Return the query execution plan (FILTER vs AGGREGATION) for a rootType and BIAPI query string
`query_find`	Execute a BIAPI query and return matching entities (supports pagination, realm override)
`query_save`	Save (insert or update) an entity by rootType
`query_delete`	Delete a single entity by its ObjectId
`query_deleteMany`	Delete multiple entities matching a BIAPI query

Tool parameters

`query_find` accepts these arguments:

Parameter	Type	Description
`rootType`	string	Entity type simple name or FQCN (use `query_rootTypes` to discover)
`query`	string	BIAPI query string (e.g. `status:ACTIVE && region:West`)
`realm`	string	Optional tenant realm (defaults to caller's realm)
`limit`	integer	Optional max results (default 50)
`skip`	integer	Optional offset for pagination (default 0)

`query_save` accepts `rootType`, `entity` (JSON object matching the schema), and optional `realm`. `query_delete` accepts `rootType`, `id` (ObjectId hex), and optional `realm`. `query_deleteMany` accepts `rootType`, `query`, and optional `realm`.

1.2.2. Resources

Resources are defined in `McpSchemaResources` and `McpQueryHintsResource` using the `@Resource` annotation. MCP clients can read these to populate LLM context with schema information and query-building guidance.

Resource URI	Description
`quantum://schema`	Lists all available root types with class name, simple name, and collection name
`quantum://query-hints`	BIAPI query grammar summary, example queries by intent, and tips (expand, wildcards, ontology)
`quantum://permission-hints`	Permission check/evaluate API summary, area/domain/action mapping, and example check requests

1.2.3. Connecting an MCP Client

Any MCP-compatible client can connect to the `/mcp` endpoint. Example configuration for Claude Desktop (`claude_desktop_config.json`):

```
{
  "mcpServers": {
    "quantum": {
      "url": "http://localhost:8080/mcp"
    }
  }
}
```

For Cursor, add the server URL in Settings > MCP Servers.

Once connected, the client can:

1. Call `tools/list` to discover the six gateway tools
2. Call `resources/list` to discover schema and hint resources
3. Call `resources/read` with `quantum://query-hints` to learn the BIAPI query syntax
4. Call `tools/call` with `query_rootTypes` to see what entity types are available
5. Call `tools/call` with `query_find` to query data

1.3. MCP Client

The MCP client side uses the Quarkiverse LangChain4j MCP extension (`quarkus-langchain4j-mcp`) to connect to external MCP servers and consume their tools. This is used to integrate with external tool providers such as Helix MCP, Brain, or HelixAI.

1.3.1. Configuration

External MCP connections are configured in `application.properties` using the `quarkus.langchain4j.mcp.<client-name>` prefix:

```
# Example: connect to an external MCP server over Streamable HTTP
quarkus.langchain4j.mcp.helix.transport-type=streamable-http
quarkus.langchain4j.mcp.helix.url=http://helix-mcp.example.com/mcp

# Example: connect to a local MCP server via stdio
quarkus.langchain4j.mcp.brain.transport-type=stdio
quarkus.langchain4j.mcp.brain.command=npx,-y,@brain/mcp-server
```

Supported transport types: `stdio`, `http`, `streamable-http`, `websocket`.

1.3.2. Injecting MCP Tools

Inject tools from an external MCP server using `@McpToolBox`:

```
import io.quarkiverse.langchain4j.mcp.runtime.McpToolBox;
import dev.langchain4j.service.SystemMessage;
import io.quarkiverse.langchain4j.RegisterAiService;

@RegisterAiService
public interface MyAiService {

    @SystemMessage("You are a helpful assistant.")
    @McpToolBox("helix")
    String chat(String userMessage);
}
```

Or inject the client directly for programmatic use:

```
import io.quarkiverse.langchain4j.mcp.runtime.McpClientName;
import dev.langchain4j.mcp.client.McpClient;

@Inject
@McpClientName("helix")
McpClient helixClient;
```

1.4. Agent REST API

The agent layer provides a REST interface at `/api/agent` that mirrors the MCP tools for non-MCP integrations. This is useful for custom agent orchestrators, webhook-based workflows, or testing.

