

QVTM SDK Reference

Public Integration Contracts, Profiles, and Developer Workflows

Status: Final Release **Scope:** SDK Reference **Companion documents:** QVTM Position Paper v1.0, QVTM White Paper v1.0, QVTM Protocol Specification v1.0 **Language:** English **Audience:** Software Developers, SDK Integrators, Solution Architects **Document Class:** Public Developer Documentation
Authors: Fernando Guerrero Patricio Guerrero

July 2026

Copyright (c) 2026

Copyright

Copyright © QVTM Project.

All rights reserved.

This document describes the public software integration model of QVTM.

No part of this document describes proprietary implementation details, internal protocol algorithms, or implementation-specific optimizations.

The authoritative behavioral definition of the protocol remains the QVTM Protocol Specification.

Document Status

This document is the frozen QVTM SDK Reference v1.0.

It describes the public programming interface exposed by QVTM SDK implementations.

Until version 1.0 is frozen, minor editorial improvements may occur without changing documented public contracts.

Intended Audience

This document is intended for:

- application developers;
- SDK integrators;
- solution architects;
- platform engineers;
- software engineering teams adopting QVTM.

Readers are expected to be familiar with general software development concepts.

Knowledge of the internal QVTM protocol is not required.

Purpose

This document explains how to integrate and use the QVTM SDK.

It intentionally focuses on the public programming model exposed by SDK implementations.

Unlike the accompanying architectural documentation, this document does not explain why QVTM exists or how the protocol operates internally.

Instead, it answers a single practical question:

Scope

This document describes:

- SDK architecture;
- public SDK objects;
- SDK operations;
- Deployment Profiles;
- integration workflows;
- error handling;
- compatibility;
- examples;
- developer best practices.

This document does **not** describe:

- protocol internals;
- Graph Puzzle;
- Trust Evaluation algorithms;
- Continuity correlation;
- proprietary implementation details;
- cryptographic implementation strategies.

Relationship to Other Documents

The QVTM documentation set is organized into four complementary documents.

Document	Primary Question
Position Paper	Why does QVTM exist?
White Paper	How does QVTM work conceptually?
Protocol Specification	What observable protocol behavior is defined?
SDK Reference	How is QVTM integrated into software?

These documents are intended to be read together.

This document references the others rather than duplicating their content.

How to Use This Document

Readers new to QVTM SHOULD read the chapters sequentially.

Developers already familiar with the SDK MAY use individual chapters as reference material.

The recommended reading order is:

1. SDK Overview
2. Core SDK
3. Operations
4. Deployment Profiles
5. Integration
6. Examples
7. Appendices

Table of Contents

- [Part I — SDK Overview](#part-i--sdk-overview)
- [1. Introduction](#1-introduction)
- [2. SDK Design Principles](#2-sdk-design-principles)
- [3. SDK Architecture](#3-sdk-architecture)
- [4. SDK Concepts](#4-sdk-concepts)
- [5. Developer Workflow](#5-developer-workflow)
- [Part II — Core SDK](#part-ii--core-sdk)
- [6. Client](#6-client)
- [7. Session](#7-session)
- [8. Identity](#8-identity)
- [9. Continuity Evidence](#9-continuity-evidence)
- [10. Continuity Evidence Package (CEP)](#10-continuity-evidence-package-cep)
- [11. Trust Evaluation](#11-trust-evaluation)
- [Part III — Operations](#part-iii--operations)
- [12. Enrollment](#12-enrollment)
- [13. Authentication](#13-authentication)
- [14. Synchronization](#14-synchronization)
- [15. Recovery](#15-recovery)
- [16. Vault Operations](#16-vault-operations)
- [Part IV — Profiles](#part-iv--profiles)
- [17. Login Profile](#17-login-profile)
- [18. Vault Profile](#18-vault-profile)
- [19. Secure Messaging Profile](#19-secure-messaging-profile)
- [20. Secure Upload Profile](#20-secure-upload-profile)
- [Part V — Integration](#part-v--integration)
- [21. SDK Lifecycle](#21-sdk-lifecycle)
- [22. Error Handling](#22-error-handling)
- [23. Version Compatibility](#23-version-compatibility)
- [24. Best Practices](#24-best-practices)
- [25. Performance Considerations](#25-performance-considerations)
- [Part VI — Examples](#part-vi--examples)
- [26. Basic Authentication](#26-basic-authentication)
- [27. File Protection](#27-file-protection)
- [28. Secure Messaging](#28-secure-messaging)
- [29. Synchronization](#29-synchronization)
- [30. Recovery](#30-recovery)
- [Part VII — Published SDK API Reference](#part-vii--published-sdk-api-reference)
- [31. Published Package Inventory](#31-published-package-inventory)
- [32. @qvm/control-plane](#32-qvmcontrol-plane)
- [33. @qvm/trust-verifier](#33-qvmtrust-verifier)
- [34. @qvm/runtime-protocol](#34-qvmruntime-protocol)
- [35. @qvm/vault](#35-qvmvault)
- [36. Operation-to-Response Traceability](#36-operation-to-response-traceability)
- [37. Public Documentation Boundary](#37-public-documentation-boundary)
- [Appendix A — Error Codes](#appendix-a-error-codes)
- [Appendix B — Common Integration Patterns](#appendix-b-common-integration-patterns)
- [Appendix C — Migration Guidelines](#appendix-c-migration-guidelines)

Reference Navigation

Package Quick Links

Package	Methods	Public object contracts	Chapter
[@qvtm/control-plane](#32-qvtmcontrol-plane)	51	52	32
[@qvtm/trust-verifier](#33-qvtmtrust-verifier)	6	9	33
[@qvtm/runtime-protocol](#34-qvtmruntime-protocol)	9	12	34
[@qvtm/vault](#35-qvtmvault)	14	13	35

Operation Index

@qvtm/control-plane: health, getRuntimeStatus, runtimeStatus, listAuthScopes, startLoginSession, getLoginSessionStatus, issueAuthChallenge, getAuthChallenge, getAuthChallengeStatus, verifyAuthChallenge, consumeAuthChallengeSessionHandle, consumeSessionHandle, consumeAuthChallengeHandoff, getSessionDiagnostics, revokeCurrentSession, getChallengeCompatibilityPayload, getChallengeCompatibilityStatus, verifyChallengeCompatibilityApproval, consumeChallengeCompatibilitySessionHandle, consumeChallengeCompatibilityHandoff, listControlPlaneTenants, upsertControlPlaneTenant, createTenant, onboardControlPlaneTenant, getControlPlaneTenant, transitionControlPlaneTenantLifecycle, getControlPlaneTenantSummary, listCustomerInvites, createCustomerInvite, listPackageDeliveries, createPackageDelivery, revokePackageDelivery, provisionTenantCustomerInvite, provisionTenantPackageDelivery, listDeviceTrustBundles, exportDeviceTrustBundleStatusList, exportSignedDeviceTrustBundleStatusList, exportDeviceTrustBundleAuditTrail, exportDeviceTrustBundleAuditTrailForBundle, exportDeviceTrustBundleEvidencePackage, exportDeviceTrustBundleReadinessReport, exportDeviceTrustBundleDistributionPackage, exportDeviceTrustBundleDistributionManifest, exportDeviceTrustDistributionClosureReport, exportDeviceTrustPhase61FinalReadinessReport, issueDeviceTrustBundle, getDeviceTrustBundle, exportDeviceTrustBundle, revokeDeviceTrustBundle, rotateDeviceTrustBundle, pollAuthChallengeStatus.

@qvtm/trust-verifier: capabilities, inspectEvidence, verifyEvidence, getArtifactPackage, verifyArtifactPackage, getTransparencyExportEnvelope.

@qvtm/runtime-protocol: version, registerDevice, getDevice, startHandshake, finishHandshake, sendMessage, getTransparencyKey, getTransparencyState, listTransparencyHistory.

@qvtm/vault: listDevices, createVault, openVault, importVault, exportVaultCopy, removeLocalVault, requestIdentityApproval, syncVault, onEvent, assertVaultFile, assertVaultFile, assertSafeVaultFileName, assertSafeLabel, assertSafeLabel.

Public Type Index

Type	Package
AdminWriteResponse	[@qvtm/control-plane](#32-qvtmcontrol-plane)
AuthChallengePollOptions	[@qvtm/control-plane](#32-qvtmcontrol-plane)
AuthChallengeResponse	[@qvtm/control-plane](#32-qvtmcontrol-plane)
AuthChallengeStatusResponse	[@qvtm/control-plane](#32-qvtmcontrol-plane)
ChallengeVerificationEvidence	[@qvtm/control-plane](#32-qvtmcontrol-plane)
CustomerInviteCreateRequest	[@qvtm/control-plane](#32-qvtmcontrol-plane)
CustomerInviteResponse	[@qvtm/control-plane](#32-qvtmcontrol-plane)
DeviceTrustBundleAuditEvent	[@qvtm/control-plane](#32-qvtmcontrol-plane)
DeviceTrustBundleAuditTrailResponse	[@qvtm/control-plane](#32-qvtmcontrol-plane)
DeviceTrustBundleDistributionManifestResponse	[@qvtm/control-plane](#32-qvtmcontrol-plane)
DeviceTrustBundleDistributionPackageResponse	[@qvtm/control-plane](#32-qvtmcontrol-plane)
DeviceTrustBundleEvidencePackageResponse	[@qvtm/control-plane](#32-qvtmcontrol-plane)
DeviceTrustBundleEvidenceReceipt	[@qvtm/control-plane](#32-qvtmcontrol-plane)
DeviceTrustBundleReadinessCheck	[@qvtm/control-plane](#32-qvtmcontrol-plane)
DeviceTrustBundleReadinessReportResponse	[@qvtm/control-plane](#32-qvtmcontrol-plane)

Type	Package
DeviceTrustBundleRecord	[@qvtm/control-plane](#32-qvtmcontrol-plane)
DeviceTrustBundleResponse	[@qvtm/control-plane](#32-qvtmcontrol-plane)
DeviceTrustBundleStatusListResponse	[@qvtm/control-plane](#32-qvtmcontrol-plane)
DeviceTrustBundleStatusListSignature	[@qvtm/control-plane](#32-qvtmcontrol-plane)
DeviceTrustBundleStatusSnapshot	[@qvtm/control-plane](#32-qvtmcontrol-plane)
DeviceTrustDistributionClosureCheck	[@qvtm/control-plane](#32-qvtmcontrol-plane)
DeviceTrustDistributionClosureReportResponse	[@qvtm/control-plane](#32-qvtmcontrol-plane)
DeviceTrustPhase61FinalReadinessReportResponse	[@qvtm/control-plane](#32-qvtmcontrol-plane)
DeviceTrustPhase61ReadinessSignal	[@qvtm/control-plane](#32-qvtmcontrol-plane)
HealthResponse	[@qvtm/control-plane](#32-qvtmcontrol-plane)
IssueAuthChallengeRequest	[@qvtm/control-plane](#32-qvtmcontrol-plane)
IssueDeviceTrustBundleRequest	[@qvtm/control-plane](#32-qvtmcontrol-plane)
ListDeviceTrustBundlesResponse	[@qvtm/control-plane](#32-qvtmcontrol-plane)
LoginSessionResponse	[@qvtm/control-plane](#32-qvtmcontrol-plane)
PackageDeliveryCreateRequest	[@qvtm/control-plane](#32-qvtmcontrol-plane)
PackageDeliveryResponse	[@qvtm/control-plane](#32-qvtmcontrol-plane)
ProvisionTenantCustomerInviteWorkflowRequest	[@qvtm/control-plane](#32-qvtmcontrol-plane)
ProvisionTenantCustomerInviteWorkflowResponse	[@qvtm/control-plane](#32-qvtmcontrol-plane)
QvtmApiErrorBody	[@qvtm/control-plane](#32-qvtmcontrol-plane)
QvtmControlPlaneClientOptions	[@qvtm/control-plane](#32-qvtmcontrol-plane)
QvtmRuntimeClientOptions	[@qvtm/runtime-protocol](#34-qvtmruntime-protocol)
QvtmTrustVerifierClientOptions	[@qvtm/trust-verifier](#33-qvtmtrust-verifier)
QvtmVaultClientOptions	[@qvtm/vault](#35-qvtmvault)
QvtmVaultCreateRequest	[@qvtm/vault](#35-qvtmvault)
QvtmVaultDescriptor	[@qvtm/vault](#35-qvtmvault)
QvtmVaultDevice	[@qvtm/vault](#35-qvtmvault)
QvtmVaultEvent	[@qvtm/vault](#35-qvtmvault)
QvtmVaultExportCopyRequest	[@qvtm/vault](#35-qvtmvault)
QvtmVaultFileRef	[@qvtm/vault](#35-qvtmvault)
QvtmVaultIdentityApprovalRequest	[@qvtm/vault](#35-qvtmvault)
QvtmVaultImportRequest	[@qvtm/vault](#35-qvtmvault)
QvtmVaultOpenRequest	[@qvtm/vault](#35-qvtmvault)
QvtmVaultOperationReceipt	[@qvtm/vault](#35-qvtmvault)
QvtmVaultSession	[@qvtm/vault](#35-qvtmvault)
QvtmVaultSyncRequest	[@qvtm/vault](#35-qvtmvault)
QvtmWorkflowIdempotencyKeys	[@qvtm/control-plane](#32-qvtmcontrol-plane)

Type	Package
RevokeDeviceTrustBundleRequest	[@qvtm/control-plane](#32-qvtmcontrol-plane)
RotateDeviceTrustBundleRequest	[@qvtm/control-plane](#32-qvtmcontrol-plane)
RuntimeDeviceRegistrationRequest	[@qvtm/runtime-protocol](#34-qvtmruntime-protocol)
RuntimeHandshakeFinishRequest	[@qvtm/runtime-protocol](#34-qvtmruntime-protocol)
RuntimeHandshakeResponse	[@qvtm/runtime-protocol](#34-qvtmruntime-protocol)
RuntimeHandshakeStartRequest	[@qvtm/runtime-protocol](#34-qvtmruntime-protocol)
RuntimeHealthResponse	[@qvtm/runtime-protocol](#34-qvtmruntime-protocol)
RuntimeMessageSendRequest	[@qvtm/runtime-protocol](#34-qvtmruntime-protocol)
RuntimeMessageSendResponse	[@qvtm/runtime-protocol](#34-qvtmruntime-protocol)
RuntimeStatusResponse	[@qvtm/control-plane](#32-qvtmcontrol-plane)
RuntimeTransparencyHistoryResponse	[@qvtm/runtime-protocol](#34-qvtmruntime-protocol)
RuntimeTransparencyKeyResponse	[@qvtm/runtime-protocol](#34-qvtmruntime-protocol)
RuntimeTransparencyStateResponse	[@qvtm/runtime-protocol](#34-qvtmruntime-protocol)
RuntimeVersionResponse	[@qvtm/runtime-protocol](#34-qvtmruntime-protocol)
SessionDiagnosticsResponse	[@qvtm/control-plane](#32-qvtmcontrol-plane)
SessionHandleConsumeRequest	[@qvtm/control-plane](#32-qvtmcontrol-plane)
SessionHandleResponse	[@qvtm/control-plane](#32-qvtmcontrol-plane)
SignedDeviceTrustBundleStatusListResponse	[@qvtm/control-plane](#32-qvtmcontrol-plane)
StartLoginSessionRequest	[@qvtm/control-plane](#32-qvtmcontrol-plane)
TenantLifecycleTransitionRequest	[@qvtm/control-plane](#32-qvtmcontrol-plane)
TenantOnboardingRequest	[@qvtm/control-plane](#32-qvtmcontrol-plane)
TenantOnboardingResponse	[@qvtm/control-plane](#32-qvtmcontrol-plane)
TenantRecord	[@qvtm/control-plane](#32-qvtmcontrol-plane)
TenantSummaryResponse	[@qvtm/control-plane](#32-qvtmcontrol-plane)
TenantUpsertRequest	[@qvtm/control-plane](#32-qvtmcontrol-plane)
TransparencyExportEnvelopeResponse	[@qvtm/trust-verifier](#33-qvtmtrust-verifier)
TrustArtifactPackageResponse	[@qvtm/trust-verifier](#33-qvtmtrust-verifier)
TrustArtifactPackageVerifyRequest	[@qvtm/trust-verifier](#33-qvtmtrust-verifier)
TrustArtifactPackageVerifyResponse	[@qvtm/trust-verifier](#33-qvtmtrust-verifier)
TrustCapabilitiesResponse	[@qvtm/trust-verifier](#33-qvtmtrust-verifier)
TrustEvidenceInspectRequest	[@qvtm/trust-verifier](#33-qvtmtrust-verifier)
TrustEvidenceInspectResponse	[@qvtm/trust-verifier](#33-qvtmtrust-verifier)
TrustHealthResponse	[@qvtm/trust-verifier](#33-qvtmtrust-verifier)
VerifyAuthChallengeRequest	[@qvtm/control-plane](#32-qvtmcontrol-plane)
VerifyAuthChallengeResponse	[@qvtm/control-plane](#32-qvtmcontrol-plane)

Error Index

Error surface	Location
Common SDK error categories and codes	[Appendix A](#appendix-a-error-codes)
QvtmControlPlaneError	[Chapter 32](#32-qvtmcontrol-plane)
QvtmTrustVerifierError	[Chapter 33](#33-qvtmtrust-verifier)
QvtmRuntimeError	[Chapter 34](#34-qvtmruntime-protocol)
QvtmVaultError	[Chapter 35](#35-qvtmvault)

Figure Index

Figure	Title
1	SDK-Centered Integration Architecture
2	Public SDK Operation Sequence
3	Fifteen-Step Developer Integration Workflow
4	Session Lifecycle
5	Identity Lifecycle
6	Trust Evaluation and Application Decision
7	Enrollment Workflow
8	Authentication Workflow
9	Synchronization Workflow
10	Recovery Workflow
11	Vault Protection Workflow
12	Login Profile Workflow
13	Vault Profile Workflow
14	Secure Messaging Workflow
15	Secure Upload Workflow
16	SDK Lifecycle
17	Basic Authentication Example
18	File Protection Example
19	Secure Messaging Example
20	Synchronization Example
21	Recovery Example

Normative References

The following documents define normative behavior referenced throughout this document.

- QVTM Position Paper v1.0
- QVTM White Paper v1.0
- QVTM Protocol Specification v1.0

Whenever a conflict exists, the Protocol Specification defines protocol behavior.

Informative References

Additional SDK-specific guidance may be provided through:

- SDK documentation
- SDK API reference
- Developer Portal
- language-specific SDK documentation
- deployment guides

These resources are informative.

This document remains the canonical SDK integration reference.

Public API Reference Model

This reference distinguishes three related but non-interchangeable public surfaces:

Surface	Purpose	Authority
Language-neutral SDK contract	Defines stable semantic objects and operations shared by language bindings.	This SDK Reference
Published SDK packages	Identifies actual distributable SDK families and their supported use cases.	Published package manifests and release catalog
HTTP API surface	Identifies real published operation identifiers, methods, and paths.	Published OpenAPI descriptions

A language-neutral signature is retained only where it defines a cross-language contract. It must not be presented as copy-and-paste code. Executable examples belong to the language-specific SDK documentation and must be generated or validated against the corresponding released package.

Terminology

This document adopts the canonical terminology defined by the QVTM White Paper and the QVTM Protocol Specification.

Terms are not redefined here.

Readers SHOULD consult the normative documents whenever additional conceptual background is required.

Conformance

SDK implementations claiming conformance with this document SHOULD expose observable behavior consistent with the public contracts described herein.

Applications SHOULD rely exclusively on documented SDK behavior.

Implementation-specific behavior is outside the scope of this document.

Part I — SDK Overview

Part I introduces the public QVTM SDK model. It explains what the SDK exposes, how applications use it, which responsibilities remain outside the SDK, and how developers move from configuration to continuity-aware application decisions.

The QVTM Position Paper explains why Trust Continuity is required. The QVTM White Paper explains the architecture through which QVTM preserves and evaluates continuity. The QVTM Protocol Specification defines the externally observable behavior required for interoperable protocol implementations. This SDK Reference explains how application software uses those capabilities through public SDK contracts.

1. Introduction

1.1 Purpose

The QVTM SDK provides application-facing contracts for integrating Trust Continuity into software systems.

A developer uses the SDK to:

- create and configure a QVTM client;
- select a Deployment Profile;
- establish and manage bounded SDK sessions;
- enroll identities, devices, applications, or other supported participants;
- request authentication and continuity-aware verification;
- submit continuity-relevant application events;
- create, preserve, exchange, and verify Continuity Evidence Packages (CEPs);
- request Trust Evaluation results;
- synchronize eligible evidence;
- initiate recovery workflows;
- protect or restore assets through the Vault Profile;
- consume structured results and errors without depending on internal protocol algorithms.

The SDK does not expose Graph Puzzle internals, correlation heuristics, proprietary continuity-scoring logic, private storage layouts, or implementation-specific protocol state. It exposes stable public objects and observable operations whose behavior is consistent with the QVTM Protocol Specification.

1.2 Audience

This reference is written for developers integrating QVTM into:

- backend services;
- desktop applications;
- mobile applications;
- embedded or systems software;
- identity and access workflows;
- protected storage and recovery workflows;
- secure messaging or transfer systems;
- tenant-aware SaaS platforms;
- intermittently connected or offline deployments.

