Table of Contents

Chapter 1. Testing in Quantum: Security Contexts, Repos, and REST APIs

1.1. Testing Framework

This guide explains how to write effective tests in the Quantum framework, with a focus on setting the security context and choosing the right Quarkus testing patterns for repository logic vs. REST APIs.

It covers four practical patterns you can use to establish the security context in tests: - Extending BaseRepoTest - Using a try-with-resources SecuritySession - Using a scoped-call pattern to wrap work under a SecuritySession - Using the @TestSecurity annotation

In addition, it outlines how to test REST endpoints vs. repository logic and highlights useful features of the Quarkus Test Framework.

1.2. Prerequisites and glossary

- RuleContext: the in-memory rule engine configuration used by authorization checks.
- PrincipalContext (pContext): who is performing the action (userId, roles, realms, data domain).
- ResourceContext (rContext): what is being acted upon (area/domain, resource, action).
- SecuritySession: a small utility that binds PrincipalContext and ResourceContext to SecurityContext for the current thread, and clears them on close.
- SecurityIdentity: Quarkus' current identity (used by @TestSecurity and HTTP request security).

1.3. Pattern 1 — Extend BaseRepoTest

For repository tests (no HTTP), the simplest approach is to extend BaseRepoTest.

What BaseRepoTest does for you: - Initializes RuleContext with default rules. - Builds default pContext and rContext for a system/test user. - Ensures test database migrations are applied. - Gives you protected fields pContext and rContext you can use in your test.

Example:

```
@QuarkusTest
class MyRepoTest extends com.e2eq.framework.persistent.BaseRepoTest {

  @Inject
  com.e2eq.framework.model.persistent.morphia.UserProfileRepo userProfileRepo;

  @Test
  void canReadUnderTestUser() {
    // Activate the security context for this block
    try (final com.e2eq.framework.securityrules.SecuritySession ignored =
```

```
            new com.e2eq.framework.securityrules.SecuritySession(pContext, rContext))
{
      var list = userProfileRepo.list(testUtils.getTestRealm());
      org.junit.jupiter.api.Assertions.assertNotNull(list);
    }
  }
}
```

Notes: - BaseRepoTest prepares contexts but does not keep them permanently active. Wrap repo calls that require authorization in a SecuritySession (see Pattern 2), or activate it in @BeforeEach if many test methods use it. - BaseRepoTest also runs migrations once using your test principal, which avoids authorization failures during initialization.

1.4. Pattern 2 — Try-with-resources SecuritySession

Use SecuritySession explicitly to scope work that should run under a specific PrincipalContext and ResourceContext. This is the most explicit pattern and works in both repository and service-level tests.

Example:

```
@QuarkusTest
class CredentialRepoTest extends com.e2eq.framework.persistent.BaseRepoTest {

  @Inject
  com.e2eq.framework.model.persistent.morphia.CredentialRepo credRepo;

  @Test
  void findByUserId_asTestUser() {
    try (final com.e2eq.framework.securityrules.SecuritySession s =
            new com.e2eq.framework.securityrules.SecuritySession(pContext, rContext))
{
      var op = credRepo.findByUserId(testUtils.getTestUserId(), testUtils
.getSystemRealm());
      org.junit.jupiter.api.Assertions.assertTrue(op.isPresent());
    }
  }
}
```

Tips: - Prefer try-with-resources so contexts are always cleared, even when assertions fail. - You can construct custom PrincipalContext/ResourceContext for specific scenarios (e.g., different roles or realms) and pass them to SecuritySession.

1.5. Pattern 3 — Scoped-call wrapper (ScopedCallScope)

If you prefer not to repeat try-with-resources blocks, wrap your work in a helper that creates a

SecuritySession, runs your logic, and ensures cleanup. This is sometimes called a “scoped call” pattern, often referred to as a ScopedCallScope.

Example helper (placed in test sources):

```
public final class SecurityScopes {
  private SecurityScopes() {}

  public static <T> T call(
      com.e2eq.framework.model.securityrules.PrincipalContext p,
      com.e2eq.framework.model.securityrules.ResourceContext r,
      java.util.concurrent.Callable<T> work) {
    try (final com.e2eq.framework.securityrules.SecuritySession s =
             new com.e2eq.framework.securityrules.SecuritySession(p, r)) {
      try { return work.call(); }
      catch (Exception e) { throw new RuntimeException(e); }
    }
  }

  public static void run(
      com.e2eq.framework.model.securityrules.PrincipalContext p,
      com.e2eq.framework.model.securityrules.ResourceContext r,
      Runnable work) {
    try (final com.e2eq.framework.securityrules.SecuritySession s =
             new com.e2eq.framework.securityrules.SecuritySession(p, r)) {
      work.run();
    }
  }
}
```

Usage:

```
var result = SecurityScopes.call(pContext, rContext, () -> repo.getByUserId(realm,
userId));
SecurityScopes.run(pContext, rContext, () -> repo.save(realm, entity));
```

This achieves the same effect as try-with-resources but centralizes the pattern.