1.4.1. Endpoints

Method	Path	Description
GET	`/api/agent/tools`	List available gateway tools (optionally filtered by realm)
GET	`/api/agent/schema`	List all root types (same as `query_rootTypes`)
GET	`/api/agent/schema/{rootType}`	JSON Schema-like structure for a single entity type
GET	`/api/agent/query-hints`	Query grammar summary and example queries
GET	`/api/agent/permission-hints`	Permission check API summary and examples
POST	`/api/agent/execute`	Execute a gateway tool by name

1.4.2. Execute Request

The execute endpoint accepts a tool name and arguments:

```
{
  "tool": "query_find",
  "arguments": {
    "rootType": "Location",
    "query": "status:ACTIVE && city:Atlanta",
    "page": { "limit": 10, "skip": 0 }
  }
}
```

```
curl -sS -X POST \
  -H 'Content-Type: application/json' \
  localhost:8080/api/agent/execute \
  -d '{
    "tool": "query_find",
    "arguments": {
      "rootType": "Location",
      "query": "status:ACTIVE && city:Atlanta"
    }
  }'
```

The response shape matches the corresponding Query Gateway REST endpoint (e.g. the Collection envelope for `query_find`).

1.5. Per-Tenant Agent Configuration

Agent behavior can be customized per realm using MicroProfile Config properties:

```
# Run agent tools as a specific user in the "acme" realm
quantum.agent.tenant.acme.runAsUserId=agent-user@acme.com

# Only allow find and plan tools for the "acme" realm
quantum.agent.tenant.acme.enabledTools=query_find,query_plan,query_rootTypes

# Cap find results at 100 for this tenant
quantum.agent.tenant.acme.maxFindLimit=100
```

Configuration properties:

Property	Type	Description
`quantum.agent.tenant.<realm>.runAsUserId`	string	Optional userId whose security context is used for tool execution

Property	Type	Description
`quantum.agent.tenant.<realm>.enabledTools`	comma-separated	Optional list of tool names to expose (all six enabled when empty)
`quantum.agent.tenant.<realm>.maxFindLimit`	integer	Optional maximum number of results for `query_find`

The default implementation (`PropertyTenantAgentConfigResolver`) reads these from `application.properties` or environment variables. You can replace it by providing a CDI bean implementing `TenantAgentConfigResolver`.

1.5.1. Run-As Principal

When `runAsUserId` is configured, tool execution runs under that user's security context. To enable this, provide a CDI bean implementing `RunAsPrincipalResolver`:

```
import com.e2eq.framework.api.agent.RunAsPrincipalResolver;
import com.e2eq.framework.model.securityrules.PrincipalContext;
import jakarta.enterprise.context.ApplicationScoped;
import java.util.Optional;

@ApplicationScoped
public class MyRunAsPrincipalResolver implements RunAsPrincipalResolver {

    @Override
    public Optional<PrincipalContext> resolvePrincipalContext(String realm, String
userId) {
        // Look up user and build PrincipalContext
        // Return Optional.empty() to fall back to caller's context
    }
}
```

1.6. Agent Configuration API

The framework stores agent configurations as realm-scoped `Agent` entities in MongoDB (`agents` collection). Each agent pairs an LLM reference with a system prompt and a tool filter, enabling different personas for different use cases (e.g., a supply-chain analyst vs. a customer-service bot).

1.6.1. Agent Entity

Field	Type	Description
`refName`	string	Unique reference name within the realm (e.g., `supply-chain-assistant`)
`name`	string	Display name

Field	Type	Description
`llmConfigRef`	string	Reference to an LLM secret or config (e.g., `claude-sonnet`, `gpt-4o`)
`context`	list of PromptStep	Ordered system/user prompt steps that define the agent's persona
`enabledTools`	list of string	Tool names this agent can use (e.g., `["query_rootTypes", "query_find", "query_plan"]`)

Each `PromptStep` has `order` (int), `role` (string: `"system"` or `"user"`), and `content` (string: prompt text).

1.6.2. CRUD Endpoints

All endpoints are at `/api/agent/config` and require a `realm` query parameter.

Method	Path	Description
GET	`/api/agent/config/list?realm=X`	List all agents in the realm
GET	`/api/agent/config/{refName}?realm=X`	Get an agent by reference name
GET	`/api/agent/config/id/{id}?realm=X`	Get an agent by ObjectId
POST	`/api/agent/config?realm=X`	Create or update an agent
DELETE	`/api/agent/config/{refName}?realm=X`	Delete an agent by reference name
DELETE	`/api/agent/config/id/{id}?realm=X`	Delete an agent by ObjectId