Readers are expected to understand ordinary application development, asynchronous operations, error handling, and secure secret management. Familiarity with public-key cryptography is useful but not required for using the public SDK contracts.

1.3 Relationship to the QVTM Documentation Set

The first-generation QVTM documentation has four complementary responsibilities.

Document	Primary question	SDK integration use
QVTM Position Paper	Why is Trust Continuity needed?	Provides the scientific and architectural motivation.
QVTM White Paper	How is QVTM architected?	Explains the architecture behind the public integration surface.
QVTM Protocol Specification	What behavior is required?	Defines normative protocol semantics and interoperability obligations.
QVTM SDK Reference	How do I use QVTM correctly?	Defines public SDK concepts, contracts, workflows, profiles, examples, and integration guidance.

This document does not restate the architecture, motivation, or complete protocol semantics. When a developer needs to understand why a protocol result has a particular meaning, this reference points to the appropriate companion document.

1.4 Scope

This reference defines the language-neutral public SDK model.

It documents:

- public concepts;
- public object responsibilities;
- public operation contracts;
- observable lifecycle behavior;
- profile selection and composition;
- result and error semantics;
- integration workflows;
- compatibility expectations;
- conceptual examples.

Language-specific packages may map these contracts to classes, structures, protocols, traits, functions, builders, asynchronous streams, callbacks, or equivalent constructs. Such mappings may differ in syntax but must preserve the public semantics described here.

1.5 Non-Goals

This reference does not document:

- internal Graph Puzzle algorithms;
- Trust Evaluation heuristics or scoring models;
- proprietary correlation methods;
- internal key derivation or key hierarchy design;
- private protocol-core modules;
- internal persistence schemas;
- infrastructure topology;
- private control-plane storage behavior;
- application authorization policy;
- operating-system or endpoint security controls;
- wire-level fields already defined by the Protocol Specification;
- implementation details of standardized cryptographic primitives.

The SDK may report that evidence is valid, invalid, incomplete, incompatible, continuous, conditional, indeterminate, or discontinuous. It does not disclose the proprietary internal process used to derive a result beyond the public supporting context defined by the result contract.

1.6 Public Integration Boundary

The SDK boundary separates application concerns from QVTM protocol concerns.

Application responsibility	SDK responsibility
Decide which business event is continuity-relevant.	Validate and submit the event through a public contract.
Protect application credentials and local secrets.	Use configured cryptographic and storage services without exposing internal material.
Decide whether to authorize, deny, retry, recover, or escalate.	Return a Trust Evaluation and supporting public context.
Define tenant and application policy.	Apply supplied policy context to supported protocol operations.
Maintain application sessions and user experience.	Maintain bounded SDK and protocol interaction contexts.
Handle business data.	Keep business data separate from Continuity Evidence unless a profile explicitly requires a protected reference or payload.

A successful QVTM operation does not automatically authorize an application action. The SDK supplies continuity context. The application remains responsible for its local decision.

1.7 Public SDK Families

A QVTM distribution may provide one complete SDK or several packages aligned to deployment responsibilities. The language-neutral model recognizes the following public capability families.

Capability family	Purpose
Core Client	Configuration, lifecycle, session creation, capability discovery, and common result handling.
Identity	Enrollment, identity references, device relationships, identity continuity, and recovery context.
Evidence and CEP	Continuity Evidence submission, CEP creation, verification, preservation, import, and export.
Trust Evaluation	Requests continuity evaluation and returns public result context.
Synchronization	Discovers and exchanges eligible evidence without synchronizing private implementation state.
Recovery	Restores operational capability by extending existing continuity history.
Profiles	Adapts the core contracts to Login, Vault, Secure Messaging, and Secure Upload workflows.
Administrative integration	Supports tenant-aware provisioning and operational workflows where a control-plane SDK is used.

An implementation may package these capabilities separately. Package boundaries do not change their semantics.

1.8 Conventions Used in This Reference

The document uses two distinct code-block classes:

- **Language-neutral SDK signatures** define the semantic API contract independently of TypeScript, Swift, or Rust syntax. They are interface-definition notation, not executable pseudocode.
- **Published API examples** use real package names, operation identifiers, HTTP methods, and paths taken from the supplied public SDK catalog.

JSON examples in this reference define public object representations at the SDK boundary. Endpoint-specific request and response bodies are authoritative only when they are present in the published OpenAPI source for that API. This document does not invent missing endpoint payload fields.

Public types use `PascalCase`, operations use `camelCase`, and result alternatives use explicit names. For example:

An operation shown without callback syntax may be synchronous, asynchronous, or awaitable in a language-specific SDK. Language-specific references define the exact execution model.

The terms **must**, **must not**, **should**, and **may** in this SDK Reference describe correct SDK use and public contract expectations. Normative protocol requirements remain defined by the QVTM Protocol Specification.

2. SDK Design Principles

2.1 Stable Contracts, Replaceable Implementations

QVTM SDKs expose semantic contracts rather than internal protocol structure.

A conforming language binding may use different runtime models, storage engines, cryptographic providers, or transport libraries. Applications should remain insulated from those differences when they use the public objects and operations defined by this reference.

The public surface is designed so that:

- internal protocol algorithms can evolve without forcing application rewrites;
- cryptographic algorithms can migrate without changing continuity semantics;
- connected and offline deployments can use the same conceptual operations;
- language bindings can remain idiomatic without creating incompatible meanings;
- profile-specific behavior can extend core operations without redefining them.

2.2 Evidence Before Assertion

QVTM applications do not submit an unsupported claim that continuity exists. They submit events, references, and verifiable context from which Continuity Evidence may be created or associated.

Likewise, the SDK does not return a bare boolean for Trust Continuity. It returns a structured evaluation result with a public outcome, evaluated context, relevant identifiers, conditions, and safe diagnostic information.

The expected application pattern is:

This pattern preserves the architectural separation established by the White Paper and Protocol Specification.

2.3 Evaluation Is Not Authorization

A Trust Evaluation result informs application policy. It does not replace it.

For example, the SDK may return `ContinuousWithConditions` because identity and device continuity remain supported but a local synchronization condition is outstanding. One application may permit read-only access. Another may require fresh synchronization. Both decisions can be correct because the applications have different policies.

Applications must not rewrite an evaluation result to create a preferred outcome. They may only decide what operational consequence follows from the returned result.

2.4 Fail Closed at Trust Boundaries

When an application cannot determine whether required QVTM conditions are satisfied, it should deny or defer the protected action.

Typical fail-closed conditions include:

- missing required evidence;
- expired challenges;
- incompatible protocol or profile versions;
- unknown or unacceptable device state;
- tenant-context mismatch;
- failed CEP integrity verification;
- indeterminate Trust Evaluation when policy requires continuous status;
- unavailable mandatory recovery validation;
- partial or ambiguous operation completion.

Fail closed does not mean destroying prior state. Failed operations must preserve previously valid Continuity Evidence and CEPs.

2.5 Explicit Context

Every security-relevant request should identify the context in which it is evaluated.

Depending on the operation, context may include:

- tenant identifier;
- application identifier;
- Deployment Profile;
- identity or participant reference;
- device reference;
- session reference;
- operation purpose;
- policy reference;
- expected protocol version;
- correlation or continuity scope;
- expiration or freshness constraints.

Implicit global context is discouraged because it increases the risk of cross-tenant, cross-application, or cross-profile confusion.

2.6 Immutable Public Evidence

Continuity Evidence and issued CEPs are immutable public protocol objects.

SDKs may provide immutable value types, read-only interfaces, handles, references, or verified views. Applications must not edit previously issued evidence or CEPs. When new information becomes available, the application creates a new event or operation that extends continuity history.

An application that needs to correct inaccurate application metadata does not modify the original evidence. It submits a new continuity-relevant event that identifies the correction relationship according to the selected profile.

2.7 Idempotent Integration Where Possible

Operations that may be retried should accept an idempotency or request identifier when supported by the binding or profile.

Repeated processing of the same logical request must not create duplicate Continuity Evidence. Applications should persist request identifiers across retry attempts for enrollment, evidence submission, synchronization commits, recovery requests, and protected transfer initiation.

2.8 Minimum Necessary Disclosure

QVTM integrations should provide only the information required by the selected operation and profile.

Applications should:

- reference protected data instead of embedding it in Continuity Evidence unless the profile requires a protected payload;
- avoid placing secrets in labels, metadata, diagnostics, or error messages;
- avoid storing provisioning credentials in browsers or untrusted clients;
- avoid exposing internal identifiers to users when a safer public handle exists;
- separate business payloads from evidence metadata;
- minimize participant and device information included in transferable packages.

2.9 Cryptographic Agility

Applications should configure capabilities and policy, not hard-code assumptions about one permanent algorithm.

A public SDK may expose algorithm identifiers, suite identifiers, provider capabilities, or profile requirements where applications need compatibility information. The application should not interpret those fields as internal cryptographic state.

Historical evidence may use earlier approved suites. New operations may use later suites. The SDK and Protocol Core preserve continuity across such transitions when the applicable profile and policy permit them.

2.10 Offline-First Continuity

Connectivity is not a prerequisite for local continuity preservation.

Profiles that support offline operation may allow applications to:

- open a local SDK context;
- create eligible Continuity Evidence;
- protect data;
- create or extend CEPs;
- preserve pending synchronization records;
- evaluate continuity from locally available evidence;
- synchronize later without rewriting history.

Applications must not assume that remote synchronization has completed merely because local evidence was created successfully. Local preservation and remote acceptance are separate observable results.

2.11 Observable Determinism

Semantically equivalent inputs, the same observable protocol state, and the same applicable profile constraints should produce semantically equivalent public results.

Applications must not depend on undocumented ordering, private storage details, or internal heuristics. They should depend only on the fields and result semantics documented by the public contract.

2.12 Safe Diagnostics

Public errors and diagnostics describe observable integration conditions. They do not expose proprietary internals, secret material, raw cryptographic keys, private correlation data, or sensitive storage paths.

A language-specific SDK may attach a cause for local debugging, but applications should not return that cause directly to untrusted clients. Production logging should use structured error category, public error code, operation identifier, and safe context.

2.13 Profile Composition Without Semantic Redefinition

Profiles adapt QVTM to application domains. They do not redefine core terms.

For example:

- the Login Profile specializes enrollment and authentication;
- the Vault Profile specializes protection, preservation, and recovery;
- the Secure Messaging Profile specializes participant and conversation continuity;
- the Secure Upload Profile specializes protected transfer and recipient continuity.

All profiles preserve the same meanings of Continuity Evidence, CEP, Session, Trust Evaluation, and application decision.

2.14 Developer-Controlled Policy, Protocol-Controlled Evidence

Applications may control policy inputs exposed by the SDK. They may not directly manufacture protocol outcomes.

Correct:

Incorrect:

The first example supplies policy constraints. The second attempts to rewrite a protocol result.

3. SDK Architecture

3.1 Architectural Overview

The SDK is the application-facing layer above the QVTM Protocol Core. It coordinates configuration, sessions, public objects, profiles, storage and transport adapters, and result delivery.

Figure 1. SDK-Centered Integration Architecture

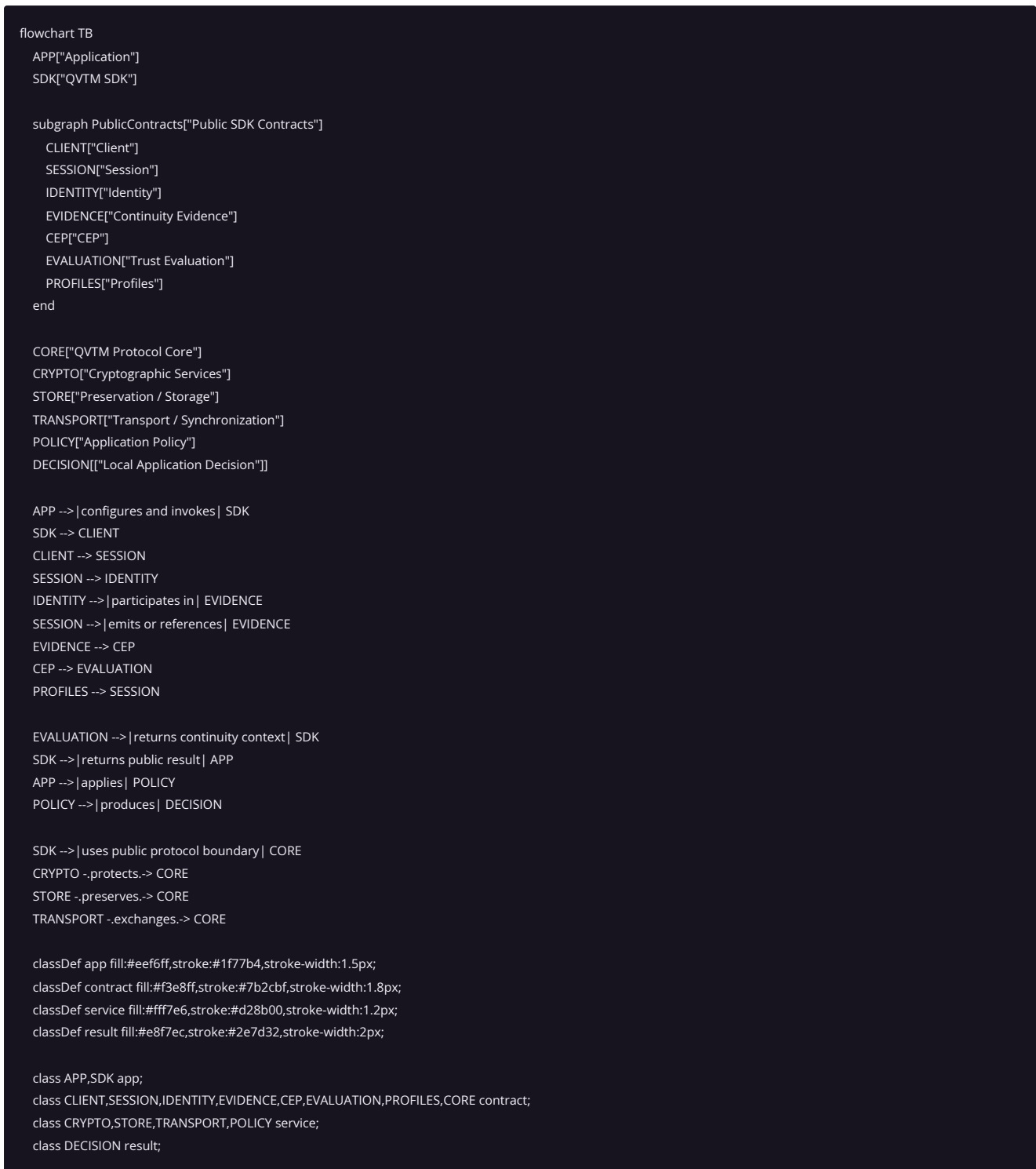


Figure 1 is a Workflow Figure. It shows the transformation and responsibility boundaries used throughout the SDK. The application invokes the SDK, the SDK uses the Protocol Core, supporting services protect and preserve protocol objects, and the application converts the returned continuity context into a local decision.

3.2 Architectural Layers

The SDK architecture contains five public-facing layers.

Layer	Responsibility
Application Integration	Business workflow, tenant context, policy, user experience, and final decisions.
SDK Facade	Client lifecycle, configuration, capability discovery, sessions, and common errors.
Public Domain Contracts	Identity, evidence, CEP, Trust Evaluation, synchronization, and recovery objects.

Layer	Responsibility
Profile Contracts	Login, Vault, Secure Messaging, and Secure Upload specialization.
Protocol Boundary	Observable operations delegated to the QVTM Protocol Core.

Supporting cryptographic, storage, transport, clock, and logging services may be supplied by the SDK, the platform, or the application. Their use remains constrained by the selected profile.

3.3 Client

`QVTMClient` is the root public SDK object.

It owns or references:

- validated configuration;
- runtime capabilities;
- selected default Deployment Profile;
- registered adapters and providers;
- session creation facilities;
- lifecycle state;
- compatibility information;
- shared local resources that are safe to reuse.

The Client does not represent an authenticated user, an application session, or a Trust State. It is a configured access point to QVTM capabilities.

Typical public responsibilities are:

Exact names may differ by language binding. Chapters 6 and 21 define the complete Client and lifecycle contracts.

3.4 Session

A `Session` is a bounded SDK execution context.

It groups related operations under explicit profile, tenant, application, participant, policy, and compatibility context. A Session may correspond to a protocol Session, but the SDK may also use a Session as a safe application-facing wrapper around one or more bounded protocol interactions.

A Session:

- is created from a Client;
- has explicit context;
- has a lifecycle;
- may perform enrollment, authentication, evidence, evaluation, synchronization, recovery, or profile-specific operations;
- emits observable results;
- may generate Continuity Evidence;
- does not authorize application access by itself;
- must be closed or allowed to expire according to the binding.

3.5 Identity

The Identity contract represents a public continuity-aware identity reference and its observable lifecycle relationships.

Identity objects may represent:

- users;
- operators;
- devices;
- applications;
- services;
- autonomous agents;
- deployment-specific participants.

An Identity is not limited to one credential. It may remain continuous across credential renewal, device replacement, recovery, or profile migration when supported by valid Continuity Evidence.

The public Identity contract exposes references, supported attributes, status, relationships, and permitted lifecycle operations. It does not expose internal identity-correlation logic.

3.6 Continuity Evidence

The SDK exposes Continuity Evidence as an immutable public object or verified reference.

Applications interact with evidence through operations such as:

- submit a continuity-relevant event;
- inspect safe public metadata;
- reference evidence in a later operation;
- verify evidence eligibility or integrity;
- include evidence in a CEP;
- preserve, export, or synchronize evidence through supported profile contracts.

Applications must not mutate issued evidence.

3.7 CEP

The SDK exposes CEP operations at a semantic level.

Common public operations include:

- create a CEP from eligible evidence;
- import a CEP;
- export a CEP;
- verify CEP integrity and compatibility;
- inspect safe package metadata;
- preserve a CEP;
- mark a CEP eligible for synchronization;
- reference a CEP in Trust Evaluation or recovery.

The SDK may return a `CEPReference`, `VerifiedCEP`, stream, byte container, file handle, or platform-native representation. The representation must preserve the same package semantics.

3.8 Trust Evaluation

The Trust Evaluation contract accepts an explicit evaluation context and returns a structured outcome.

The public outcome model contains at least the following semantic classes:

Outcome	Meaning
<code>Continuous</code>	Available validated evidence supports continuity for the evaluated context.
<code>ContinuousWithConditions</code>	Continuity is supported subject to stated public conditions.
<code>Indeterminate</code>	Available evidence is insufficient to determine continuity.
<code>Discontinuous</code>	Available evidence indicates that continuity is not preserved for the evaluated context.

A result may include:

- evaluation identifier;
- evaluated context;
- outcome;
- public conditions;
- referenced evidence or CEP identifiers;
- freshness information;
- compatibility information;
- safe warnings;
- recommended next operations such as synchronize, refresh, recover, or deny.

A result does not expose private correlation graphs, scoring weights, internal candidate paths, or proprietary heuristics.

3.9 Profiles

A Deployment Profile selects domain-specific behavior while preserving core QVTM semantics.

The SDK Reference defines four primary public profiles.

Profile	Primary purpose
Login Profile	Passwordless, device-aware authentication and identity continuity.
Vault Profile	Offline asset protection, preservation, opening, modification, and recovery.
Secure Messaging Profile	Participant, device, conversation, and message-context continuity.
Secure Upload Profile	Protected transfer, recipient continuity, transfer evidence, and optional preservation.

A language-specific SDK may expose profiles through typed clients, builders, namespaces, capability flags, or session options.

3.10 Providers and Adapters

SDK implementations may expose provider or adapter interfaces for platform integration.

Common provider categories include:

Provider	Purpose
Cryptographic Provider	Supplies profile-approved signing, verification, key establishment, encryption, and secure random services.
Secure Storage Provider	Protects local secret material and sensitive SDK state.
CEP Preservation Provider	Stores and retrieves immutable CEPs.
Transport Provider	Exchanges protocol messages or synchronization material.
Clock Provider	Supplies trusted or policy-qualified time context where required.
Logger / Telemetry Provider	Receives safe structured diagnostics and operational metrics.
User Interaction Provider	Connects profile operations to approval, biometric, device, or recovery UX where supported.

Provider contracts are implementation-specific unless a language SDK reference declares them public. Applications should not replace a built-in provider unless they can satisfy the relevant profile and security requirements.

3.11 Connected and Embedded Modes

The same public SDK model may be realized in different deployment modes.

Mode	Description
Local / Embedded	Protocol Core and required services execute within the application process or trusted local component.
Remote Service	The SDK calls an authorized QVTM service over a protected transport.
Hybrid	Some operations execute locally while governance, synchronization, or verification uses remote services.
Offline	Eligible operations execute from local context and preserved evidence without current network connectivity.

Applications should use capability discovery rather than infer mode from package name or platform.

3.12 Tenant Boundary

Tenant context is an explicit application and governance boundary.

An SDK request that is scoped to a tenant must carry or inherit a validated `TenantReference`. Applications must not reuse identity, device, challenge, session, or policy objects across tenants unless a specific public operation explicitly supports a cross-tenant relationship.

A tenant mismatch must produce a defined error and must not generate a partial continuity update.

