# Table of Contents

1. MCP Server and Client 1
  1.1. Module Overview 1
  1.2. MCP Server 1
    1.2.1. Tools 1
    1.2.2. Resources 2
    1.2.3. Connecting an MCP Client 3
  1.3. MCP Client 3
    1.3.1. Configuration 3
    1.3.2. Injecting MCP Tools 4
  1.4. Agent REST API 4
    1.4.1. Endpoints 4
    1.4.2. Execute Request 5
  1.5. Per-Tenant Agent Configuration 5
    1.5.1. Run-As Principal 6
  1.6. Architecture 6

# 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:

```xml
<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`:

```java
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:

```java
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:

```json
{
  "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:

```java
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. Architecture

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

```
                        MCP Clients                    Custom Agents
                   (Claude, Cursor, etc.)            (REST, webhooks)
                          |                                |
                    JSON-RPC /mcp                   REST /api/agent/*
                          |                                |
                  +-------+--------+              +--------+-------+
                  | McpGatewayTools |              | AgentResource  |
                  | McpSchema*      |              +--------+-------+
                  | McpQueryHints*  |                       |
                  +-------+--------+                       |
                          |                                |
                          +---------- shared ----------+    |
                                                       |    |
                                              +--------+---+-------+
                                              | AgentExecuteHandler |
                                              +--------+-----------+
                                                       |
                                              +--------+-----------+
                                              | QueryGatewayResource|
                                              +--------+-----------+
                                                       |
                                                   MongoDB

    External MCP Servers (Helix, Brain, HelixAI)
                  |
          quarkus-langchain4j-mcp
                  |
        MCP Client (tool consumer)
```

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.