1.6.3. Example: Creating an Agent

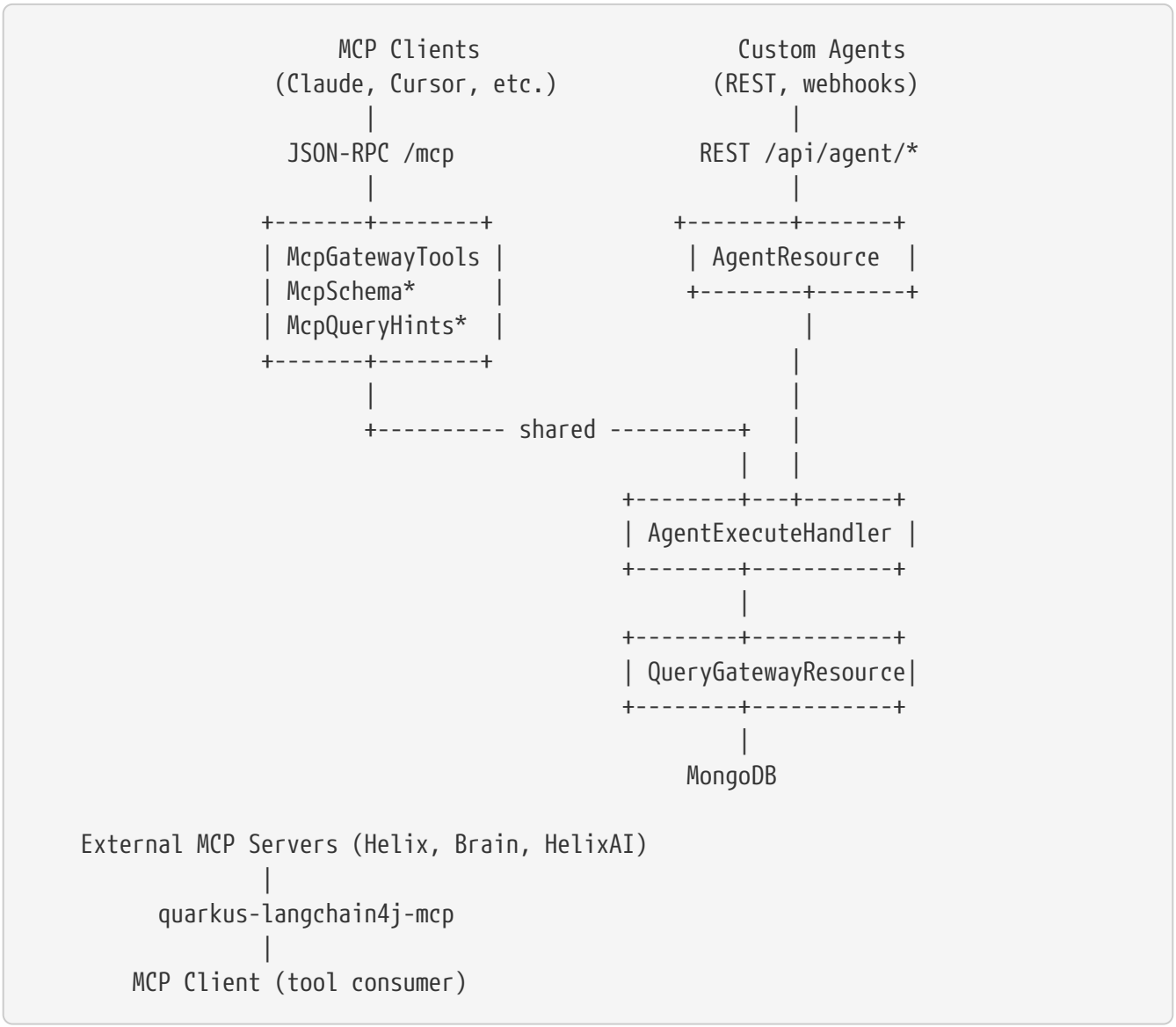
```
curl -sS -X POST \
  -H 'Content-Type: application/json' \
  'localhost:8080/api/agent/config?realm=acme-corp' \
  -d '{
    "refName": "supply-chain-assistant",
    "name": "Supply Chain Assistant",
    "llmConfigRef": "claude-sonnet",
    "context": [
      { "order": 1, "role": "system", "content": "You are a supply-chain analyst. Help
users query orders, track shipments, and monitor inventory." }
    ],
    "enabledTools": ["query_rootTypes", "query_plan", "query_find", "query_count"]
  }'
```

Agent configuration complements the property-based tenant configuration (Per-Tenant Agent Configuration). Properties set operator-level overrides (runAsUserId, maxFindLimit); Agent entities store per-use-case personas and tool filters that can

be managed via API.