Commercial customer ownership and tenant operation are separate concepts. A customer may own or manage one or more tenants, while a tenant remains the operational boundary used by QVTM integrations.

3.13 SDK Operation Path

Figure 2. Public SDK Operation Sequence

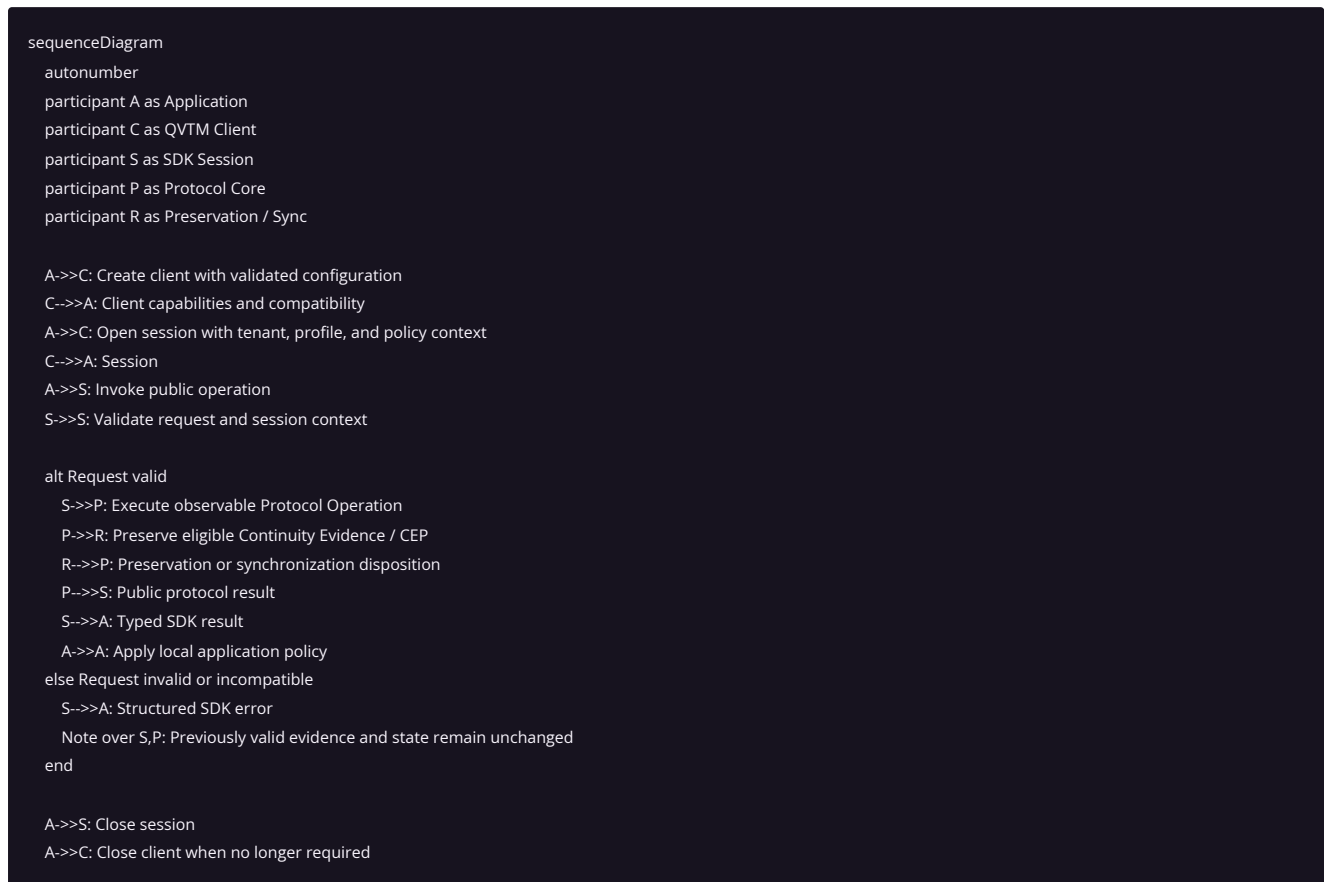


Figure 2 is a Sequence Figure. It shows the participant interaction for a generic SDK operation. Validation occurs before protocol execution. Persistence or synchronization disposition is observable when relevant. The application applies policy only after it receives the typed result.

3.14 State Ownership

The SDK distinguishes several kinds of state.

State category	Owner	Examples
Application state	Application	UI state, business transaction, local access session.
SDK lifecycle state	SDK	Client open/closed, Session active/closing/closed.
Protocol State	Protocol Core	Initialized, Enrolled, Operational, Synchronizing, Recovering, Suspended, Archived.
Preserved continuity information	Preservation layer	Continuity Evidence, CEPs, synchronization disposition.
Application policy state	Application or policy service	Required outcome, freshness limits, permitted conditions.

Applications must not infer Protocol State solely from an application screen or SDK object existence. They should use the relevant public status or result operation.

3.15 SDK Security Boundary

The SDK cannot compensate for an untrusted host that exposes secrets or rewrites application decisions.

The application is responsible for:

- platform hardening;
- secure software development;

- secret storage configuration;
- provisioning credential protection;
- dependency management;
- authorization enforcement;
- safe logging;
- user-interface integrity;
- secure update and release practices.

The SDK is responsible for enforcing public request validation, immutable evidence handling, protocol operation boundaries, compatibility checks, structured errors, and failure containment within its defined boundary.

4. SDK Concepts

4.1 Configuration

`QVTMConfiguration` defines how a Client is created.

A configuration commonly contains:

Field group	Purpose
Runtime mode	Local, remote, hybrid, or profile-defined mode.
Endpoint configuration	Authorized service endpoints when remote capabilities are used.
Credentials	Backend or service credentials supplied through secure references.
Default profile	Default Deployment Profile for sessions that do not override it.
Tenant context	Optional default tenant reference for tenant-scoped clients.
Providers	Cryptographic, storage, transport, clock, logging, or interaction providers.
Compatibility policy	Supported protocol versions, SDK versions, and mandatory extensions.
Operational policy	Timeouts, retry behavior, synchronization policy, and safe diagnostics.

Configuration must be validated before the Client becomes usable. Invalid configuration must fail without creating an operational Client.

Sensitive values should be supplied through secret references, secure storage handles, environment-backed configuration, or platform credential facilities. They should not be embedded in source code or serialized into ordinary application preferences.

4.2 Capability Discovery

A Client may expose `Capabilities` describing observable support.

Capabilities may include:

- supported profiles;
- supported protocol versions;
- supported SDK feature set;
- available offline operations;
- available synchronization mechanisms;
- supported canonical encodings;
- approved cryptographic suite identifiers;
- recovery support;
- local or remote execution availability;
- optional extension support.

Capability discovery is informational until an operation is negotiated or opened. Applications must still handle incompatibility errors because capabilities may change between discovery and execution.

4.3 References and Handles

The SDK uses public references to identify protocol objects without exposing internal storage.

Representative references include:

A reference:

- identifies one public object within an applicable context;
- may be opaque;
- may contain a version or type discriminator;
- must not be parsed unless the language-specific SDK explicitly documents its structure;
- must not be reused outside its valid tenant, profile, or lifecycle scope.

4.4 Requests

Every public operation accepts a typed request or an equivalent set of validated parameters.

A request should contain:

- explicit operation purpose;
- applicable references;
- profile context;
- request or idempotency identifier when supported;
- policy constraints;
- expected compatibility context;
- optional application metadata allowed by the profile.

Requests are input objects. The SDK may copy, validate, normalize, or reject them. Applications must not assume that mutating a request after invocation changes an in-flight operation.

4.5 Results

SDK operations return typed results.

A result typically contains:

Result element	Purpose
Operation identifier	Correlates logs and subsequent operations.
Status	Indicates completed, pending, rejected, failed, or profile-specific disposition.
Public output	Returns references, verified objects, evaluation context, or protected data.
Continuity effect	Identifies whether Continuity Evidence was generated, referenced, or unchanged.
Modified objects	Identifies newly created or transitioned public objects.
Warnings or conditions	Communicates safe non-fatal information.
Compatibility context	Records the selected protocol/profile compatibility where relevant.

A completed SDK call may return a pending protocol workflow. For example, creation of an authentication challenge may complete successfully while the challenge remains pending approval.

4.6 Operation Status

The common operation status model includes the following semantic states.

Status	Meaning
Completed	The requested bounded operation completed successfully.
Pending	The operation was accepted but requires another observable event or participant action.
Rejected	The operation was validly processed but not accepted under protocol or policy conditions.

Status	Meaning
Failed	The operation could not complete because of an error.
Cancelled	The operation was cancelled before successful completion.
Expired	The operation or its transient material exceeded its permitted lifetime.

Not every binding or operation exposes every status. Profile chapters define applicable statuses.

4.7 Continuity Effect

Operations identify their observable effect on continuity.

Continuity effect	Meaning
None	The result is not continuity-relevant and generated no evidence.
Referenced	Existing evidence was referenced without modification.
Extended	New Continuity Evidence extended an existing continuity history.
Established	Enrollment generated the initial continuity context.
PendingCommit	Evidence was produced but final profile-defined preservation or acceptance is pending.
RejectedPreserved	The operation failed or was rejected while prior continuity remained preserved; failure evidence may exist.

The SDK does not claim `Extended` until the applicable commit boundary has been satisfied.

4.8 Sessions and Context

`SessionOptions` define the bounded context for subsequent operations.

Typical fields include:

A Session may expose its public state:

SDK Session state	Meaning
Open	The Session accepts permitted operations.
Busy	A non-concurrent operation is executing.
PendingExternalAction	The Session awaits an external approval or event.
Closing	No new operations are accepted; cleanup is in progress.
Closed	The Session no longer accepts operations.
Expired	The Session exceeded its valid lifetime.
Failed	A non-recoverable Session-local failure prevents further use.

These are SDK lifecycle states, not Protocol States.

4.9 Events

An `ApplicationEvent` is a structured request to record or process a continuity-relevant application observation.

The event contract commonly includes:

- event identifier;
- event type;
- occurrence context;
- source reference;
- tenant and application context;
- profile context;

- related object references;
- allowed metadata;
- optional protected payload reference;
- ordering or causality references where supported.

Submitting an event does not guarantee that new Continuity Evidence will be created. The SDK validates whether the event is eligible and continuity-relevant under the selected profile.

4.10 Continuity Evidence

A public `ContinuityEvidence` view exposes only safe, contractually stable information.

Typical fields are:

The exact time representation may differ. Applications should not treat wall-clock time as the only ordering mechanism.

4.11 CEP Objects

A public CEP object may move through several observable SDK forms.

Form	Meaning
<code>CEPReference</code>	Opaque identifier for a locally or remotely addressable CEP.
<code>CEPInput</code>	Importable serialized, streamed, or file-based representation.
<code>VerifiedCEP</code>	CEP whose syntax, integrity, compatibility, and required semantics have been validated.
<code>CEPExport</code>	Profile-approved representation for storage or transfer.
<code>CEPDisposition</code>	Result describing accepted, duplicate, conflicting, rejected, or pending treatment.

A `VerifiedCEP` does not by itself mean that Trust Continuity is continuous. It means the package passed the applicable package verification boundary.

4.12 Trust Evaluation Request

A `TrustEvaluationRequest` identifies what continuity question the application is asking.

It should specify:

- evaluated subject or relationship;
- tenant and application context;
- selected profile;
- purpose;
- evidence and CEP scope;
- policy constraints;
- freshness requirements;
- required outcome or acceptable conditions, if supplied as policy;
- optional synchronization or recovery preference.

The SDK may reject an ambiguous request rather than infer a security-critical scope.

4.13 Trust Evaluation Result

A `TrustEvaluationResult` contains public decision support.

Representative structure:

The result must be treated as immutable.

4.14 Policy

Policy constrains how an operation or evaluation is performed and how the application interprets results.

The SDK may accept:

- a `PolicyReference` resolved by an authorized service;

- a typed local `EvaluationPolicy`;
- profile-specific policy options;
- a composition of local and tenant policy where supported.

Policy inputs must not redefine core protocol semantics. They may require stronger evidence, shorter freshness windows, specific profiles, additional approval, or stricter handling of indeterminate outcomes.

4.15 Challenges

Profiles such as Login and Recovery may use a `Challenge` object.

A Challenge is transient protocol material with:

- a challenge reference;
- purpose;
- participant and tenant context;
- creation and expiration context;
- current public status;
- approved transport or presentation forms;
- optional response reference after completion.

Challenge payloads should be treated as short-lived and sensitive. A challenge must not be accepted solely because it can be decoded or displayed.

4.16 Synchronization

Synchronization exchanges eligible evidence and CEPs. It does not replicate private implementation state.

Public synchronization concepts include:

The result distinguishes accepted, duplicate, conflicting, missing, rejected, and deferred material. Conflict visibility does not mean the SDK has made an application trust decision.

4.17 Recovery

Recovery restores operational capability while extending existing continuity.

Public recovery concepts include:

A successful Recovery operation returns a new or restored operational context and references newly generated recovery Continuity Evidence. It does not rewrite the prior history.

4.18 Vault

The Vault Profile introduces public protected-storage concepts.

Representative objects include:

A Vault operation may create, open, add, replace, remove, export, synchronize, archive, or recover protected assets depending on implementation maturity and profile support. Every modifying operation must report which public objects changed and which Continuity Evidence was generated.

4.19 Secure Messaging

The Secure Messaging Profile introduces:

QVTM does not replace transport routing or delivery semantics. It supplies continuity-aware participant, device, cryptographic transition, and message-context contracts.

4.20 Secure Upload

The Secure Upload Profile introduces:

The profile protects transfer and recipient continuity without granting access to the sender's Vault. The transferred asset and the sender's preservation boundary remain separate.

4.21 Errors

All public SDK errors map to observable categories.

Core categories include:

- configuration;
- validation;
- invalid state;
- authentication;
- authorization at the SDK service boundary;
- compatibility;
- integrity;
- evidence;
- CEP;
- synchronization;
- recovery;
- transport;
- timeout and expiration;
- storage;
- cryptographic provider;
- internal failure without internal disclosure.

Chapter 22 defines the complete error model and Appendix A defines public error codes.

4.22 Warnings

Warnings communicate non-fatal conditions such as:

- optional extension ignored;
- evidence freshness approaching a policy boundary;
- remote synchronization deferred;
- deprecated SDK feature in use;
- historical cryptographic suite accepted for verification but not for new operations;
- partial optional capability unavailable.

Applications should log and monitor warnings. They should not convert warnings into success assumptions.

4.23 Public JSON Representations

QVTM uses three explicitly different JSON classes. They must not be confused.

JSON class	Authority	Intended use
Contractual JSON	Published OpenAPI or generated binding	Exact HTTP request and response contracts.
Canonical SDK object representation	This SDK Reference	Cross-binding meaning for public results and references.
Illustrative JSON	Marked examples only	Explains composition; not a wire contract.

A JSON block must identify its class in the surrounding text. Implementations must not present illustrative JSON as an OpenAPI contract.

4.23.1 Canonical Reference

Canonical SDK object representation

```
{
  "type": "identity",
  "value": "idn_01JEXAMPLE",
  "tenantReference": "ten_01JEXAMPLE"
}
```

`type` identifies the public reference category. `value` is opaque and must not be parsed. `tenantReference` is present when the reference is tenant-scoped.

4.23.2 Canonical Operation Result

Canonical SDK object representation

```
{
  "operationReference": "op_01JEXAMPLE",
  "status": "completed",
  "continuityEffect": "extended",
  "evidenceReferences": ["ev_01JEXAMPLE"],
  "cepReferences": ["cep_01JEXAMPLE"],
  "warnings": []
}
```

4.23.3 Canonical Trust Evaluation Result

Canonical SDK object representation

```
{
  "evaluationReference": "eval_01JEXAMPLE",
  "outcome": "continuous-with-conditions",
  "evaluatedContext": {
    "tenantReference": "ten_01JEXAMPLE",
    "identityReference": "idn_01JEXAMPLE",
    "purpose": "application-login"
  },
  "conditions": [
    {
      "code": "synchronization-recommended",
      "message": "Eligible continuity information has not yet been synchronized."
    }
  ],
  "evidenceReferences": ["ev_01JEXAMPLE"],
  "cepReferences": ["cep_01JEXAMPLE"],
  "recommendations": ["synchronize"],
  "continuityEffect": "referenced"
}
```

4.23.4 Canonical SDK Error

Canonical SDK object representation

```
{
  "category": "invalid-state",
  "code": "SDK_SESSION_CLOSED",
  "message": "The session no longer accepts operations.",
  "operationReference": "op_01JEXAMPLE",
  "recoverable": false,
  "retryable": false,
  "safeDetails": {}
}
```

The public error object must not expose secret material, private storage paths, raw cryptographic values, or proprietary correlation data.

4.23.5 Canonical Capabilities Result

Canonical SDK object representation

```
{
  "sdkVersion": "1.0.0",
  "contractVersion": "1",
  "protocolVersions": ["1.0"],
  "profiles": ["login", "vault", "secure-messaging", "secure-upload"],
  "operations": ["enrollment", "authentication", "synchronization", "recovery"],
  "offlineCapabilities": ["vault-open", "vault-protect", "cep-preservation"],
  "optionalCapabilities": []
}
```

4.23.6 Canonical CEP Disposition

Canonical SDK object representation

```
{
  "cepReference": "cep_01JEXAMPLE",
  "disposition": "accepted",
  "integrityStatus": "valid",
  "compatibilityStatus": "compatible",
  "continuityEffect": "extended",
  "conflictReference": null
}
```

Allowed disposition classes are `accepted`, `duplicate`, `conflicting`, `rejected`, and `deferred`.

5. Developer Workflow

5.1 Overview

The recommended integration workflow begins with a narrow protected use case and expands only after the application can correctly handle evaluation results, errors, evidence preservation, and lifecycle transitions.

Figure 3. Fifteen-Step Developer Integration Workflow

```
flowchart TD
    S01["1. Choose protected use case"] --> S02["2. Select Deployment Profile"]
    S02 --> S03["3. Define trust and tenant boundaries"]
    S03 --> S04["4. Place credentials correctly"]
    S04 --> S05["5. Configure the Client"]
    S05 --> S06["6. Validate capabilities"]
    S06 --> S07["7. Open a bounded Session"]
    S07 --> S08["8. Execute the primary operation"]
    S08 --> S09["9. Handle pending workflows"]
    S09 --> S10["10. Evaluate Trust"]
    S10 --> S11["11. Apply local policy"]
    S11 --> S12["12. Preserve references and outcomes"]
    S12 --> S13["13. Close the Session"]
    S13 --> S14["14. Test failure paths"]
    S14 --> S15["15. Monitor without exposing secrets"]
    S15 -. next continuity-relevant event .-> S08

    classDef input fill:#eef6ff,stroke:#1f77b4,stroke-width:1.5px;
    classDef operation fill:#f3e8ff,stroke:#7b2cbf,stroke-width:1.8px;
    classDef evidence fill:#fff7e6,stroke:#d28b00,stroke-width:1.2px;
    classDef result fill:#e8f7ec,stroke:#2e7d32,stroke-width:2px;

    class S01,S02,S03,S04,S05,S06 input;
    class S07,S08,S09,S10 operation;
    class S11,S12,S13 evidence;
    class S14,S15 result;
```

Figure 3 is a Workflow Figure. Its numbered nodes map one-to-one to Sections 5.2 through 5.16; no textual step is omitted.

5.2 Step 1 — Choose One Protected Use Case

Begin with one explicit application action.

Examples:

- grant a local application session after passwordless approval;
- protect one file in a personal Vault;
- verify conversation continuity before sending a sensitive message;
- upload one protected asset to a continuity-verified recipient;
- recover a Vault after device replacement;
- synchronize locally preserved CEPs after reconnection.

Avoid integrating every QVTM capability at once. A narrow use case makes trust boundaries, failure handling, and evidence expectations testable.

5.3 Step 2 — Select a Deployment Profile

Select the profile that owns the primary workflow.

Use case	Primary profile
Passwordless sign-in or approval	Login Profile
Local or offline protected storage	Vault Profile
Conversation or participant continuity	Secure Messaging Profile
Protected recipient-oriented transfer	Secure Upload Profile

Profiles may compose. For example, Secure Upload may use Vault preservation for the sender and Login identity continuity for recipient approval. Composition must remain explicit.

5.4 Step 3 — Define Boundaries

Before writing SDK calls, define:

- tenant boundary;
- customer ownership context, if applicable;
- application boundary;
- identity and device roles;
- trusted backend boundary;
- client and mobile boundary;
- local decision point;
- secret-storage boundary;
- evidence-preservation boundary;
- synchronization boundary;
- recovery authority or validator model;
- acceptable Trust Evaluation outcomes.

The most common integration failures are boundary errors rather than syntax errors.

5.5 Step 4 — Place Credentials Correctly

Backend provisioning or administrative credentials belong only in trusted backend environments.

They must not be embedded in:

- browser JavaScript;
- mobile application bundles;
- desktop application resources accessible to ordinary users;
- QR codes;
- challenge payloads;
- public configuration files;
- source repositories;

- logs or crash reports.

User-facing applications should receive only short-lived, purpose-bound material such as challenges, invitations, transfer offers, or profile-approved tokens.

5.6 Step 5 — Configure the Client

Conceptual example:

Client creation must validate configuration. Applications should treat creation failure as a startup or dependency-health failure, not as an authentication denial.