1.6. Pattern 4 — @TestSecurity annotation (Quarkus)

For tests that run through HTTP (and in some repo tests), you can use Quarkus’ @io.quarkus.test.security.TestSecurity to set the SecurityIdentity without creating a SecuritySession.

Quantum integrates with this in two places: - SecurityFilter: when a request has a SecurityIdentity but no JWT, it builds a PrincipalContext from the identity (user and roles). You can also pass X-Realm to control the realm. - MorphiaRepo: when repo methods are invoked under @TestSecurity with no active SecuritySession, MorphiaRepo lazily builds PrincipalContext from SecurityIdentity and sets a safe default ResourceContext to enable rule evaluation.

Example (HTTP):

```
@QuarkusTest
class SecureResourceTest {

  @Inject com.e2eq.framework.util.TestUtils testUtils;

  @Test
  @io.quarkus.test.security.TestSecurity(user = "test@system.com", roles = {"user"})
  void listProfiles_asUser() {
    io.restassured.RestAssured.given()
      .header("X-Realm", testUtils.getTestRealm())
      .when().get("/user/userProfile/list")
      .then().statusCode(200);
  }
}
```

Example (repo call under @TestSecurity fallback, no SecuritySession):

```
@QuarkusTest
class RepoFallbackTest {

  @Inject com.e2eq.framework.util.TestUtils testUtils;
  @Inject com.e2eq.framework.model.persistent.morphia.CredentialRepo credentialRepo;

  @Test
  @io.quarkus.test.security.TestSecurity(user = "test@system.com", roles = {"user"})
  void repoUsesIdentityWhenNoSecuritySession() {
    // Internally, MorphiaRepo will ensure PrincipalContext exists using
SecurityIdentity
    credentialRepo.findByUserId("nonexistent@end2endlogic.com", testUtils.
getTestRealm(), false);
    // Optionally assert that SecurityContext has been initialized
    org.junit.jupiter.api.Assertions.assertTrue(
      com.e2eq.framework.model.securityrules.SecurityContext.getPrincipalContext()
.isPresent());
  }
}
```

Notes: - @TestSecurity is perfect for authorizing requests in HTTP tests without generating JWTs. - For repo tests that require precise ResourceContext (area/domain/action), prefer SecuritySession; MorphiaRepo sets a generic default ResourceContext when needed.

1.7. Testing REST APIs vs. Repository Logic

When to prefer REST (HTTP) tests: - End-to-end authorization: validate request filters, identity mapping, realm headers, and JWT handling. - Request/response shape and status codes. - Role-based access checks via @TestSecurity.

How to test REST APIs: - Use @QuarkusTest and RestAssured: [source,java] ---- var resp = io.restassured.RestAssured.given() .header("Content-Type", "application/json") .header("X-Realm", testUtils.getTestRealm()) .when().get("/user/userProfile/list") .then().statusCode(200).extract().response(); ---- - To test JWT-protected endpoints end-to-end, first call the login API to obtain a token, then pass Authorization: Bearer <token>. See SecurityTest.testGetUserProfileRESTAPI for a complete example.

When to prefer repository/service tests: - You want precise control over PrincipalContext/ResourceContext and rule evaluation without HTTP overhead. - You are asserting persistence logic, query filters, or domain rules.

How to test repository logic: - Extend BaseRepoTest (Pattern 1) for ready-to-use pContext/rContext and migrations. - Wrap calls with SecuritySession (Pattern 2) or use a scoped-call helper (Pattern 3).

1.8. Useful Quarkus Test features

- @QuarkusTest: boots the app for integration tests with CDI, config, and persistence.
- RestAssured: fluent HTTP client baked into Quarkus tests; supports JSON assertions and extraction.
- @TestSecurity: set SecurityIdentity (user, roles) for tests.
- @InjectMock/@InjectSpy (quarkus-junit5-mockito): replace beans with mocks/spies for isolation.
- @QuarkusTestResource: manage external resources (e.g., starting/stopping containers) for a test class or suite.
- @TestHTTPEndpoint and @TestHTTPResource: convenient endpoint URI injection.

1.9. Real-world tips

- Clearing thread locals: If you manipulate SecurityContext directly in advanced tests, clear it in @AfterEach to avoid cross-test leakage: [source,java] ---- @AfterEach void cleanup() { com.e2eq.framework.model.securityrules.SecurityContext.clear(); } ----
- Realm routing: pass X-Realm in REST tests to select the target realm. SecurityFilter also validates realm access against user credentials when present.
- Data prep: If your test needs specific users/roles, create them under a SecuritySession beforehand (see SecurityTest.ensureTestUserExists()).
- Logging: enable DEBUG for com.e2eq to inspect rule evaluation and identity resolution during tests.

1.10. Summary

- Use BaseRepoTest for repository tests and migrations, and wrap work in SecuritySession.
- For less ceremony, create a simple scoped-call helper to run code under a SecuritySession.
- For REST/API tests and quick identity setup, use @TestSecurity, realm headers, and RestAssured.
- For full e2e security, obtain a JWT via the login API and include it in requests.