1.7. Architecture

The following diagram shows how the MCP server, agent REST API, and MCP client relate:



Key design points:

- **Shared execution path**: Both MCP tools and REST agent endpoints delegate to `AgentExecuteHandler`, which routes to `QueryGatewayResource`. Security rules, realm resolution, and query execution are identical regardless of entry point.
- **Zero reverse dependencies**: The `quantum-mcp-server` module depends on `quantum-framework` but the framework has no knowledge of MCP. Applications that do not need MCP simply omit this module.
- **Tenant isolation**: Realm-scoped execution, optional per-tenant tool filtering, and run-as support ensure multi-tenant safety.

1.8. Ontology Tools

In addition to the six gateway tools, the MCP server exposes two ontology discovery tools via `McpOntologyTools`:

Tool Name	Description
`query_relationships`	Find ontology edges (relationships) for an entity — outgoing, incoming, or both, with optional predicate filtering
`query_predicates`	List ontology predicates (relationship types) defined in the TBox schema, with optional domain/range filtering

These tools allow AI assistants to explore the entity relationship graph. For example, an agent can discover that an `Order` entity has a `placedInOrg` relationship to an `Organization`, then use `query_find` with appropriate filters to retrieve related data.

1.9. Integrating with Claude

Claude supports MCP through Claude Desktop, Claude Code (CLI), and the browser-based Claude.ai (via Custom Connectors). The Quantum MCP server works with all three.

1.9.1. Claude Desktop

Edit `~/Library/Application Support/Claude/claude_desktop_config.json` (macOS) or the equivalent on your platform:

```
{
  "mcpServers": {
    "quantum": {
      "url": "http://localhost:8080/mcp"
    }
  }
}
```

Restart Claude Desktop. The Quantum tools appear in the tool picker and Claude can call them during conversations.

1.9.2. Claude Code (CLI)

Claude Code discovers MCP servers from project-level or user-level configuration.

Project-level — create `.claude/settings.json` in your repository root:

```
{
  "mcpServers": {
    "quantum-mcp": {
      "url": "http://localhost:8080/mcp"
```

```
    }
  }
}
```

User-level — edit `~/.claude/settings.local.json` to add the server globally:

```
{
  "mcpServers": {
    "quantum-mcp": {
      "url": "http://localhost:8080/mcp"
    }
  }
}
```

Once configured, Claude Code can use the Quantum tools in any conversation:

```
> Use query_rootTypes to list available entity types

Claude calls query_rootTypes and returns:
  Location, Order, CodeList, UserProfile, ...

> Find all active locations in the West region

Claude calls query_find with:
  rootType: "Location"
  query: "status:ACTIVE && region:West"
  limit: 10

Returns matching Location entities.
```

1.9.3. Claude.ai (Browser) with Custom Connectors

For cloud-hosted deployments, expose the `/mcp` endpoint over HTTPS and configure a Claude Custom Connector:

1. In Claude.ai, go to Settings > Integrations > Custom Connectors.
2. Add a new connector with your server URL (e.g., `https://api.mycompany.com/mcp`).
3. Configure authentication headers if required by your deployment.

1.9.4. Cursor IDE

Create `.cursor/mcp.json` in your project root:

```
{
  "mcpServers": {
    "quantum": {
```

```
      "url": "http://localhost:8080/mcp"
    }
  }
}
```

Or add the server URL in Cursor Settings > MCP Servers.

1.10. How MCP Maps to Framework Agent and Tool Concepts

The framework uses a layered architecture where MCP is one of several access paths to the same underlying tool execution engine. Understanding how these concepts relate helps when extending the framework or building custom integrations.