5.7 Step 6 — Validate Capabilities

Before starting a protected workflow, verify that the required profile and capabilities are available.

Capability checks improve diagnostics but do not replace operation-level error handling.

5.8 Step 7 — Open a Bounded Session

Do not keep a Session open indefinitely. Use a lifetime appropriate to the workflow. A long-lived application may reuse a Client while creating multiple bounded Sessions.

5.9 Step 8 — Execute the Primary Operation

A Login flow may create an authentication challenge:

A Vault flow may protect an asset:

Each operation chapter defines inputs, outputs, errors, modified objects, possible Continuity Evidence, and protocol relationships.

5.10 Step 9 — Handle Pending Workflows

Some operations involve another participant or device.

A pending result may include:

- challenge reference;
- expiration;
- presentation payload;
- polling or subscription capability;
- allowed cancellation operation;
- current status.

The application must not treat `Pending` as success.

Polling, callback, event stream, or push mechanisms are binding-specific.

5.11 Step 10 — Evaluate Trust

After the required operation completes, request or consume Trust Evaluation.

The application should branch on the explicit outcome.

A verified challenge alone is not sufficient when policy requires device, tenant, evidence, and freshness validation.

5.12 Step 11 — Apply Local Policy

The application owns the final decision.

Examples of local consequences are:

- create an ephemeral application session;
- permit read-only access;
- require a second approval;
- request synchronization;
- initiate recovery;
- deny access;
- quarantine a transfer;

- notify an operator;
- preserve incident evidence.

The application should record the evaluation reference associated with the local decision without copying sensitive internal details.

5.13 Step 12 — Preserve References and Outcomes

Persist only what the application needs for later operations and audit.

Commonly preserved public values include:

- operation reference;
- evaluation reference;
- identity reference;
- device reference;
- CEP reference;
- challenge or transfer reference;
- application decision and policy version;
- safe error code;
- continuity effect.

Do not duplicate full CEPs or evidence in ordinary application databases when the configured preservation provider already owns them, unless the deployment architecture explicitly requires controlled export.

5.14 Step 13 — Close the Session

Closing the SDK Session does not delete Continuity Evidence, CEPs, or protocol history. It releases the bounded execution context and transient resources.

5.15 Step 14 — Test Failure Paths

A production integration must test at least:

- invalid configuration;
- missing credentials;
- tenant mismatch;
- unsupported profile;
- protocol-version incompatibility;
- expired challenge;
- duplicate request;
- invalid or tampered CEP;
- missing evidence;
- indeterminate evaluation;
- discontinuous evaluation;
- synchronization interruption;
- storage failure;
- recovery rejection;
- cancellation;
- application crash after local evidence creation but before remote synchronization;
- retry after uncertain transport outcome.

Tests must confirm that previously valid evidence remains preserved after failure.

5.16 Step 15 — Monitor Without Exposing Secrets

Recommended production telemetry includes:

- operation counts and latency;
- public error-code counts;
- evaluation outcome counts;
- pending and expired challenge counts;
- synchronization backlog;

- CEP verification failures;
- compatibility warnings;
- recovery attempts and dispositions;
- profile capability health.

Telemetry must not include private keys, provisioning credentials, protected asset contents, raw challenge secrets, or unrestricted CEP payloads.

5.17 Development-to-Production Progression

Stage	Required focus
Local development	Correct request construction, typed result handling, and deterministic tests.
Integration	Tenant boundaries, remote dependencies, credential placement, and profile workflows.
Staging	Realistic expiration, synchronization, recovery, and failure injection.
Production readiness	Hardening, audit, monitoring, compatibility, secret rotation, and incident procedures.
Production operation	Version management, evidence preservation, capacity, recovery drills, and migration planning.

5.18 Minimal Login Integration

The smallest correct Login integration contains:

1. a trusted backend with required QVTM service credentials;
2. explicit tenant and application context;
3. identity and optional device references;
4. a short-lived authentication challenge;
5. device or participant approval through an approved flow;
6. challenge completion handling;
7. Trust Evaluation;
8. local application policy;
9. an ephemeral application session created only after accepted evaluation;
10. structured failure and expiration handling.

5.19 Minimal Vault Integration

The smallest correct Vault integration contains:

1. a configured Vault Profile;
2. secure local secret storage;
3. CEP preservation;
4. a Vault reference and policy;
5. a protected asset source;
6. a protection result identifying modified objects and continuity effect;
7. a safe open operation;
8. remove and update operations handled as explicit continuity-relevant changes where supported;
9. recovery planning;
10. offline and later synchronization behavior where used.

5.20 Minimal Secure Messaging Integration

The smallest correct Secure Messaging integration contains:

1. explicit participant and conversation references;
2. participant and device continuity evaluation;
3. protected message creation through the profile;
4. transport handled by the application or messaging service;
5. received message verification;
6. clear treatment of participant change, device replacement, and cryptographic transition;

7. local policy for indeterminate or discontinuous conversation state.

5.21 Minimal Secure Upload Integration

The smallest correct Secure Upload integration contains:

1. sender and recipient context;
2. recipient continuity evaluation;
3. protected transfer creation;
4. transfer reference and expiration;
5. application-managed upload transport or profile-approved transport integration;
6. recipient verification and acceptance;
7. transfer evidence where required;
8. no implicit access to the sender's Vault;
9. failure and cancellation handling;
10. optional preservation or recovery of the source asset through the Vault Profile.

5.22 Integration Completion Criteria

An integration is not complete merely because the success path works.

It is ready for profile-specific review when:

- public references are scoped correctly;
- credentials remain inside their trusted boundary;
- all typed outcomes are handled;
- application authorization remains separate from Trust Evaluation;
- Continuity Evidence and CEPs are never mutated;
- retries are idempotent where supported;
- failure containment is tested;
- compatibility is validated;
- logs and diagnostics are safe;
- synchronization and recovery behavior are defined;
- SDK and protocol lifecycle cleanup is correct;
- version upgrade and rollback procedures exist.

5.23 Integration Validation

Before an application is considered production-ready, the integration SHOULD be validated against the observable behavior defined by the QVTM Protocol Specification and the selected Deployment Profile.

Integration validation focuses on the correct use of the SDK rather than on protocol conformance. The SDK provides stable integration contracts; protocol interoperability remains defined by the Protocol Specification.

The following aspects SHOULD be verified during integration validation:

- SDK initialization succeeds.
- Required identities are available.
- Session establishment follows the expected lifecycle.
- Continuity-relevant operations complete successfully.
- CEP generation, preservation, and synchronization behave as expected.
- Trust Evaluation results are correctly consumed.
- Error conditions are handled without violating application consistency.
- Recovery procedures can be executed successfully when applicable.

Applications SHOULD validate observable behavior rather than implementation-specific internal state.

5.24 Integration Responsibilities

Successful integrations maintain a clear separation of responsibilities between the application, the SDK, and the underlying protocol.

Component	Responsibility
Application	Business logic, authorization decisions, user experience, and local policy
SDK	Public integration APIs, lifecycle management, and protocol orchestration
Protocol Core	Continuity semantics, CEP lifecycle, and Protocol Operations
Deployment Profile	Domain-specific behavior and operational constraints

Applications **MUST NOT** duplicate responsibilities already provided by the SDK.

Likewise, SDK consumers **SHOULD** avoid implementing protocol behavior directly.

5.25 Integration Boundaries

The SDK intentionally exposes only the public integration surface required by applications.

The following responsibilities remain internal to QVTM implementations:

- Graph Puzzle execution.
- Internal Trust Evaluation algorithms.
- Continuity correlation mechanisms.
- Internal optimization strategies.
- Proprietary storage models.
- Internal protocol scheduling.

Applications interact exclusively through documented SDK contracts.

This separation allows QVTM implementations to evolve internally without affecting application compatibility.

5.26 Summary

The developer workflow presented in this chapter defines the canonical integration model for QVTM SDKs.

A typical integration consists of five stages:

1. Initialize the SDK.
2. Create or obtain a Client.
3. Open a Session.
4. Invoke one or more public SDK operations.
5. Consume the observable results and apply application-specific policy.

Throughout this process, the SDK manages protocol interactions while preserving the abstraction boundary between application code and the QVTM Protocol Core.

The following chapters introduce the public SDK object model, beginning with the Client, the primary entry point for all SDK operations.

Part II — Core SDK

The Core SDK defines the public object model exposed by every QVTM SDK implementation. It provides a stable abstraction over the underlying protocol while preserving implementation independence.

The public object model is centered around six primary concepts:

- Client
- Session
- Identity
- Continuity Evidence
- Continuity Evidence Package (CEP)
- Trust Evaluation

These objects cooperate to allow applications to interact with QVTM without requiring knowledge of the underlying protocol implementation.

6. Client

6.1 Overview

The Client is the primary entry point for every QVTM SDK.

Applications create or obtain a Client before performing any protocol operation.

The Client coordinates SDK initialization, session creation, identity selection, protocol execution, and access to profile-specific capabilities.

Applications SHOULD maintain a Client for the duration appropriate to their execution model.

The Client does not expose protocol internals.

6.2 Responsibilities

The Client is responsible for:

- initializing SDK services;
- managing SDK configuration;
- creating Sessions;
- selecting active identities;
- exposing profile capabilities;
- coordinating protocol operations;
- providing access to public SDK services.

The Client is not responsible for application authorization, business logic, or user interface management.

6.3 Published Surface Mapping

The language-neutral Client surface contains the following semantic operations:

Individual SDK implementations MAY adapt naming conventions to match their implementation language.

The observable behavior of these operations SHOULD remain semantically equivalent.

6.4 Lifecycle

A Client has three observable lifecycle conditions: `Created`, `Operational`, and `Closed`. Creation validates configuration; the operational condition permits Session creation and capability discovery; closure rejects new work and releases runtime resources. Closing a Client does not modify previously generated Continuity Evidence or CEPs.

6.5 Configuration

A Client is initialized using an SDK-specific configuration object.

Typical configuration information includes:

- selected Deployment Profile;
- protocol version preferences;
- storage configuration;
- synchronization configuration;
- logging preferences;
- cryptographic provider selection;
- runtime options.

SDK implementations MAY expose additional configuration parameters.

Unknown configuration values SHOULD be ignored when compatible behavior can be preserved.

6.6 Session Management

The Client is responsible for creating Session objects.

Each Session represents a bounded execution context for one or more protocol operations.

Applications SHOULD avoid manipulating Session state directly.

Instead, Sessions SHOULD be created and managed through the Client.

6.7 Identity Management

The Client provides access to the active Identity used during protocol operations.

Identity selection mechanisms depend on the selected Deployment Profile.

Examples include:

- Login Profile
- Vault Profile
- Secure Messaging Profile
- Secure Upload Profile

The Client does not define how identities are authenticated.

Authentication procedures are described in Chapter 13.

6.8 Trust Evaluation

Applications MAY request Trust Evaluation through the Client.

The SDK exposes observable Trust Evaluation results without exposing internal evaluation mechanisms.

Applications consume Trust Evaluation results as protocol outputs.

Applications remain responsible for making authorization decisions.

6.9 Error Model

Client operations MAY produce errors including:

Error Category	Typical Cause
Initialization Error	SDK could not be initialized
Configuration Error	Invalid configuration
Session Error	Session creation failed
Identity Error	No usable identity available
Profile Error	Unsupported Deployment Profile
Runtime Error	SDK runtime failure

Detailed error semantics are defined in Chapter 22.

6.10 Observable Guarantees

A successfully initialized Client guarantees that:

- SDK services are available;
- Sessions can be created;
- supported Deployment Profiles are accessible;
- observable protocol operations can be executed;
- protocol compatibility requirements have been satisfied.

The Client does not guarantee successful protocol execution.

Individual operations remain subject to protocol validation and runtime conditions.

6.11 Best Practices

Applications SHOULD:

- create one long-lived Client whenever practical;
- reuse the Client across multiple Sessions;
- initialize the Client before accessing SDK services;
- gracefully shut down the Client during application termination;

- avoid exposing the Client directly to untrusted components.

Following these recommendations simplifies lifecycle management while reducing unnecessary SDK initialization overhead.

7. Session

7.1 Overview

A **Session** represents a bounded execution context for one or more QVTM protocol operations.

Every protocol interaction occurs within a Session. Sessions provide the observable lifecycle through which applications invoke SDK operations while isolating transient execution state from long-term continuity information.

A Session is temporary.

Continuity Evidence generated during a Session may outlive the Session itself.

7.2 Responsibilities

A Session is responsible for:

- providing an execution context for protocol operations;
- maintaining operation-specific runtime state;
- coordinating protocol requests;
- exposing observable operation results;
- reporting execution errors.

A Session is not responsible for:

- identity persistence;
- authorization decisions;
- long-term storage;
- Continuity Evidence preservation;
- Trust Evaluation algorithms.

7.3 Published Surface Mapping

The language-neutral Session surface contains the following semantic operations:

```
Session

execute()

close()

state()

identity()

profile()

lastResult()
```

SDK implementations MAY adapt naming conventions according to language-specific conventions.

The observable behavior SHOULD remain semantically equivalent.

7.4 Lifecycle

Figure 4. Session Lifecycle

```
stateDiagram-v2
    [*] --> Open
    Open --> Busy: operation starts
    Busy --> Open: operation completes
    Busy --> PendingExternalAction: external action required
    PendingExternalAction --> Busy: action received
    Open --> Closing: close requested
    Busy --> Closing: cancel and close
    PendingExternalAction --> Expired: lifetime exceeded
    Closing --> Closed
    Open --> Expired: lifetime exceeded
    Open --> Failed: non-recoverable session error
    Busy --> Failed: non-recoverable session error
    Closed --> [*]
    Expired --> [*]
    Failed --> [*]
```

Figure 4 is a Lifecycle Figure because it shows the temporal evolution of one Session object. Once a Session reaches `Closed`, `Expired`, or `Failed`, it must not be reused.

7.5 Session State

The observable Session state reflects the current execution context.

Typical states include:

State	Description
Created	Session object exists but is not yet active.
Open	Operations may be executed.
Busy	One or more operations are currently executing.
Completed	Execution finished successfully.
Closed	Session has been terminated.

SDK implementations MAY expose additional implementation-specific states.

Applications SHOULD rely only on documented observable states.

7.6 Executing Operations

A Session executes protocol operations exposed by the SDK.

Examples include:

- Enrollment
- Authentication
- Synchronization
- Recovery
- Vault Operations
- Trust Evaluation

Each operation executes independently while sharing the same Session context.

Operation ordering depends on protocol requirements and the selected Deployment Profile.

7.7 Observable Results

Each operation returns an observable result.

Depending on the operation, results may include:

- operation status;

- generated objects;
- Continuity Evidence references;
- CEP references;
- Trust Evaluation results;
- profile-specific data.

The Session exposes only public SDK objects.

Internal protocol state remains encapsulated.

7.8 Continuity Evidence

Some Session operations generate Continuity Evidence.

Whether evidence is generated depends on the observable protocol behavior defined by the Protocol Specification.

Applications SHOULD treat generated evidence as immutable.

Applications MUST NOT attempt to modify Continuity Evidence.

7.9 Session Isolation

Sessions isolate transient execution state.

Closing a Session does not:

- delete identities;
- remove previously generated Continuity Evidence;
- invalidate CEPs;
- modify protocol history.

Applications MAY safely terminate Sessions without affecting historical continuity.

7.10 Error Model

Session operations MAY produce errors including:

Error Category	Typical Cause
Invalid Session	Session is closed or unavailable.
Operation Error	Requested operation failed.
Timeout Error	Operation exceeded execution limits.
Profile Error	Operation not supported by the active profile.
Runtime Error	SDK runtime failure.

Detailed error definitions appear in Chapter 22.

7.11 Best Practices

Applications SHOULD:

- keep Sessions short-lived;
- close Sessions after completing operations;
- avoid sharing mutable Session objects across unrelated workflows;
- create independent Sessions for independent execution contexts;
- treat Session objects as transient runtime resources.

Following these recommendations simplifies resource management while improving application reliability.

7.12 Relationship to the Protocol Specification

The SDK Session is the public programming abstraction corresponding to the protocol execution context defined in the QVTM Protocol Specification.

Applications interact exclusively with the Session object.

Protocol state transitions, message processing, and Continuity Evidence generation remain governed by the Protocol Specification and are intentionally abstracted by the SDK.

8. Identity

8.1 Overview

An **Identity** represents the public SDK abstraction used to perform continuity-aware protocol operations.

An Identity provides the observable context required for Enrollment, Authentication, Recovery, Synchronization, and profile-specific operations.

The SDK intentionally abstracts the underlying identity implementation.

Applications interact with Identity objects rather than protocol-specific identity representations.

8.2 Responsibilities

An Identity is responsible for:

- representing an operational identity;
- participating in protocol operations;
- providing identity metadata;
- exposing observable identity state;
- supporting continuity-preserving lifecycle operations.

An Identity is not responsible for:

- authorization;
 - Trust Evaluation;
 - Continuity Evidence correlation;
 - protocol state management;
 - credential storage implementation.
-

8.3 Published Surface Mapping

The language-neutral Identity surface contains the following semantic members:

```
Identity
identifier()
state()
profile()
attributes()
capabilities()
isAvailable()
```

SDK implementations MAY expose additional convenience APIs.

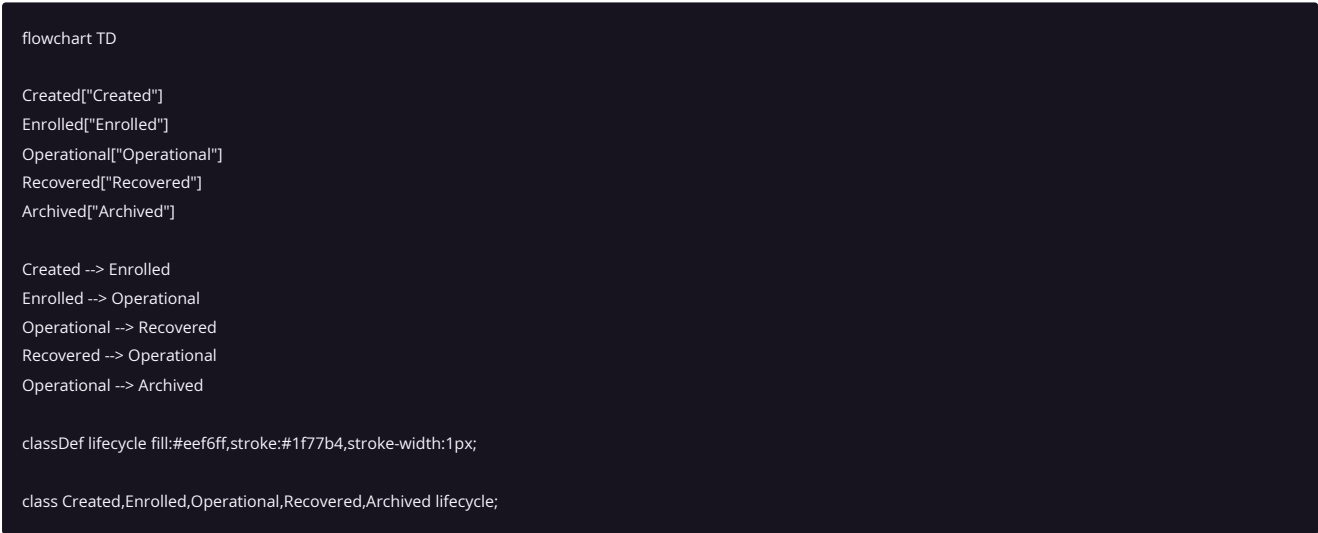
Observable behavior SHOULD remain semantically equivalent across implementations.

8.4 Identity Lifecycle

An Identity evolves throughout its operational lifetime.

The SDK exposes only the observable lifecycle.

Figure 5. Identity Lifecycle



The lifecycle illustrates observable evolution only.
Internal protocol processing remains implementation-specific.

8.5 Identity State

Applications MAY query the observable state of an Identity.
Typical observable states include:

State	Description
Available	Identity can participate in protocol operations.
Suspended	Identity is temporarily unavailable.
Recovering	Recovery is currently in progress.
Archived	Identity is no longer operational.

SDK implementations MAY expose additional informational states.
Applications SHOULD rely only on documented observable behavior.

8.6 Identity Attributes

- An Identity MAY expose public metadata including:
- identifier;
 - active Deployment Profile;
 - creation information;
 - supported capabilities;
 - profile-specific attributes.

Applications SHOULD treat Identity attributes as descriptive metadata.
Applications MUST NOT derive protocol semantics from undocumented attributes.

8.7 Identity Capabilities

Capabilities describe which protocol operations are available for the current Identity.
Typical capabilities include:

- Enrollment
- Authentication
- Synchronization
- Recovery
- Vault Operations
- Secure Messaging
- Secure Upload

Capability availability depends on:

- Deployment Profile;
- SDK implementation;
- runtime configuration;
- observable protocol state.

8.8 Identity Selection

Applications MAY manage multiple Identities.

The SDK SHOULD provide a mechanism for selecting the active Identity before executing protocol operations.

Identity selection is outside the scope of the protocol itself.

Selection behavior depends on the selected Deployment Profile.

8.9 Relationship with Sessions

An Identity participates in one or more Sessions.

Multiple Sessions MAY reference the same Identity.

Likewise, an Identity MAY participate in independent protocol operations over time without affecting its continuity history.

Closing a Session does not invalidate an Identity.

8.10 Relationship with Continuity

Identity continuity is defined by the QVTM Protocol Specification.

The SDK exposes observable identity lifecycle events without exposing internal continuity mechanisms.

Applications interact with observable Identity behavior.

The SDK remains responsible for coordinating protocol interactions.

8.11 Error Model

Identity-related operations MAY produce errors including:

Error Category	Typical Cause
Identity Not Found	Requested identity does not exist.
Identity Unavailable	Identity cannot currently participate in protocol operations.
Identity Archived	Identity is no longer operational.
Profile Error	Requested capability is unsupported.
Runtime Error	SDK runtime failure.

Detailed error semantics are defined in Chapter 22.

8.12 Best Practices

Applications SHOULD:

- treat Identity objects as long-lived resources;
- reuse Identities across multiple Sessions;
- avoid modifying identity metadata outside documented SDK APIs;
- verify capability availability before executing profile-specific operations;
- avoid assuming implementation-specific identity formats.

These recommendations improve portability across SDK implementations and Deployment Profiles.

8.13 Relationship to the Protocol Specification

The Identity object provides the public SDK abstraction corresponding to the protocol identity concepts defined by the QVTM Protocol Specification.

The SDK intentionally hides protocol-specific identity processing.

Applications interact only with observable Identity behavior and documented public contracts.

9. Continuity Evidence

9.1 Overview

Continuity Evidence represents the observable SDK abstraction of continuity-relevant information generated during protocol operations.

Applications do not create Continuity Evidence directly.

Instead, Continuity Evidence is produced by successful SDK operations whenever required by the QVTM Protocol Specification and the active Deployment Profile.

The SDK exposes Continuity Evidence as immutable public objects.

9.2 Responsibilities

Continuity Evidence is responsible for:

- representing one continuity-relevant event;
- providing an immutable reference to generated evidence;
- supporting future protocol operations;
- enabling preservation and synchronization through CEPs;
- providing observable continuity context.

Continuity Evidence is not responsible for:

- evaluating trust;
 - performing authorization;
 - storing application data;
 - exposing internal protocol processing.
-

9.3 Published Surface Mapping

The language-neutral Continuity Evidence surface contains the following semantic members:

```
ContinuityEvidence

identifier()

type()

timestamp()

profile()

metadata()

isValid()
```

The exact naming MAY differ across SDK implementations.

Observable behavior SHOULD remain semantically equivalent.

9.4 Immutability

Once created, Continuity Evidence is immutable.

Applications MUST NOT modify:

- identifiers;
- metadata;
- timestamps;
- evidence type;
- profile association.

Any change to continuity history results in new Continuity Evidence rather than modification of existing evidence.

This guarantees consistency with the append-only continuity model defined by the Protocol Specification.

9.5 Evidence Types

The SDK does not define implementation-specific evidence categories.

Instead, evidence is generated as required by observable protocol behavior.

Typical evidence sources include:

Source	Example
Enrollment	Initial continuity establishment
Authentication	Successful or failed authentication
Synchronization	Evidence exchange
Recovery	Continuity restoration
Vault Operations	Protection of digital assets
Secure Messaging	Conversation continuity
Secure Upload	Protected transfer continuity

Deployment Profiles MAY introduce additional evidence types.

9.6 Metadata

Continuity Evidence MAY expose observable metadata including:

- evidence identifier;
- evidence type;
- creation timestamp;
- protocol version;
- Deployment Profile;
- implementation-independent attributes.

Applications SHOULD treat metadata as descriptive information.

Applications MUST NOT derive undocumented protocol semantics from metadata.

9.7 Generation

Continuity Evidence is generated automatically by SDK operations.

Applications SHOULD NOT attempt to generate evidence manually.

Whether evidence is produced depends entirely on:

- the executed protocol operation;
 - the observable outcome;
 - the active Deployment Profile;
 - the requirements defined by the Protocol Specification.
-

9.8 Lifetime

Continuity Evidence outlives the Session that generated it.

Evidence MAY later participate in:

- Synchronization;
- Recovery;
- Trust Evaluation;
- future protocol operations.

Applications SHOULD assume that Continuity Evidence remains valid until explicitly superseded by protocol evolution.

Historical evidence is never rewritten.

9.9 Relationship with CEP

Continuity Evidence is normally preserved inside one or more Continuity Evidence Packages (CEPs).

Applications interact with individual evidence objects through the SDK.

The SDK manages the relationship between evidence and CEPs.

Applications SHOULD avoid making assumptions about how evidence is packaged internally.

9.10 Observable Guarantees

The SDK guarantees that exposed Continuity Evidence:

- represents an observable protocol event;
- is immutable;
- can participate in future protocol operations;
- is suitable for preservation and synchronization;
- remains independent of implementation-specific internal representations.

The SDK intentionally does not expose internal correlation mechanisms.

9.11 Error Model

Continuity Evidence operations MAY produce errors including:

Error Category	Typical Cause
Evidence Not Found	Requested evidence is unavailable.
Invalid Evidence	Evidence cannot be interpreted.
Unsupported Evidence	Evidence type is unsupported.
Runtime Error	SDK runtime failure.

Detailed error semantics are defined in Chapter 22.

9.12 Best Practices

Applications SHOULD:

- treat Continuity Evidence as immutable;
- store only SDK references when possible;
- avoid duplicating evidence data;
- preserve evidence identifiers when exchanging application data;
- allow the SDK to manage evidence lifecycle.

Applications SHOULD NOT infer protocol state from individual evidence objects.

Continuity is evaluated across observable protocol context rather than individual evidence instances.

9.13 Relationship to the Protocol Specification

The SDK exposes Continuity Evidence as a public programming abstraction.

Its observable behavior corresponds to the Continuity Evidence semantics defined by the QVTM Protocol Specification.

Internal evidence generation, correlation, validation, and lifecycle management remain implementation details and are intentionally hidden from SDK consumers.

10. Continuity Evidence Package (CEP)

10.1 Overview

A **Continuity Evidence Package (CEP)** is the canonical public SDK object used to preserve, transport, and synchronize one or more Continuity Evidence objects.

While Continuity Evidence represents individual continuity-relevant events, a CEP represents the unit through which evidence is preserved and exchanged.

The SDK intentionally abstracts the internal representation of a CEP.

Applications interact with CEPs exclusively through documented public APIs.

10.2 Responsibilities

A CEP is responsible for:

- preserving one or more Continuity Evidence objects;
- providing a portable continuity container;
- supporting synchronization;
- supporting recovery;

- enabling future protocol operations that require preserved continuity information.

A CEP is not responsible for:

- Trust Evaluation;
- authorization;
- protocol execution;
- application data management.

10.3 Published Surface Mapping

The language-neutral CEP surface contains the following semantic members:

```
CEP

identifier()

version()

profile()

evidence()

metadata()

isValid()
```

SDK implementations MAY expose additional convenience APIs.

Observable behavior SHOULD remain semantically equivalent.

10.4 Immutability

Once created, a CEP is immutable.

Applications MUST NOT modify:

- contained Continuity Evidence;
- identifiers;
- metadata;
- protocol version information;
- profile association.

Changes to preserved continuity information result in the creation of new CEPs rather than modification of existing ones.

This behavior preserves the append-only continuity model defined by the QVTM Protocol Specification.

10.5 CEP Contents

A CEP MAY contain:

- one or more Continuity Evidence objects;
- protocol metadata;
- profile metadata;
- integrity information;
- implementation-independent extension metadata.

Applications SHOULD treat CEP contents as read-only.

The SDK manages the internal organization of preserved evidence.

10.6 Preservation

Applications MAY request that a CEP be preserved according to the active Deployment Profile.

Preservation behavior depends on:

- SDK implementation;
- Deployment Profile;
- runtime configuration;
- application policy.

The SDK intentionally abstracts storage implementation details.

Applications interact only with observable preservation behavior.

10.7 Synchronization

A CEP MAY participate in Synchronization operations.

Synchronization exchanges preserved continuity information between compatible protocol participants.

Applications invoke synchronization through SDK operations.

Applications SHOULD NOT manipulate CEP synchronization directly.

Synchronization semantics are defined by the QVTM Protocol Specification.

10.8 Recovery

Recovery operations MAY consume one or more CEPs.

The SDK determines which preserved continuity information is required for successful recovery.

Applications SHOULD provide CEPs only through documented SDK interfaces.

The SDK remains responsible for protocol coordination.

10.9 Relationship with Continuity Evidence

A CEP contains one or more Continuity Evidence objects.

Applications MAY inspect contained evidence using public SDK APIs.

Applications SHOULD avoid making assumptions regarding:

- evidence ordering;
- internal storage layout;
- packaging strategies;
- implementation-specific optimizations.

These behaviors remain implementation-specific.

10.10 Observable Guarantees

The SDK guarantees that every exposed CEP:

- preserves immutable Continuity Evidence;
- represents observable protocol information;
- supports future protocol operations;
- can participate in synchronization;
- remains compatible with the selected Deployment Profile.

The SDK intentionally hides serialization details.

Wire formats are defined by the Protocol Specification.

10.11 Error Model

CEP operations MAY produce errors including:

Error Category	Typical Cause
CEP Not Found	Requested CEP is unavailable.
Invalid CEP	CEP validation failed.
Unsupported CEP	CEP version or profile is unsupported.
Synchronization Error	CEP cannot participate in synchronization.
Runtime Error	SDK runtime failure.

Detailed error semantics are defined in Chapter 22.

10.12 Best Practices

Applications SHOULD:

- treat CEPs as immutable;
- preserve CEP identifiers when exchanging application data;
- avoid modifying preserved continuity information;
- allow the SDK to manage CEP lifecycle;
- use public SDK operations for preservation and synchronization.

Applications SHOULD NOT interpret internal CEP structure.

Only observable SDK contracts are considered stable.

10.13 Relationship to the Protocol Specification

The SDK exposes the Continuity Evidence Package as the public programming abstraction corresponding to the CEP defined by the QVTM Protocol Specification.

Serialization, canonical encoding, integrity protection, and protocol interoperability remain governed by the Protocol Specification and are intentionally abstracted by the SDK.

11. Trust Evaluation

11.1 Overview

Trust Evaluation is the public SDK abstraction through which applications obtain the observable continuity assessment produced by QVTM.

Applications consume Trust Evaluation results to support business decisions.

The SDK intentionally separates continuity evaluation from authorization.

A Trust Evaluation result describes the continuity context observed by the protocol.

It does not authorize or deny application behavior.

11.2 Responsibilities

Trust Evaluation is responsible for:

- exposing observable continuity results;

- summarizing protocol evaluation outcomes;
- providing continuity context to applications;
- supporting profile-specific workflows.

Trust Evaluation is not responsible for:

- authorization;
- access control;
- business policy;
- application logic;
- internal evidence correlation.

11.3 Published Surface Mapping

The language-neutral Trust Evaluation surface contains the following semantic members:

```
TrustEvaluation

result()

context()

profile()

timestamp()

references()

isSuccessful()
```

SDK implementations MAY expose additional convenience APIs.

Observable behavior SHOULD remain semantically equivalent.

11.4 Evaluation Results

The SDK exposes observable evaluation outcomes.

Typical result categories include:

Result	Description
Continuous	Available Continuity Evidence supports the requested operation.
ContinuousWithConditions	Continuity is supported but additional application policy may apply.
Indeterminate	Available evidence is insufficient for a definitive continuity assessment.
Discontinuous	Continuity requirements are not satisfied.

Applications SHOULD interpret these results according to local policy.

The meaning of each category is defined by the QVTM Protocol Specification.

11.5 Evaluation Context

A Trust Evaluation MAY expose observable contextual information including:

- evaluation timestamp;
- Deployment Profile;
- protocol version;

- operation context;
- references to generated Continuity Evidence;
- references to related CEPs.

The SDK intentionally omits implementation-specific evaluation details.

11.6 Evaluation Requests

Applications MAY request Trust Evaluation through public SDK operations.

Depending on the Deployment Profile, evaluation MAY occur during:

- Enrollment;
- Authentication;
- Synchronization;
- Recovery;
- Vault Operations;
- Secure Messaging;
- Secure Upload.

Applications SHOULD rely on the SDK to determine when evaluation is required.

11.7 Relationship with Continuity Evidence

Trust Evaluation consumes Continuity Evidence produced by protocol operations.

Applications interact only with the resulting observable evaluation.

The SDK intentionally hides:

- evidence correlation;
- continuity path selection;
- internal scoring;
- proprietary algorithms;
- optimization strategies.

These mechanisms remain implementation-specific.

11.8 Relationship with CEP

Trust Evaluation MAY reference one or more CEPs that contributed to the observable result.

Applications SHOULD treat these references as informational.

Applications MUST NOT assume that every contributing CEP is exposed by the SDK.

The SDK remains responsible for selecting the appropriate continuity context.

11.9 Observable Guarantees

The SDK guarantees that every Trust Evaluation:

- represents an observable protocol result;
- is independent of application authorization;
- can be consumed through stable SDK contracts;
- preserves implementation independence.

The SDK intentionally does not guarantee a particular authorization outcome.

Authorization always remains the responsibility of the application.

11.10 Error Model

Trust Evaluation operations MAY produce errors including:

Error Category	Typical Cause
Evaluation Error	Continuity evaluation could not be completed.
Context Error	Required evaluation context is unavailable.
Unsupported Profile	Evaluation is unavailable for the active Deployment Profile.
Runtime Error	SDK runtime failure.

Detailed error semantics are defined in Chapter 22.

11.11 Best Practices

Applications SHOULD:

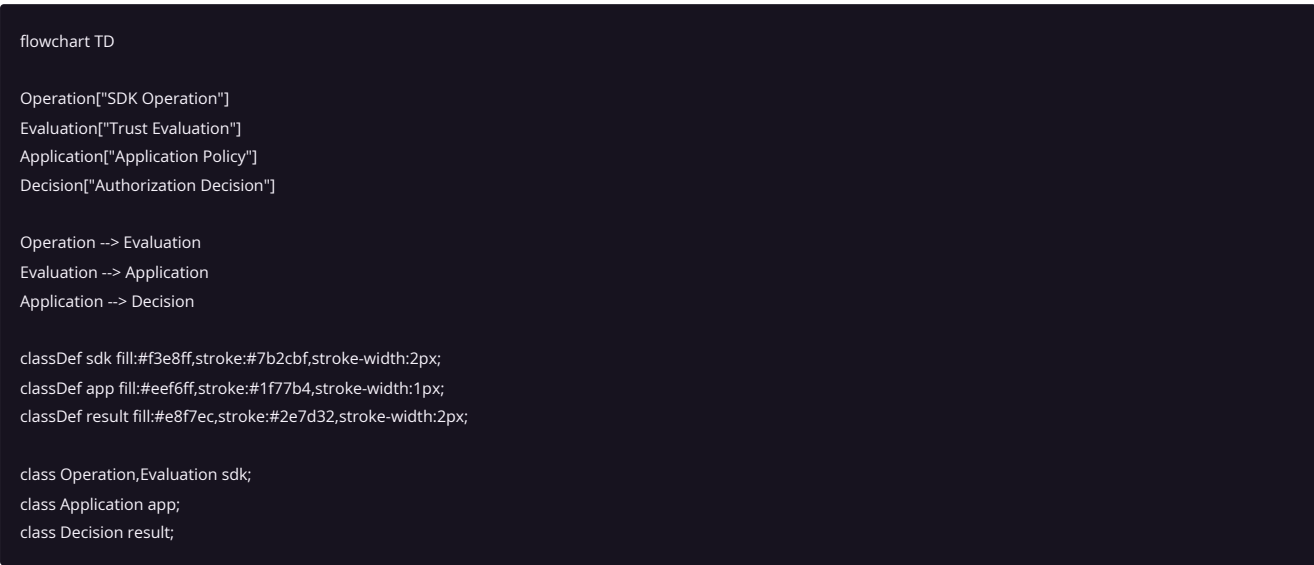
- use Trust Evaluation to inform business decisions;
- avoid deriving authorization directly from evaluation results;
- preserve evaluation references when auditing protocol operations;
- avoid assuming implementation-specific evaluation behavior;
- rely exclusively on documented SDK contracts.

Applications SHOULD NOT attempt to reproduce Trust Evaluation internally.

11.12 Relationship with Application Authorization

Trust Evaluation and authorization are intentionally independent.

Figure 6. Trust Evaluation and Application Decision



The SDK produces the observable continuity assessment.

The application remains solely responsible for determining the operational consequence of that assessment.

11.13 Relationship to the Protocol Specification

The Trust Evaluation object provides the public SDK abstraction corresponding to the Trust Evaluation defined by the QVTM Protocol Specification.

Internal evaluation mechanisms, Graph Puzzle processing, continuity correlation, and evaluation algorithms remain implementation details and are intentionally hidden from SDK consumers.

Part III — Operations

The Operations layer defines the public SDK workflows used to perform continuity-aware protocol operations.

Each operation represents an observable interaction between an application and the QVTM SDK.

Operations intentionally abstract the underlying protocol implementation while preserving the observable behavior defined by the QVTM Protocol Specification.

The SDK currently defines the following primary operations:

- Enrollment
- Authentication
- Synchronization
- Recovery
- Vault Operations

Additional operations MAY be introduced by future SDK versions without altering the semantics of existing operations.

12. Enrollment

12.1 Overview

Enrollment establishes the initial operational relationship between an application, an Identity, and the selected Deployment Profile.

Enrollment is typically the first protocol operation executed for a newly created Identity.

Successful enrollment establishes the continuity context required for subsequent protocol operations.

Applications interact exclusively with the public SDK APIs.

The SDK coordinates all protocol-specific behavior.

12.2 Responsibilities

Enrollment is responsible for:

- initializing an operational Identity;
- establishing the initial continuity context;
- creating protocol state required for future operations;
- generating Continuity Evidence when required;
- creating CEPs when required.

Enrollment is not responsible for:

- application authorization;
 - long-term identity management;
 - Trust Evaluation algorithms;
 - synchronization.
-

12.3 Published Surface Mapping

The language-neutral Enrollment operation is:

The released binding or OpenAPI contract is authoritative for the exact name and signature.

12.4 Prerequisites

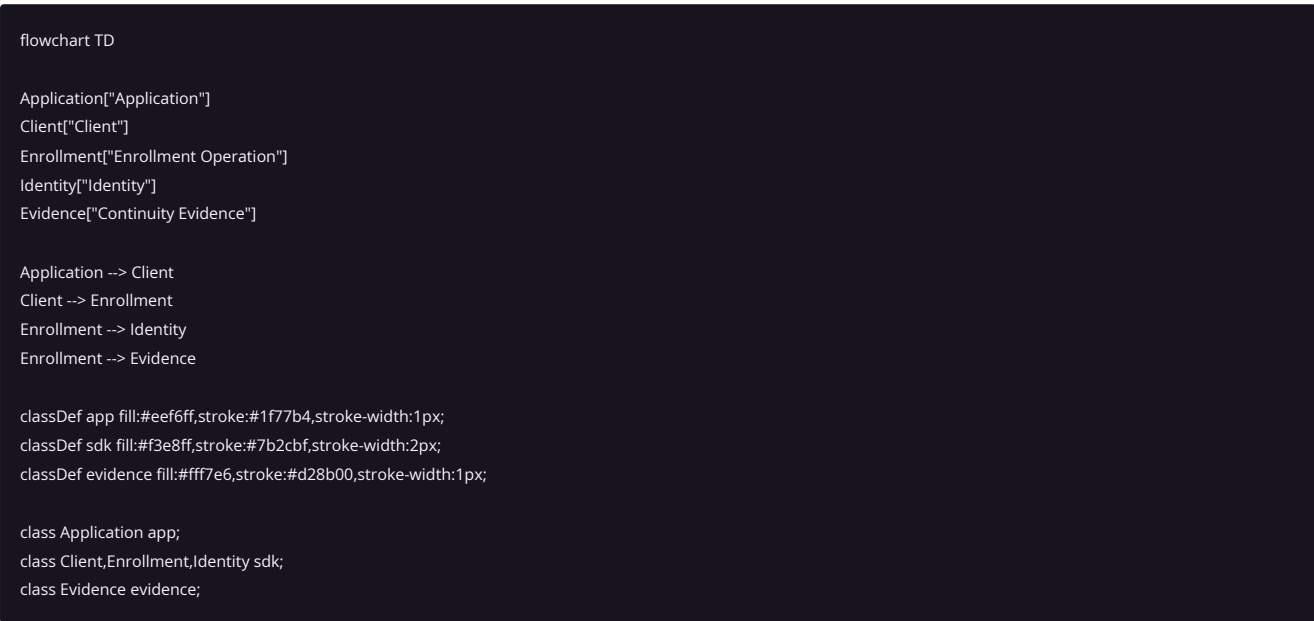
Before invoking Enrollment, applications SHOULD ensure that:

- the SDK has been initialized;
- a valid Client is available;
- the selected Deployment Profile supports Enrollment;
- required runtime configuration has been completed.

Additional profile-specific requirements MAY apply.

12.5 Observable Workflow

Figure 7. Enrollment Workflow



Enrollment establishes the observable continuity foundation used by future protocol operations.

12.6 Result

A successful Enrollment operation MAY return:

- Enrollment status;
- Identity reference;
- generated Continuity Evidence;
- generated CEP references;
- profile-specific information.