1.10.1. Concept Mapping

MCP Concept	Framework Concept	Relationship
MCP Tool	`@Tool`-annotated method	Each MCP tool is a Java method annotated with `@Tool` (Quarkiverse MCP Server extension) in `McpGatewayTools`. Tools are code-defined and discovered at startup via classpath scanning — there is no database-backed tool registry.
MCP Tool Name	Method-derived name	The tool name used in MCP `tools/call` (e.g., `query_find`) is defined in the `@Tool` annotation or derived from the method name.
MCP `tools/call`	`AgentExecuteHandler.execute(tool, arguments)`	Both the MCP server (`McpGatewayTools`) and the REST agent endpoint (`POST /api/agent/execute`) delegate to the same handler.
MCP Resource	`SchemaService` / `QueryHintsProvider`	MCP resources (`quantum://schema`, `quantum://query-hints`, `quantum://permission-hints`) are backed by the same provider beans that serve the REST `/api/agent/schema` and `/api/agent/query-hints` endpoints.
MCP Client	`quarkus-langchain4j-mcp`	The framework can also *consume* tools from external MCP servers. External MCP servers are configured via `ToolProviderConfig` (connection details) and `quarkus.langchain4j.mcp.*` properties.
Agent Config	`Agent` entity	An `Agent` is a realm-scoped configuration envelope that pairs an LLM reference with a system prompt (PromptSteps) and a tool filter (enabledTools). Managed via `/api/agent/config/*` CRUD API.

1.10.2. Execution Flow Comparison

Both MCP and REST reach the same execution path:

```
MCP Client                          REST Client
    |                                    |
tools/call "query_find"          POST /api/agent/execute
    |                            { "tool":"query_find", ... }
    v                                    v
McpGatewayTools                   AgentResource
    |                                    |
    +--------> AgentExecuteHandler <------+
                    |
                    v
            QueryGatewayResource
                    |
                    v
                MongoDB
```

Security (realm resolution, permission checks, `@FunctionalAction`) is enforced identically at the `QueryGatewayResource` level, regardless of whether the request came through MCP or REST.

1.11. End-to-End Example: Claude Querying a Supply Chain Application

This walkthrough demonstrates a complete flow from starting the application to having Claude interact with live data through MCP.

1.11.1. Prerequisites

- Java 17+, Maven 3.9+
- MongoDB running locally (or via Quarkus Dev Services)
- Claude Desktop or Claude Code installed
- Your Quantum-based application (e.g., a supply chain app with `Location`, `Order`, `Shipment` entities)

1.11.2. Step 1: Add the MCP Server Module

In your application POM, add the `quantum-mcp-server` dependency:

```
<dependency>
    <groupId>com.end2endlogic</groupId>
    <artifactId>quantum-mcp-server</artifactId>
    <version>${quantum.version}</version>
</dependency>
```

1.11.3. Step 2: Configure Agent Properties

In `src/main/resources/application.properties`:

```
# --- Authentication ---
auth.provider=custom
quarkus.smallrye-jwt.enabled=true
mp.jwt.verify.publickey.location=publicKey.pem
mp.jwt.verify.issuer=https://myapp.example.com/issuer
mp.jwt.verify.audiences=my-api-client
auth.jwt.secret=${JWT_SECRET:dev-secret-change-in-prod}
auth.jwt.expiration=60
auth.jwt.refresh-expiration=120

# --- Per-tenant agent configuration ---
# The "default" realm: agent runs as a service user, limited to read-only tools
quantum.agent.tenant.default.runAsUserId=agent-svc@mycompany.com
quantum.agent.tenant.default.enabledTools=query_rootTypes,query_plan,query_find,query_
count
quantum.agent.tenant.default.maxFindLimit=200

# The "acme-corp" realm: full CRUDL access
quantum.agent.tenant.acme-corp.runAsUserId=bot@acme-corp.com
quantum.agent.tenant.acme-
corp.enabledTools=query_rootTypes,query_plan,query_find,query_count,query_save,query_d
elete
```

1.11.4. Step 3: Start the Application

```
mvn quarkus:dev
```

The MCP endpoint is available at `http://localhost:8080/mcp`.

1.11.5. Step 4: Configure Claude

Claude Desktop — edit `claude_desktop_config.json`:

```
{
  "mcpServers": {
    "supply-chain": {
      "url": "http://localhost:8080/mcp"
    }
  }
}
```

Claude Code — create `.claude/settings.json` in the project root:

```
{
  "mcpServers": {
    "supply-chain": {
      "url": "http://localhost:8080/mcp"
    }
  }
}
```

1.11.6. Step 5: Interact with Claude

Below is a realistic conversation showing how Claude uses the MCP tools:

User: What entity types are available in the supply chain system?

Claude calls `query_rootTypes` and responds:

> The system has these entity types: Location, Order, Shipment, Product, Customer, CodeList, UserProfile, Organization.

User: Show me all active warehouses in the Southeast region.