The SDK intentionally abstracts protocol-specific artifacts.

12.7 Generated Objects

Depending on the active Deployment Profile, Enrollment MAY generate:

Object	Generated
Identity	Yes
Continuity Evidence	When required
CEP	When required

Object	Generated
Trust Evaluation	When required by the operation

The exact observable results depend on the selected Deployment Profile.

12.8 Error Model

Enrollment MAY produce errors including:

Error Category	Typical Cause
Enrollment Error	Enrollment could not be completed.
Configuration Error	Required configuration is unavailable.
Identity Error	Identity cannot be created.
Profile Error	Enrollment unsupported for the selected profile.
Runtime Error	SDK runtime failure.

Detailed error semantics are defined in Chapter 22.

12.9 Best Practices

Applications SHOULD:

- perform Enrollment only when establishing a new operational identity;
- preserve returned SDK objects;
- allow the SDK to manage generated Continuity Evidence;
- avoid repeating successful Enrollment unnecessarily;
- treat Enrollment as the beginning of a continuity lifecycle.

12.10 Relationship to the Protocol Specification

The Enrollment operation corresponds to the Enrollment Protocol Operation defined by the QVTM Protocol Specification.

The SDK exposes only the observable programming model.

Protocol negotiation, message exchange, and internal state transitions remain implementation details.

13. Authentication

13.1 Overview

Authentication verifies the continuity of an existing Identity within the current protocol context.

Unlike Enrollment, Authentication does not establish a new identity relationship.

Instead, it confirms that an existing continuity relationship remains valid according to the QVTM Protocol Specification and the active Deployment Profile.

Applications invoke Authentication through the SDK without interacting directly with protocol messages or cryptographic mechanisms.

13.2 Responsibilities

Authentication is responsible for:

- verifying an existing Identity;

- extending continuity when appropriate;
- generating Continuity Evidence when required;
- exposing observable authentication results;
- supporting profile-specific authentication workflows.

Authentication is not responsible for:

- creating new identities;
- authorization decisions;
- credential management;
- application session management.

13.3 Published Surface Mapping

The language-neutral Authentication operation is:

Language-specific names are valid only when published by that binding; OpenAPI `operationId` values preserve cross-language traceability.

13.4 Prerequisites

Before invoking Authentication, applications SHOULD ensure that:

- the SDK has been initialized;
- a valid Client is available;
- an operational Identity has been selected;
- the active Deployment Profile supports Authentication.

Additional profile-specific prerequisites MAY apply.

13.5 Observable Workflow

Figure 8. Authentication Workflow

```
flowchart TD
    Application["Application"]
    Client["Client"]
    Authentication["Authentication Operation"]
    Identity["Identity"]
    Evaluation["Trust Evaluation"]

    Application --> Client
    Client --> Authentication
    Authentication --> Identity
    Authentication --> Evaluation

    classDef app fill:#eef6ff,stroke:#1f77b4,stroke-width:1px;
    classDef sdk fill:#f3e8ff,stroke:#7b2cbf,stroke-width:2px;
    classDef result fill:#e8f7ec,stroke:#2e7d32,stroke-width:2px;

    class Application app;
    class Client,Authentication,Identity sdk;
    class Evaluation result;
```

Authentication verifies continuity and returns an observable protocol result.

Applications remain responsible for deciding how that result affects local authorization.

13.6 Result

A successful Authentication operation MAY return:

- authentication status;
- Identity reference;
- Trust Evaluation;
- generated Continuity Evidence;
- generated CEP references;
- profile-specific information.

Returned objects depend on the active Deployment Profile.

13.7 Generated Objects

Authentication MAY generate:

Object	Generated
Continuity Evidence	When required
CEP	When required
Trust Evaluation	Yes
Identity	Existing Identity reference

Authentication normally operates on an existing Identity.

13.8 Continuity Extension

Successful Authentication MAY extend an existing continuity history.

Applications SHOULD understand that extending continuity does not modify historical Continuity Evidence.

Instead, new Continuity Evidence MAY be generated describing the completed authentication operation.

13.9 Error Model

Authentication MAY produce errors including:

Error Category	Typical Cause
Authentication Error	Authentication failed.
Identity Error	Identity unavailable.
Session Error	No valid execution context exists.
Profile Error	Authentication unsupported.
Runtime Error	SDK runtime failure.

Detailed error semantics are defined in Chapter 22.

13.10 Best Practices

Applications SHOULD:

- authenticate before executing protected operations when required by the active Deployment Profile;
- preserve returned Trust Evaluation results;

- avoid performing redundant Authentication operations;
 - rely on observable SDK results rather than implementation-specific behavior;
 - allow the SDK to manage generated Continuity Evidence.
-

13.11 Relationship to the Protocol Specification

The Authentication operation corresponds to the Authentication Protocol Operation defined by the QVTM Protocol Specification.

The SDK intentionally abstracts protocol handshakes, message exchange, and internal continuity processing while preserving observable behavior.

14. Synchronization

14.1 Overview

Synchronization exchanges eligible continuity information between compatible QVTM participants.

Synchronization extends the locally available continuity context without altering previously preserved Continuity Evidence.

Applications invoke Synchronization through the SDK.

The SDK coordinates protocol execution and preserves the synchronization semantics defined by the QVTM Protocol Specification.

14.2 Responsibilities

Synchronization is responsible for:

- exchanging eligible Continuity Evidence;
- exchanging eligible CEPs;
- extending locally available continuity information;
- exposing observable synchronization results;
- supporting offline-to-online continuity transitions.

Synchronization is not responsible for:

- rewriting historical continuity;
 - resolving application conflicts;
 - authorization;
 - modifying application data.
-

14.3 Published Surface Mapping

The language-neutral Synchronization operation is:

Language-specific names are valid only when published by that binding; OpenAPI `operationId` values preserve cross-language traceability.

14.4 Prerequisites

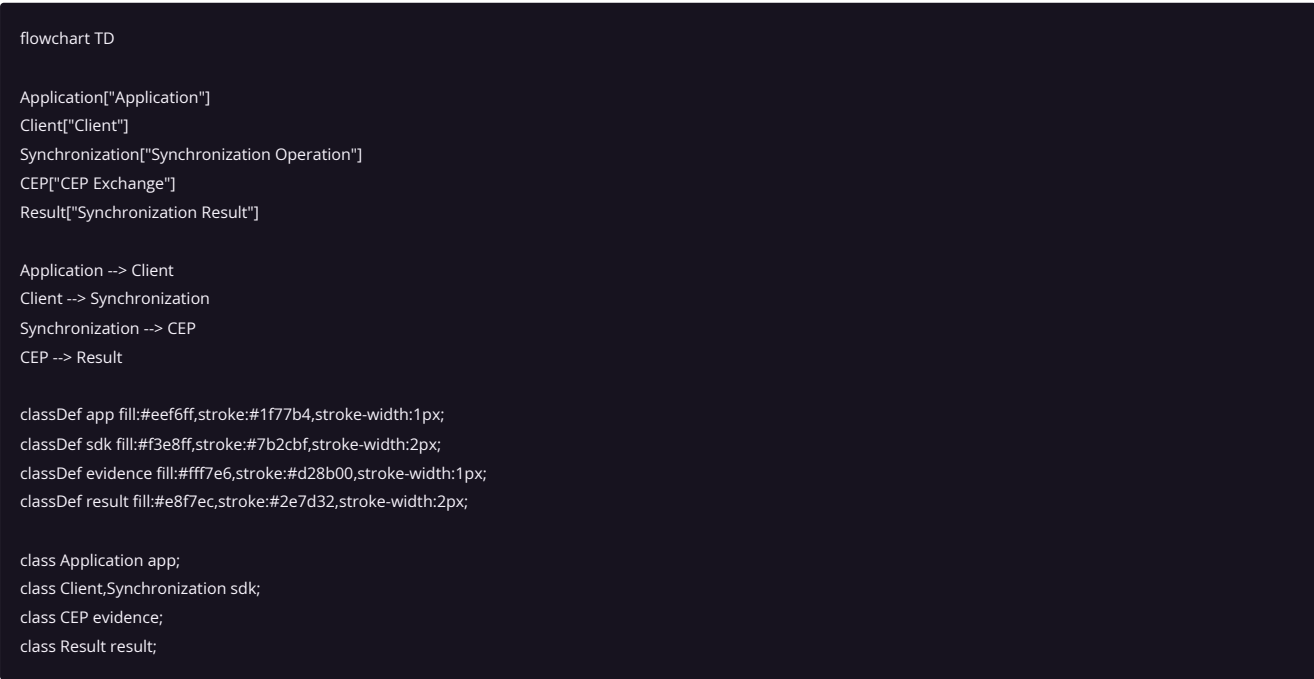
Before invoking Synchronization, applications SHOULD ensure that:

- the SDK has been initialized;
- a valid Client is available;
- an operational Identity is available when required;
- the selected Deployment Profile supports Synchronization.

Some Deployment Profiles MAY perform Synchronization automatically.

14.5 Observable Workflow

Figure 9. Synchronization Workflow



Synchronization extends available continuity information without modifying previously preserved protocol history.

14.6 Result

A successful Synchronization operation MAY return:

- synchronization status;
- accepted Continuity Evidence;
- accepted CEPs;
- synchronization statistics;
- profile-specific information.

Applications SHOULD interpret the result as an observable protocol outcome.

14.7 Generated Objects

Synchronization MAY generate:

Object	Generated
Continuity Evidence	When required
CEP	When required
Trust Evaluation	When required
Synchronization Result	Yes

Not every Synchronization operation produces new Continuity Evidence.

Generation depends on observable protocol behavior.

14.8 Offline Operation

One of the primary goals of Synchronization is to support disconnected operation.

Applications MAY continue generating continuity-relevant protocol activity while offline.

Once connectivity becomes available, Synchronization exchanges eligible continuity information without restarting protocol history.

Applications SHOULD rely on the SDK to determine which information is eligible for synchronization.

14.9 Error Model

Synchronization MAY produce errors including:

Error Category	Typical Cause
Synchronization Error	Synchronization could not be completed.
Compatibility Error	Participants are incompatible.
Profile Error	Synchronization unsupported.
Connectivity Error	Required communication unavailable.
Runtime Error	SDK runtime failure.

Detailed error semantics are defined in Chapter 22.

14.10 Best Practices

Applications SHOULD:

- synchronize whenever connectivity becomes available;
- allow the SDK to determine synchronization scope;
- avoid implementing custom evidence exchange;
- preserve synchronization results for diagnostic purposes when appropriate;
- treat Synchronization as an incremental operation.

Applications SHOULD NOT assume that every local object is synchronized during a single operation.

14.11 Relationship to the Protocol Specification

The Synchronization operation corresponds to the Synchronization Protocol Operation defined by the QVTM Protocol Specification.

The SDK intentionally abstracts protocol negotiation, evidence exchange, duplicate detection, and validation while preserving observable synchronization behavior.

15. Recovery

15.1 Overview

Recovery restores operational capability while preserving the continuity established by previous protocol operations.

Recovery is intended for situations in which an operational context can no longer continue normally, such as device replacement, identity restoration, or other continuity-preserving recovery scenarios supported by the active Deployment Profile.

Recovery extends an existing continuity history.

It does not create a new continuity relationship.

Applications invoke Recovery through the SDK.

The SDK coordinates the protocol workflow while exposing only observable behavior.

15.2 Responsibilities

Recovery is responsible for:

- restoring operational capability;
- extending an existing continuity history;
- exposing observable recovery results;
- generating Continuity Evidence when required;
- coordinating recovery-specific protocol operations.

Recovery is not responsible for:

- recreating historical continuity;
- application authorization;
- backup management;
- implementation-specific recovery mechanisms.

15.3 Published Surface Mapping

The language-neutral Recovery operation is:

SDK implementations MAY adapt naming conventions according to implementation language.

Observable behavior SHOULD remain semantically equivalent.

15.4 Prerequisites

Before invoking Recovery, applications SHOULD ensure that:

- the SDK has been initialized;
- a valid Client is available;
- the selected Deployment Profile supports Recovery;
- the required recovery context is available.

Additional profile-specific requirements MAY apply.

15.5 Observable Workflow

Figure 10. Recovery Workflow

flowchart TD

```
Application["Application"]
Client["Client"]
Recovery["Recovery Operation"]
Identity["Operational Identity"]
Evaluation["Trust Evaluation"]
```

```
Application --> Client
Client --> Recovery
Recovery --> Identity
Recovery --> Evaluation
```

```
classDef app fill:#eef6ff,stroke:#1f77b4,stroke-width:1px;
classDef sdk fill:#f3e8ff,stroke:#7b2cbf,stroke-width:2px;
classDef result fill:#e8f7ec,stroke:#2e7d32,stroke-width:2px;
```

```
class Application app;
class Client,Recovery,Identity sdk;
class Evaluation result;
```

Recovery restores operational capability while preserving observable continuity.

15.6 Result

A successful Recovery operation MAY return:

- recovery status;
- restored Identity;
- Trust Evaluation;
- generated Continuity Evidence;
- generated CEP references;
- profile-specific information.

The returned objects depend on the active Deployment Profile.

15.7 Generated Objects

Recovery MAY generate:

Object	Generated
Continuity Evidence	When required
CEP	When required
Trust Evaluation	Yes
Identity	Restored operational Identity

Applications SHOULD treat generated objects as new additions to the continuity history.

Previously generated objects remain unchanged.

15.8 Continuity Preservation

Recovery preserves continuity by extending the existing history.

Applications SHOULD understand that Recovery never modifies historical Continuity Evidence.

Instead, Recovery MAY generate additional Continuity Evidence describing the recovery event.

Historical continuity remains intact.

15.9 Recovery Scenarios

Typical Recovery scenarios include:

- device replacement;
- operational identity restoration;
- cryptographic migration;
- Vault restoration;
- Deployment Profile-specific recovery procedures.

Individual SDK implementations MAY support only a subset of these scenarios.

15.10 Error Model

Recovery MAY produce errors including:

Error Category	Typical Cause
Recovery Error	Recovery could not be completed.
Recovery Context Error	Required recovery information unavailable.
Identity Error	Identity cannot be restored.
Profile Error	Recovery unsupported.
Runtime Error	SDK runtime failure.

Detailed error semantics are defined in Chapter 22.

15.11 Best Practices

Applications SHOULD:

- invoke Recovery only when normal operation cannot continue;
- preserve Recovery results for subsequent operations;
- avoid repeating successful Recovery unnecessarily;
- allow the SDK to manage recovery-specific Continuity Evidence;
- resume normal operation only after successful Recovery.

Applications SHOULD avoid implementing recovery workflows outside documented SDK APIs.

15.12 Relationship to the Protocol Specification

The Recovery operation corresponds to the Recovery Protocol Operation defined by the QVTM Protocol Specification.

The SDK intentionally abstracts recovery negotiation, validation, evidence processing, and protocol state transitions while preserving observable recovery behavior.

16. Vault Operations

16.1 Overview

The **Vault Operations** API provides the public SDK interface for protecting, accessing, and managing digital assets using the QVTM Vault Deployment Profile.

Vault Operations extend the Core SDK with functionality specific to long-term confidentiality, offline preservation, recovery, and continuity-aware protection of digital assets.

Applications interact exclusively with the public SDK APIs.

The SDK coordinates all protocol-specific processing and cryptographic operations.

16.2 Responsibilities

Vault Operations are responsible for:

- protecting digital assets;
- opening protected assets;
- updating protected assets;
- preserving continuity information;
- supporting synchronization;
- supporting recovery workflows.

Vault Operations are not responsible for:

- file management outside the Vault;

- application authorization;
 - storage implementation;
 - cryptographic algorithm selection;
 - Trust Evaluation algorithms.
-

16.3 Published Surface Mapping

The language-neutral Vault operation set is:

SDK implementations MAY expose additional convenience APIs.

Observable behavior SHOULD remain semantically equivalent.

16.4 Protected Objects

The Vault SDK operates on protected digital assets.

Typical protected assets include:

- documents;
- images;
- archives;
- credentials;
- cryptographic material;
- application-specific data.

The SDK intentionally does not prescribe supported file formats.

Protection applies to arbitrary application data.

16.5 Protecting Assets

Applications invoke **protect()** to create protected Vault objects.

The SDK performs the required protocol operations transparently.

A successful protection operation MAY generate:

- protected Vault object;
- Continuity Evidence;
- CEP references;
- Trust Evaluation;
- profile-specific metadata.

Applications SHOULD preserve returned SDK objects.

16.6 Opening Protected Assets

Applications invoke **open()** to access previously protected assets.

The SDK validates the required continuity context before exposing the protected content.

Applications SHOULD consume only the observable SDK result.

Validation procedures remain implementation-specific.

16.7 Updating Protected Assets

Applications MAY modify previously protected assets through **update()**.

Updating an asset does not modify historical continuity.

Instead, the SDK MAY generate new Continuity Evidence describing the updated protected object.

Historical protocol information remains unchanged.

16.8 Removing Protected Assets

Applications MAY remove protected Vault objects using **remove()**.

Removal affects application availability of the protected asset.

Removal does not rewrite previously generated Continuity Evidence.

Depending on the Deployment Profile, the SDK MAY generate additional Continuity Evidence describing the removal event.

16.9 Listing Protected Assets

Applications MAY enumerate available protected assets using **list()**.

The returned information is limited to observable SDK metadata.

The SDK intentionally abstracts:

- storage organization;
 - indexing strategies;
 - persistence mechanisms;
 - implementation-specific optimizations.
-

16.10 Synchronization

Vault Operations MAY participate in Synchronization.

Synchronization exchanges eligible continuity information while preserving previously generated protocol history.

Applications SHOULD invoke synchronization through the public SDK API.

The SDK determines which protected information is eligible for exchange.

16.11 Recovery

Vault Recovery restores access to protected assets while preserving continuity.

Recovery does not recreate Vault history.

Instead, Recovery extends the existing continuity history according to the QVTM Protocol Specification.

Applications SHOULD invoke Recovery only through documented SDK operations.

16.12 Observable Workflow

Figure 11. Vault Protection Workflow

flowchart TD

```
Application["Application"]
Vault["Vault Operation"]
Protection["Protected Asset"]
Evidence["Continuity Evidence"]
CEP["CEP"]
```

```
Application --> Vault
Vault --> Protection
Vault --> Evidence
Evidence --> CEP
```

```
classDef app fill:#eef6ff,stroke:#1f77b4,stroke-width:1px;
classDef sdk fill:#f3e8ff,stroke:#7b2cbf,stroke-width:2px;
classDef evidence fill:#fff7e6,stroke:#d28b00,stroke-width:1px;
```

```
class Application app;
class Vault,Protection sdk;
class Evidence,CEP evidence;
```

Vault Operations protect digital assets while preserving observable continuity information.

16.13 Error Model

Vault Operations MAY produce errors including:

Error Category	Typical Cause
Protection Error	Asset protection failed.
Open Error	Protected asset cannot be opened.
Update Error	Protected asset cannot be updated.
Removal Error	Protected asset cannot be removed.
Recovery Error	Vault Recovery failed.
Synchronization Error	Vault Synchronization failed.
Runtime Error	SDK runtime failure.

Detailed error semantics are defined in Chapter 22.

16.14 Best Practices

Applications SHOULD:

- use Vault Operations exclusively through the SDK;
- preserve returned SDK objects;
- synchronize protected assets regularly when appropriate;
- avoid modifying protected objects outside documented SDK APIs;
- allow the SDK to manage continuity-related artifacts.

Applications SHOULD treat Vault objects as long-lived protected resources.

16.15 Relationship to the Protocol Specification

Vault Operations compose multiple protocol operations defined by the QVTM Protocol Specification, including Authentication, Synchronization, Recovery, Continuity Evidence generation, and CEP management.

The SDK intentionally abstracts protocol sequencing and internal processing while preserving the observable behavior required by the selected Vault Deployment Profile.

Part IV — Profiles

The QVTM SDK supports multiple **Deployment Profiles**.

A Deployment Profile adapts the common SDK object model to a specific application domain without modifying the semantics defined by the QVTM Protocol Specification.

Every Deployment Profile:

- reuses the Core SDK;
- preserves Trust Continuity semantics;
- preserves Continuity Evidence semantics;
- preserves CEP semantics;
- exposes only profile-specific behavior.

Applications interact with Profiles through the same Client, Session, Identity, and operation model described in the previous chapters.

17. Login Profile

17.1 Overview

The **Login Profile** enables passwordless authentication workflows based on Trust Continuity.

Rather than introducing a separate SDK model, the Login Profile specializes the Core SDK for identity continuity and authentication scenarios.

The Login Profile is intended for applications requiring secure user authentication while preserving continuity across devices, credential evolution, and recovery events.

17.2 Responsibilities

The Login Profile is responsible for:

- passwordless authentication;
- identity continuity;
- device continuity;
- enrollment;
- authentication;
- recovery.

The Login Profile is not responsible for:

- application authorization;
- user provisioning;
- access control;
- business policy.

17.3 Supported Operations

The Login Profile typically supports:

Operation	Supported
Enrollment	Yes
Authentication	Yes
Synchronization	Optional

Operation	Supported
Recovery	Yes
Vault Operations	No

Additional capabilities MAY be supported by individual SDK implementations.

17.4 Typical Workflow

A typical Login Profile integration follows these stages:

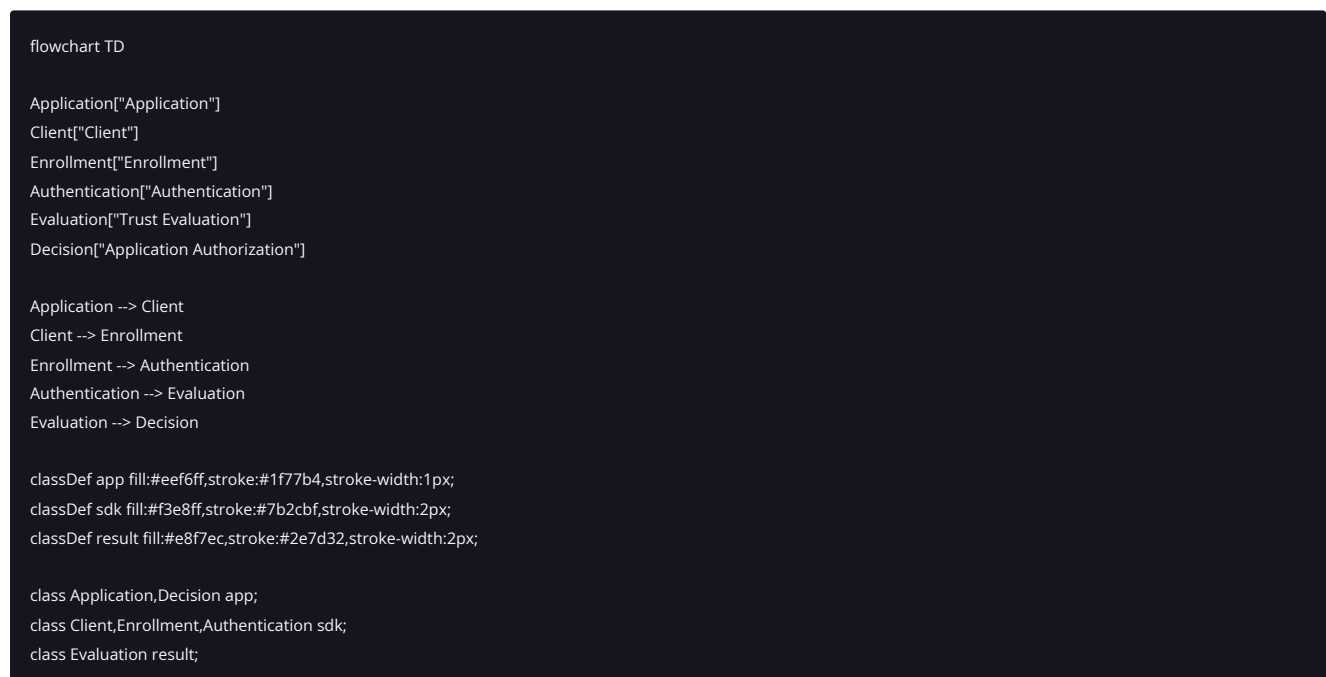
1. Initialize the SDK.
2. Create or obtain a Client.
3. Select or create an Identity.
4. Execute Enrollment when required.
5. Authenticate the Identity.
6. Consume the returned Trust Evaluation.
7. Apply application-specific authorization.

Authentication does not replace application authorization.

The application remains responsible for deciding whether access is granted.

17.5 Observable Workflow

Figure 12. Login Profile Workflow



17.6 Generated Objects

Depending on the executed operations, the Login Profile MAY generate:

- Identity;
- Continuity Evidence;
- CEP;
- Trust Evaluation.

The generated objects follow the same contracts defined by the Core SDK.

17.7 Error Model

The Login Profile MAY produce all Core SDK errors together with profile-specific authentication failures.

Applications SHOULD handle these errors using the common SDK error model described in Chapter 22.

17.8 Best Practices

Applications SHOULD:

- reuse a long-lived Client;
 - authenticate only when required;
 - preserve Trust Evaluation results;
 - avoid implementing custom authentication workflows;
 - rely exclusively on documented SDK APIs.
-

17.9 Relationship to the Protocol Specification

The Login Profile specializes the Enrollment, Authentication, Recovery, and Trust Evaluation operations defined by the QVTM Protocol Specification.

It does not introduce new protocol semantics.

Only the observable integration workflow is profile-specific.

18. Vault Profile

18.1 Overview

The **Vault Profile** extends the Core SDK with capabilities for protecting digital assets while preserving Trust Continuity.

Unlike the Login Profile, which focuses on operational identity, the Vault Profile focuses on long-term confidentiality, offline operation, preservation, synchronization, and recovery of protected assets.

The Vault Profile builds upon the same Core SDK object model.

Applications continue to interact with Client, Session, Identity, CEP, and Trust Evaluation through identical public contracts.

18.2 Responsibilities

The Vault Profile is responsible for:

- protecting digital assets;
- preserving protected assets;
- supporting offline operation;
- synchronizing preserved continuity information;
- recovering protected assets;
- extending continuity during asset lifecycle events.

The Vault Profile is not responsible for:

- file management outside the Vault;
 - authorization;
 - storage implementation;
 - cryptographic algorithm selection;
 - application backup strategies.
-

18.3 Supported Operations

The Vault Profile typically supports:

Operation	Supported
Enrollment	Yes
Authentication	Yes
Synchronization	Yes
Recovery	Yes
Vault Operations	Yes

Additional capabilities MAY be introduced by future SDK versions.

18.4 Typical Workflow

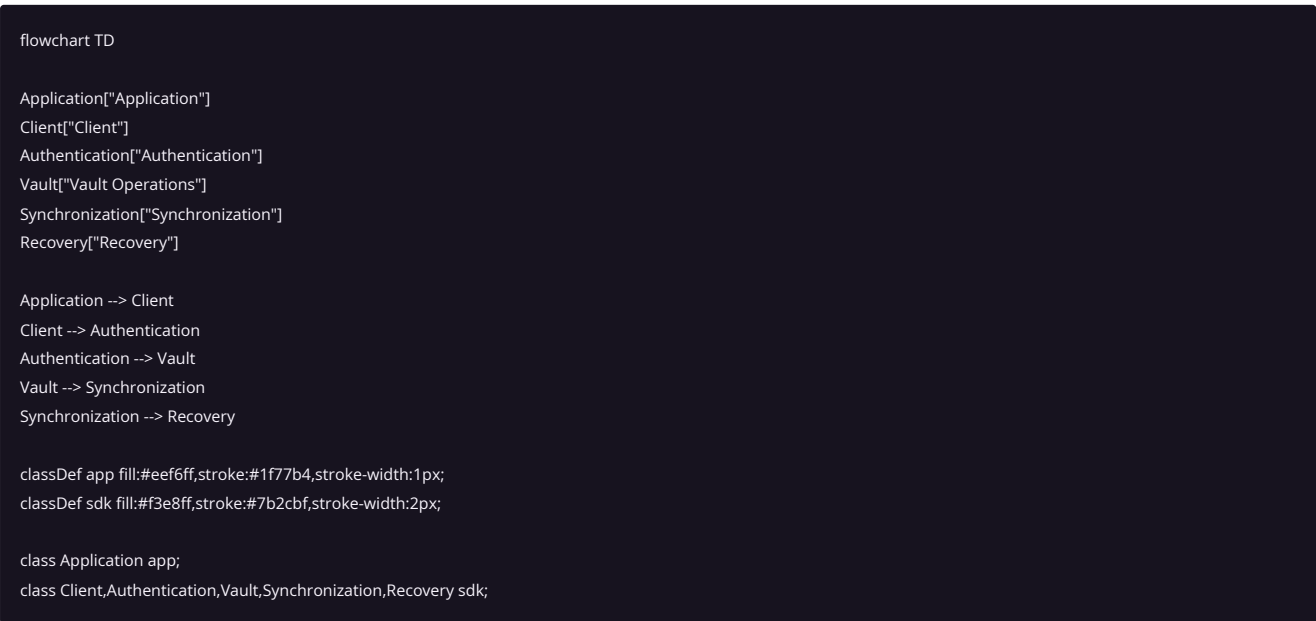
A typical Vault integration follows these stages:

1. Initialize the SDK.
2. Create a Client.
3. Select an operational Identity.
4. Authenticate the Identity when required.
5. Protect one or more digital assets.
6. Synchronize preserved continuity information when appropriate.
7. Recover protected assets if necessary.

The exact workflow depends on the selected Deployment Profile configuration.

18.5 Observable Workflow

Figure 13. Vault Profile Workflow



The workflow illustrates a typical operational lifecycle.

Applications MAY execute only the operations required by their deployment.

18.6 Protected Assets

The Vault Profile supports arbitrary application data.

Typical protected assets include:

- documents;
- images;
- archives;
- credentials;
- cryptographic material;
- application databases;
- private application data.

The SDK intentionally remains independent of file format.

18.7 Offline Operation

One of the defining characteristics of the Vault Profile is support for offline operation.

Applications MAY continue protecting assets without continuous connectivity.

Synchronization MAY occur later without restarting continuity history.

The SDK manages continuity preservation throughout disconnected operation.

18.8 Continuity Preservation

The Vault Profile preserves continuity throughout the lifecycle of protected assets.

Operations such as:

- protection;
- update;
- synchronization;
- recovery;
- removal;

MAY generate additional Continuity Evidence when required by the Protocol Specification.

Historical continuity is never modified.

18.9 Generated Objects

Vault operations MAY generate:

- protected assets;
- Continuity Evidence;
- CEPs;
- Trust Evaluation;
- Vault-specific metadata.

Applications SHOULD preserve returned SDK objects whenever appropriate.

18.10 Error Model

The Vault Profile MAY produce all Core SDK errors together with profile-specific Vault errors.

Typical additional errors include:

Error Category	Typical Cause
Asset Protection Error	Protection failed.
Asset Access Error	Protected asset unavailable.
Asset Update Error	Update could not be completed.
Asset Recovery Error	Recovery failed.
Vault Synchronization Error	Synchronization failed.

Applications SHOULD handle these errors through the common SDK error model.

18.11 Best Practices

Applications SHOULD:

- treat protected assets as immutable Vault resources;
- preserve returned CEP references;
- synchronize periodically when connectivity is available;
- avoid modifying protected assets outside documented SDK APIs;
- allow the SDK to manage continuity preservation.

18.12 Relationship to the Protocol Specification

The Vault Profile combines several protocol operations defined by the QVTM Protocol Specification, including Authentication, Synchronization, Recovery, Continuity Evidence generation, and CEP preservation.

The SDK exposes these capabilities as a cohesive profile while hiding protocol sequencing and implementation details.

19. Secure Messaging Profile

19.1 Overview

The **Secure Messaging Profile** adapts the Core SDK for applications that require continuity-aware messaging.

The profile enables applications to preserve identity continuity, conversation continuity, and long-term communication integrity while remaining independent of any specific messaging protocol or transport.

The Secure Messaging Profile complements existing secure messaging technologies.

It does not replace message transport, delivery, or application-specific messaging features.

19.2 Responsibilities

The Secure Messaging Profile is responsible for:

- preserving conversation continuity;
- preserving identity continuity;
- supporting secure message exchange;
- generating continuity information for messaging operations when required;
- supporting recovery of messaging continuity.

The Secure Messaging Profile is not responsible for:

- message routing;
- message delivery guarantees;
- transport protocols;
- notification services;
- application-specific conversation management.

19.3 Supported Operations

The Secure Messaging Profile typically supports:

Operation	Supported
Enrollment	Yes
Authentication	Yes
Synchronization	Yes
Recovery	Yes
Vault Operations	Optional

Individual SDK implementations MAY expose additional messaging capabilities.

19.4 Typical Workflow

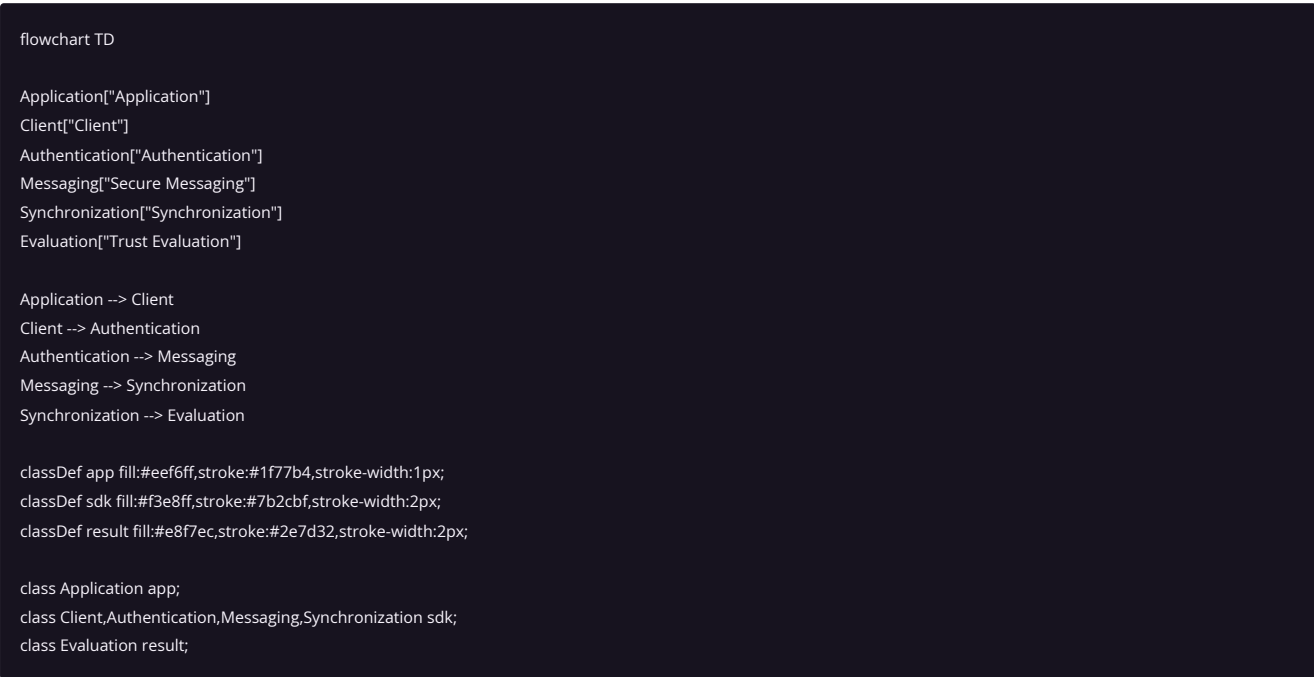
A typical Secure Messaging integration follows these stages:

1. Initialize the SDK.
2. Create a Client.
3. Authenticate the active Identity.
4. Establish a messaging session.
5. Exchange protected messages.
6. Synchronize continuity information when appropriate.
7. Recover messaging continuity if required.

The SDK coordinates protocol interactions throughout the messaging lifecycle.

19.5 Observable Workflow

Figure 14. Secure Messaging Workflow



The messaging application interacts only with the SDK.

The SDK manages the continuity aspects of the communication.

19.6 Conversation Continuity

The Secure Messaging Profile preserves continuity throughout the lifetime of a conversation.

Conversation continuity may span:

- multiple Sessions;
- multiple devices;
- cryptographic migrations;
- recovery events;
- synchronization events.

Applications SHOULD treat conversations as continuity-aware resources.

19.7 Generated Objects

Messaging operations MAY generate:

- Continuity Evidence;
- CEPs;
- Trust Evaluation;
- conversation-related metadata;
- profile-specific messaging information.

Applications SHOULD preserve returned SDK objects whenever appropriate.

19.8 Synchronization

Messaging continuity MAY be synchronized across compatible participants.

Synchronization exchanges eligible continuity information without modifying previously established protocol history.

Applications SHOULD rely on the SDK to determine synchronization timing and scope.

19.9 Error Model

The Secure Messaging Profile MAY produce all Core SDK errors together with messaging-specific failures.

Typical additional errors include:

Error Category	Typical Cause
Message Delivery Error	Protected message could not be processed.
Conversation Error	Conversation continuity unavailable.
Synchronization Error	Messaging synchronization failed.
Recovery Error	Messaging continuity recovery failed.

Applications SHOULD process these errors using the common SDK error model.

19.10 Best Practices

Applications SHOULD:

- authenticate participants before protected messaging;
- synchronize conversation continuity regularly;
- preserve Trust Evaluation results when auditing conversations;
- avoid bypassing documented SDK messaging APIs;

- allow the SDK to manage continuity artifacts.

19.11 Relationship to the Protocol Specification

The Secure Messaging Profile composes Authentication, Synchronization, Recovery, Continuity Evidence generation, CEP management, and Trust Evaluation as defined by the QVTM Protocol Specification.

The SDK intentionally abstracts protocol sequencing while exposing a stable messaging integration surface.

20. Secure Upload Profile

20.1 Overview

The **Secure Upload Profile** adapts the Core SDK for applications that exchange protected digital assets between trusted participants.

Unlike the Vault Profile, which focuses on long-term preservation, the Secure Upload Profile focuses on the protected transfer of digital assets while preserving Trust Continuity throughout the transfer lifecycle.

The profile is independent of any specific storage provider, cloud service, or transfer protocol.

Applications continue to interact with the same Core SDK objects and operations.

20.2 Responsibilities

The Secure Upload Profile is responsible for:

- protecting digital assets during transfer;
- preserving sender continuity;
- preserving recipient continuity;
- supporting continuity-aware asset exchange;
- generating continuity information when required by the protocol.

The Secure Upload Profile is not responsible for:

- cloud storage implementation;
- transport protocols;
- content delivery;
- access control;
- application-specific sharing policies.

20.3 Supported Operations

The Secure Upload Profile typically supports:

Operation	Supported
Enrollment	Yes
Authentication	Yes
Synchronization	Yes
Recovery	Yes
Vault Operations	Optional

Individual SDK implementations MAY expose additional upload capabilities.

20.4 Typical Workflow

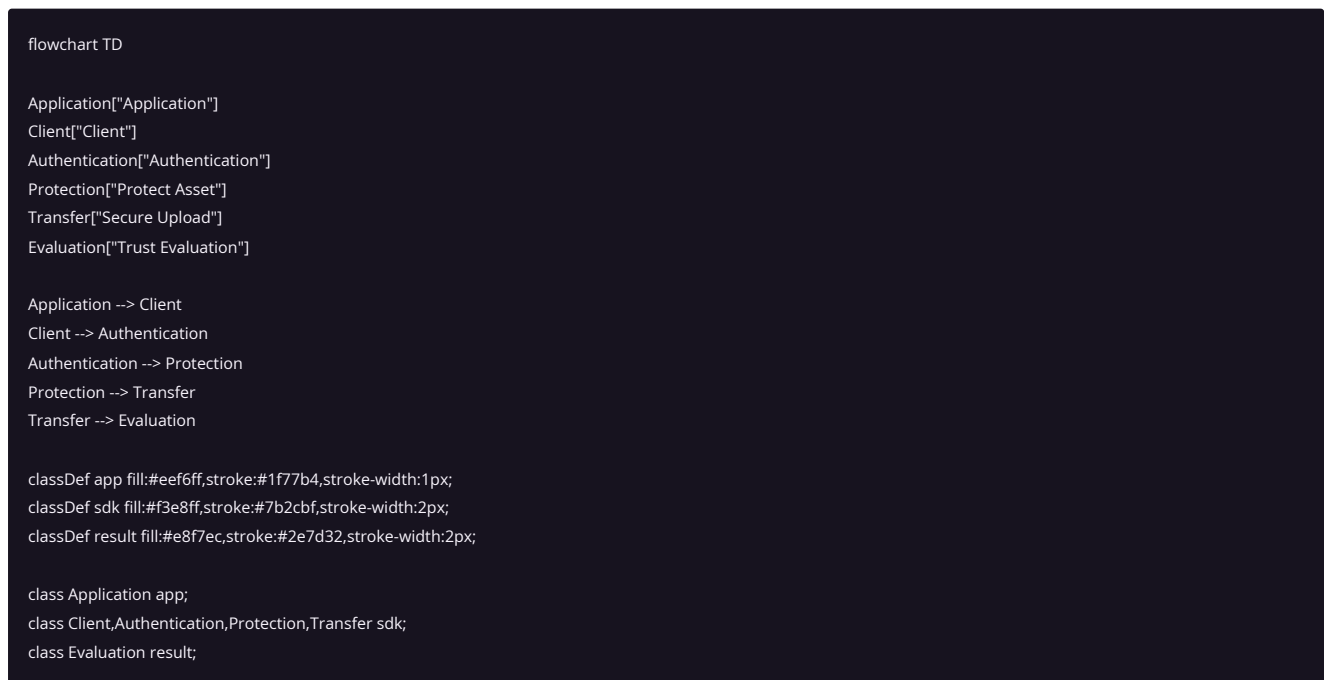
A typical Secure Upload integration follows these stages:

1. Initialize the SDK.
2. Create a Client.
3. Authenticate the active Identity.
4. Protect the digital asset for transfer.
5. Transfer the protected asset.
6. Verify recipient continuity when required.
7. Synchronize generated continuity information.

The SDK coordinates all continuity-related protocol operations.

20.5 Observable Workflow

Figure 15. Secure Upload Workflow



The Secure Upload Profile preserves continuity throughout the transfer process while remaining independent of the underlying transport technology.

20.6 Protected Transfers

A protected transfer MAY include any application-defined digital asset.