Claude calls `query_find`:

- `rootType`: `"Location"`
- `query`: `"status:ACTIVE && type:WAREHOUSE && region:Southeast"`
- `limit`: `20`

> Found 3 active warehouses in the Southeast:
>
> 1. ATL-WH-01 — Atlanta Distribution Center (Atlanta, GA)
> 2. MIA-WH-03 — Miami Fulfillment Hub (Miami, FL)
> 3. CLT-WH-02 — Charlotte Regional Warehouse (Charlotte, NC)

User: How many open orders are there for the Atlanta warehouse?

Claude calls `query_count`:

- `rootType`: `"Order"`
- `query`: `"status:OPEN && warehouseRef:ATL-WH-01"`

> There are 47 open orders for the Atlanta Distribution Center (ATL-WH-01).

User: Show me the last 5 orders with their customer details expanded.

Claude calls `query_plan` first to check expand support:

- `rootType`: `"Order"`
- `query`: `"expand(customer) && warehouseRef:ATL-WH-01"`

Then calls `query_find`:

- `rootType`: `"Order"`
- `query`: `"expand(customer) && warehouseRef:ATL-WH-01"`
- `limit`: `5`

> Here are the 5 most recent orders for ATL-WH-01, with customer details:
>
> 1. ORD-9921 — Acme Corp (acme@example.com), 12 items, $2,340
> 2. ORD-9918 — Beta Industries (beta@example.com), 3 items, $890 …

User: What relationships does the Atlanta warehouse have?

Claude calls `query_relationships`:

- `entityId`: `"ATL-WH-01"`
- `direction`: `"both"`

> The Atlanta warehouse has these relationships:
>
> Outgoing: - `locatedIn` → Organization "Southeast Division" - `servesRegion` → Region "Southeast"
>
> Incoming: - `fulfilledAt` ← 47 open Orders - `shipsFrom` ← 12 active Shipments

1.11.7. What Happened Under the Hood

```
Claude Desktop / Claude Code
    |
    | (1) MCP tools/list -> discovers 9 tools + 3 resources
    |
    | (2) MCP resources/read "quantum://query-hints"
    |     -> learns BIAPI query syntax, expand(), ontology edges
    |
    | (3) MCP tools/call "query_rootTypes"
    |     -> McpGatewayTools.query_rootTypes()
    |        -> AgentExecuteHandler.execute("query_rootTypes", {})
    |           -> QueryGatewayResource.listRootTypes()
    |
    | (4) MCP tools/call "query_find" { rootType, query, limit }
    |     -> McpGatewayTools.query_find(rootType, query, realm, limit, skip)
    |        -> AgentExecuteHandler.execute("query_find", args)
    |           -> QueryGatewayResource.find(FindRequest)
```

```
    |              -> Morphia Query -> MongoDB
    |
    | (5) MCP tools/call "query_relationships" { entityId, direction }
    |     -> McpOntologyTools.query_relationships(entityId, direction, ...)
    |        -> OntologyEdgeRepo.findBySrc() / findByDst()
    |           -> MongoDB edges collection
    |
    v
JSON response serialized and returned to Claude
```

At every step, the Quantum security filter validates the caller's identity, resolves the realm, and applies permission rules. The agent never bypasses `@FunctionalAction` checks or data scoping.

1.11.8. Adding Ontology Context for Richer Queries

When your application has an ontology TBox defined (see Ontologies in Quantum), Claude can use the `query_predicates` tool to discover relationship types and then use `query_relationships` to traverse the graph.

For example:

User: What types of relationships exist between Orders and Organizations?

Claude calls `query_predicates`:

- `domainFilter`: `"Order"`
- `rangeFilter`: `"Organization"`

> Found 2 predicates:
>
> 1. `placedByOrg` — Order → Organization (the org that placed the order)
> 2. `fulfilledByOrg` — Order → Organization (the org fulfilling the order)

This lets Claude build more targeted queries and provide richer answers about entity relationships.

1.12. MCP Server Configuration Reference

The MCP server requires no additional configuration beyond including the `quantum-mcp-server` module. The `/mcp` endpoint is automatically registered by the Quarkus MCP Server extension.

Optional properties:

Property	Default	Description
`quarkus.mcp.server.traffic-logging`	`NONE`	Set to `ALL` to log MCP JSON-RPC messages for debugging
`feature.queryGateway.execution.enabled`	`false`	Set to `true` to enable `expand()` (AGGREGATION mode) in `query_find`

Property	Default	Description
`quantum.queryGateway.deleteMany.maxMatches`	`2000`	Maximum entities that `query_deleteMany` will delete in a single call

See AI Agent Integration for the full agent API design and additional scenarios.