Typical examples include:

- documents;
- media files;
- archives;
- credentials;
- configuration data;
- application-specific protected objects.

The SDK intentionally remains independent of content type.

20.7 Recipient Continuity

The Secure Upload Profile enables applications to verify recipient continuity before or during protected transfer when required by the selected Deployment Profile.

Applications consume only the observable SDK result.

Recipient evaluation remains governed by the Protocol Specification.

20.8 Generated Objects

Secure Upload operations MAY generate:

- protected transfer objects;
- Continuity Evidence;
- CEPs;
- Trust Evaluation;
- transfer-related metadata.

Applications SHOULD preserve returned SDK objects whenever appropriate.

20.9 Synchronization

Transfer-related continuity information MAY later participate in Synchronization.

Synchronization exchanges eligible continuity information without modifying historical protocol state.

Applications SHOULD invoke Synchronization only through documented SDK APIs.

20.10 Error Model

The Secure Upload Profile MAY produce all Core SDK errors together with transfer-specific failures.

Typical additional errors include:

Error Category	Typical Cause
Protection Error	Asset could not be prepared for transfer.
Transfer Error	Protected transfer failed.
Recipient Error	Recipient continuity requirements not satisfied.
Synchronization Error	Transfer continuity could not be synchronized.
Recovery Error	Protected transfer context could not be recovered.

Applications SHOULD process these errors using the common SDK error model.

20.11 Best Practices

Applications SHOULD:

- authenticate participants before initiating protected transfers;
- preserve returned Trust Evaluation results when required for auditing;
- synchronize continuity information after completed transfers;
- avoid implementing custom continuity validation;
- rely exclusively on documented SDK APIs.

Applications SHOULD treat transfer continuity independently from application-specific file sharing policies.

20.12 Relationship to the Protocol Specification

The Secure Upload Profile combines Authentication, Synchronization, Recovery, Continuity Evidence generation, CEP management, and Trust Evaluation as defined by the QVTM Protocol Specification.

The SDK intentionally abstracts protocol coordination while providing a stable integration model for protected digital asset exchange.

Part V — Integration

Part V describes how applications integrate the QVTM SDK into their operational lifecycle.

Where the Core SDK defines the public object model and the Operations layer defines protocol interactions, this section explains how those elements are combined within a production application.

The guidance presented in this chapter is implementation-independent.

Individual SDK implementations MAY provide additional language-specific recommendations without modifying the observable behavior defined here.

21. SDK Lifecycle

21.1 Overview

Every QVTM application follows a common SDK lifecycle.

Although Deployment Profiles provide different capabilities, they all share the same high-level lifecycle:

1. SDK initialization.
2. Client creation.
3. Identity selection.
4. Session creation.
5. Protocol operations.
6. Session termination.
7. SDK shutdown.

Applications SHOULD follow this lifecycle whenever practical.

21.2 Lifecycle Phases

The SDK lifecycle consists of five observable phases.

Phase	Description
Initialization	Configure and initialize the SDK.
Operational	Execute protocol operations.
Maintenance	Perform synchronization and recovery when required.
Shutdown	Release SDK resources.
Restart	Resume operation using the preserved continuity state.

Each phase is independent of the selected Deployment Profile.

21.3 Initialization

Initialization prepares the SDK for operation.

Typical initialization activities include:

- loading SDK configuration;
- selecting a Deployment Profile;
- creating a Client;

- preparing runtime resources;
- validating SDK compatibility.

Applications SHOULD initialize the SDK once during application startup.

21.4 Operational Phase

Once initialized, the application performs protocol operations through one or more Sessions.

Typical operational activities include:

- Enrollment;
- Authentication;
- Vault Operations;
- Secure Messaging;
- Secure Upload;
- Synchronization.

Applications SHOULD reuse the same Client throughout this phase.

21.5 Maintenance Phase

Some applications require periodic maintenance.

Typical maintenance activities include:

- Synchronization;
- Recovery;
- SDK updates;
- compatibility verification;
- continuity preservation.

Maintenance activities SHOULD use the same public SDK APIs as operational workflows.

21.6 Shutdown

Applications SHOULD gracefully terminate the SDK before exiting.

Shutdown typically includes:

- closing active Sessions;
- releasing SDK resources;
- completing pending operations;
- flushing implementation-specific state.

Shutdown does not invalidate Identities, CEPs, or Continuity Evidence.

21.7 Restart

Applications MAY restart at any time.

Previously preserved continuity information remains available according to the active Deployment Profile.

The SDK restores operational capability using observable SDK objects rather than application-managed protocol state.

21.8 Observable Lifecycle

Figure 16. SDK Lifecycle

flowchart LR

Initialize["Initialize SDK"]

Client["Create Client"]

Operate["Execute Operations"]

Maintain["Synchronize / Recover"]

Shutdown["Shutdown SDK"]

Initialize --> Client

Client --> Operate

Operate --> Maintain

Maintain --> Shutdown

classDef sdk fill:#f3e8ff,stroke:#7b2cbf,stroke-width:2px;

class Initialize,Client,Operate,Maintain,Shutdown sdk;

The lifecycle illustrates the canonical integration pattern used by every Deployment Profile.

21.9 Best Practices

Applications SHOULD:

- initialize the SDK once;
- reuse a single Client whenever possible;
- create short-lived Sessions;
- synchronize periodically when appropriate;
- perform graceful shutdown.

Following these recommendations simplifies lifecycle management while maximizing compatibility across SDK implementations.

21.10 Relationship to the Protocol Specification

The SDK lifecycle is an application integration model.

The Protocol Specification defines protocol behavior.

The SDK coordinates protocol execution while exposing a stable lifecycle suitable for application development.

22. Error Handling

22.1 Overview

The QVTM SDK exposes a consistent and implementation-independent error model across all Deployment Profiles.

The purpose of the error model is to allow applications to respond predictably to observable SDK failures without requiring knowledge of internal protocol processing.

Errors describe observable conditions only.

They do not expose implementation-specific protocol state.

22.2 Error Principles

Every SDK implementation SHOULD follow these principles:

- errors are deterministic;
- errors are implementation-independent;
- errors describe observable behavior;
- errors do not expose sensitive internal information;
- errors are stable across SDK versions whenever practical.

Applications SHOULD rely only on documented error categories.

22.3 Error Categories

The SDK groups errors into high-level categories.

Category	Description
Configuration Error	SDK configuration is invalid or incomplete.
Initialization Error	SDK initialization failed.
Session Error	Session lifecycle could not continue.
Identity Error	Identity-related operation failed.
Enrollment Error	Enrollment could not complete.
Authentication Error	Authentication failed.
Synchronization Error	Synchronization failed.
Recovery Error	Recovery failed.
Vault Error	Vault operation failed.
Compatibility Error	SDK compatibility requirements were not satisfied.
Runtime Error	Unexpected runtime condition occurred.

Detailed error codes are defined in Appendix A.

22.4 Observable Error Contract

The language-neutral SDK error contract is:

SDK implementations MAY expose additional language-specific information.

Observable behavior SHOULD remain semantically equivalent.

22.5 Recoverable Errors

Some errors allow normal operation to continue after corrective action.

Typical recoverable errors include:

- temporary connectivity loss;
- unavailable synchronization endpoint;
- expired execution context;
- temporary resource unavailability.

Applications SHOULD retry operations only when appropriate.

Retry behavior depends on the application.

22.6 Non-Recoverable Errors

Certain errors require termination of the current operation.

Typical examples include:

- invalid SDK configuration;
- unsupported protocol version;
- incompatible Deployment Profile;

- corrupted SDK installation.

Applications SHOULD terminate the affected workflow before attempting further operations.

22.7 Error Propagation

Errors propagate through the public SDK APIs.

Applications SHOULD process errors at the boundary between the SDK and the application.

The SDK intentionally avoids exposing implementation-specific exception hierarchies.

22.8 Error Reporting

Applications MAY record observable SDK errors for operational diagnostics.

Typical recorded information includes:

- error category;
- error code;
- operation name;
- timestamp;
- Deployment Profile.

Applications SHOULD avoid recording sensitive application data together with SDK errors.

22.9 Retry Strategy

The SDK intentionally does not prescribe retry policies.

Applications SHOULD determine retry behavior according to:

- business requirements;
- user experience;
- Deployment Profile;
- operational environment.

Repeated retries SHOULD NOT replace proper error handling.

22.10 User Feedback

Applications SHOULD translate SDK errors into user-appropriate messages.

SDK error messages are intended primarily for developers and operational diagnostics.

User-facing messages SHOULD:

- avoid implementation details;
 - avoid protocol terminology when unnecessary;
 - describe corrective actions whenever possible.
-

22.11 Best Practices

Applications SHOULD:

- process all SDK errors explicitly;
- distinguish recoverable from non-recoverable failures;
- preserve diagnostic information;
- avoid exposing internal SDK information to end users;
- rely exclusively on documented error categories.

Applications SHOULD NOT infer protocol state from individual error conditions.

22.12 Relationship to the Protocol Specification

The SDK error model describes observable integration behavior.

Protocol validation rules, message processing failures, and implementation-specific protocol diagnostics remain defined by the QVTM Protocol Specification and are intentionally abstracted by the SDK.

23. Version Compatibility

23.1 Overview

The QVTM SDK is designed to evolve while preserving stable application integration contracts.

Version compatibility enables SDK implementations, Deployment Profiles, and protocol versions to evolve independently whenever possible.

Applications SHOULD rely only on documented public SDK contracts.

Behavior outside the documented public API is not considered part of the compatibility guarantee.

23.2 Compatibility Principles

Every SDK implementation SHOULD follow these principles:

- preserve public API stability whenever practical;
- maintain observable behavior across compatible versions;
- isolate protocol evolution from application code;
- preserve Deployment Profile semantics;
- provide predictable upgrade paths.

Compatibility applies to observable SDK behavior.

It does not guarantee implementation-specific internal behavior.

23.3 Compatibility Layers

The QVTM technical stack and Infinitrust product ecosystem consist of several independently versioned layers.

Layer	Compatibility Scope
SDK API	Public programming interface
SDK Runtime	SDK implementation
Deployment Profile	Profile-specific behavior
Protocol Specification	Observable protocol semantics
Application	Business logic and integration

Each layer evolves independently while preserving documented compatibility boundaries.

23.4 SDK Versioning

Every SDK implementation exposes its own version.

The SDK version identifies:

- the public API surface;
- supported Deployment Profiles;

- supported Protocol Specification versions;
- implementation capabilities.

Applications MAY inspect SDK version information using public SDK APIs.

23.5 Protocol Compatibility

SDK implementations support one or more versions of the QVTM Protocol Specification.

Applications SHOULD avoid assuming protocol compatibility beyond what is explicitly reported by the SDK.

Protocol negotiation remains an implementation responsibility.

The SDK exposes only the observable compatibility result.

23.6 Deployment Profile Compatibility

Deployment Profiles evolve independently.

A newer SDK MAY:

- introduce additional profiles;
- extend existing profiles;
- introduce optional capabilities.

Existing profile semantics SHOULD remain backward compatible whenever practical.

Applications SHOULD verify profile availability before invoking profile-specific operations.

23.7 Forward Compatibility

Applications SHOULD tolerate:

- additional optional metadata;
- additional observable result fields;
- additional error categories;
- additional SDK capabilities.

Ignoring unknown optional information improves forward compatibility.

23.8 Backward Compatibility

SDK implementations SHOULD preserve:

- documented public APIs;
- observable object behavior;
- existing operation semantics;
- documented lifecycle behavior.

When breaking changes become unavoidable, SDK implementations SHOULD introduce a new major version.

23.9 Capability Discovery

Applications SHOULD determine supported functionality through capability discovery rather than version comparisons whenever possible.

Conceptually, capability discovery follows the pattern:

Capability-based integration is generally more resilient than explicit version checks.

23.10 Compatibility Errors

Compatibility verification MAY produce errors including:

Error Category	Typical Cause
Unsupported SDK Version	Application requires a newer SDK.
Unsupported Protocol Version	Protocol versions are incompatible.
Unsupported Deployment Profile	Required profile unavailable.
Unsupported Capability	Requested feature not implemented.
Compatibility Error	General compatibility failure.

Detailed error codes are defined in Appendix A.

23.11 Best Practices

Applications SHOULD:

- rely on capability discovery whenever possible;
- avoid hardcoding SDK version numbers;
- verify Deployment Profile availability before use;
- gracefully disable unsupported optional features;
- follow documented upgrade recommendations.

Applications SHOULD NOT depend on undocumented implementation behavior.

23.12 Relationship to the Protocol Specification

The Protocol Specification defines protocol interoperability requirements.

The SDK defines application integration compatibility.

Together they provide a stable evolution model in which protocol implementations may evolve while preserving application-facing integration contracts.

24. Best Practices

24.1 Overview

This chapter summarizes the recommended integration practices for applications using the QVTM SDK.

These recommendations are independent of any particular Deployment Profile and apply to every SDK implementation unless explicitly stated otherwise.

Following these practices improves portability, interoperability, maintainability, and long-term compatibility.

24.2 Use the SDK as the Integration Boundary

Applications SHOULD treat the SDK as the exclusive integration boundary.

Applications SHOULD NOT:

- implement protocol operations directly;
- recreate protocol state;
- manipulate Continuity Evidence;
- construct CEPs manually;
- infer internal protocol behavior.

All protocol interactions SHOULD occur through documented SDK APIs.

24.3 Keep Business Logic Separate

The SDK provides continuity services.

Applications provide business behavior.

Applications SHOULD keep:

- authorization;
- business rules;
- user experience;
- application workflows;
- persistence policies;

outside the SDK.

This separation simplifies future SDK upgrades.

24.4 Reuse the Client

Applications SHOULD create a single long-lived Client whenever practical.

A reusable Client:

- reduces initialization overhead;
- simplifies lifecycle management;
- improves operational consistency;
- minimizes resource consumption.

Applications SHOULD avoid repeatedly creating and destroying Clients during normal operation.

24.5 Keep Sessions Short-Lived

Sessions represent transient execution contexts.

Applications SHOULD:

- create Sessions when required;
- complete the intended operations;
- close Sessions promptly.

Applications SHOULD NOT use a single Session as permanent application state.

24.6 Preserve SDK Objects

Applications SHOULD preserve SDK-managed objects returned by public operations.

Typical examples include:

- Identity references;
- Continuity Evidence;
- CEP references;
- Trust Evaluation results.

Applications SHOULD avoid reconstructing these objects independently.

24.7 Prefer Capability Discovery

Applications SHOULD discover available functionality dynamically rather than assuming feature availability.

Capability discovery improves compatibility across:

- SDK versions;
- Deployment Profiles;
- implementation languages;
- future protocol evolution.

Whenever practical, applications SHOULD query capabilities instead of relying on version numbers.

24.8 Handle Errors Explicitly

Applications SHOULD process every SDK error explicitly.

Typical handling strategies include:

- retrying recoverable operations;
- requesting user interaction;
- selecting alternative workflows;
- recording operational diagnostics;
- terminating unrecoverable operations.

Ignoring SDK errors SHOULD be avoided.

24.9 Synchronize Opportunistically

Applications supporting disconnected operation SHOULD synchronize continuity information whenever suitable connectivity becomes available.

Synchronization SHOULD be treated as a routine maintenance activity rather than an exceptional event.

Applications SHOULD allow the SDK to determine the appropriate synchronization scope.

24.10 Preserve Compatibility

Applications SHOULD depend only on:

- documented public APIs;
- documented object behavior;
- documented Deployment Profiles;
- documented error categories.

Applications SHOULD NOT depend on:

- undocumented metadata;
 - internal protocol sequencing;
 - implementation-specific optimizations;
 - proprietary behavior.
-

24.11 Design for Recovery

Applications SHOULD assume that Recovery may occur during the operational lifetime of an Identity.

Recovery SHOULD be treated as a normal operational capability rather than an exceptional failure condition.

Applications SHOULD preserve the SDK objects required by the selected Deployment Profile to support future recovery operations.

24.12 Protect SDK Objects

Applications SHOULD treat SDK objects as integrity-sensitive resources.

Typical recommendations include:

- avoid unauthorized modification;
- avoid exposing internal identifiers unnecessarily;
- preserve returned metadata;
- protect persistent SDK-managed information.

The SDK remains responsible for protecting protocol-specific state.

24.13 Follow Deployment Profile Guidance

Each Deployment Profile introduces additional integration recommendations.

Applications SHOULD follow the guidance provided for the active profile rather than assuming identical operational behavior across all profiles.

The Core SDK remains consistent, while operational workflows may differ.

24.14 Summary

Applications that follow the practices described in this chapter obtain the greatest benefit from the QVTM SDK.

In particular, applications SHOULD:

- integrate exclusively through the public SDK;
- separate business policy from continuity services;
- reuse long-lived Clients;
- create short-lived Sessions;
- preserve SDK-managed objects;
- synchronize regularly;
- design for recovery;
- rely on capability discovery;
- avoid implementation-specific assumptions.

These recommendations maximize long-term compatibility while allowing both the SDK and the underlying protocol to evolve independently.

25. Performance Considerations

25.1 Overview

The QVTM SDK is designed to provide predictable operational behavior across a wide range of deployment environments.

Although performance characteristics vary between SDK implementations and Deployment Profiles, the public programming model is intended to remain stable and independent of implementation-specific optimizations.

Applications SHOULD optimize their own integration without relying on undocumented SDK behavior.

25.2 Performance Principles

The SDK is designed according to the following principles:

- predictable observable behavior;
- implementation independence;
- scalable object lifecycle management;
- efficient continuity preservation;
- compatibility across deployment environments.

Applications SHOULD consider these principles when designing integrations.

25.3 Client Reuse

Client initialization may involve implementation-specific preparation.

Applications SHOULD create a Client once and reuse it throughout the operational lifetime of the application whenever practical.

Repeated Client creation increases initialization overhead without improving protocol behavior.

25.4 Session Management

Sessions are lightweight execution contexts.

Applications SHOULD:

- create Sessions only when required;
- complete the intended operations;
- close Sessions promptly.

Keeping Sessions unnecessarily active may increase resource consumption.

25.5 Batch Operations

Applications that perform multiple protocol operations SHOULD minimize unnecessary lifecycle transitions.

For example, related operations MAY execute within the same Session when supported by the selected Deployment Profile.

Applications SHOULD avoid repeatedly opening and closing Sessions for tightly related operations.

25.6 Synchronization Frequency

Synchronization frequency depends on application requirements.

Applications SHOULD balance:

- continuity freshness;
- network availability;
- operational latency;
- resource consumption.

Disconnected applications SHOULD synchronize opportunistically rather than continuously.

25.7 Object Preservation

SDK-managed objects are intended to be reused.

Applications SHOULD preserve references to:

- Identity;
- Continuity Evidence;
- CEP;
- Trust Evaluation results.

Repeated reconstruction of equivalent application state SHOULD be avoided whenever possible.

25.8 Concurrency

Individual SDK implementations MAY support concurrent execution.

Applications SHOULD assume only the concurrency guarantees explicitly documented by the selected SDK implementation.

When multiple Sessions execute concurrently, applications SHOULD avoid introducing dependencies on execution order unless explicitly documented.

25.9 Resource Management

Applications SHOULD release SDK resources when they are no longer required.

Typical recommendations include:

- close completed Sessions;
- perform graceful SDK shutdown;
- avoid retaining obsolete SDK objects;
- allow the SDK to reclaim temporary resources.

Long-lived objects such as Identities and CEP references SHOULD be preserved according to application requirements.

25.10 Logging and Diagnostics

Operational diagnostics are valuable during development and production.

Applications SHOULD configure logging according to their operational requirements.

Production deployments SHOULD avoid excessive diagnostic output that could negatively affect performance.

The SDK intentionally leaves logging policy to the application.

25.11 Scalability

The Core SDK object model is designed to support applications ranging from single-device deployments to large distributed environments.

Scalability is achieved through:

- reusable Clients;
- independent Sessions;
- immutable Continuity Evidence;
- immutable CEPs;
- implementation-independent object contracts.

Applications SHOULD rely on these public abstractions rather than implementation-specific optimizations.

25.12 Deployment Considerations

Different Deployment Profiles may emphasize different operational characteristics.

For example:

Deployment Profile	Typical Optimization Goal
Login	Low authentication latency
Vault	Efficient protection of large digital assets
Secure Messaging	Sustained conversation continuity
Secure Upload	Efficient protected asset transfer

Applications SHOULD optimize for their operational profile rather than attempting universal optimization strategies.

25.13 Performance Best Practices

Applications SHOULD:

- initialize the SDK only once;
- reuse long-lived Clients;

- create short-lived Sessions;
- preserve SDK-managed objects;
- synchronize only when appropriate;
- release unused SDK resources;
- avoid unnecessary protocol operations;
- rely on documented public APIs rather than implementation-specific behavior.

Following these recommendations provides consistent performance while preserving long-term compatibility.

25.14 Relationship to the Protocol Specification

Performance characteristics are implementation-dependent.

The QVTM Protocol Specification defines protocol semantics, not implementation performance.

The SDK preserves the observable protocol behavior while allowing individual implementations to optimize execution internally without affecting application compatibility.

Part VI — Examples

The examples presented in this part illustrate the canonical use of the QVTM SDK.

Examples in this part use published `operationId` values, released package names, or validated JSON contracts. The document does not use language-neutral pseudocode. Where the supplied artifact bundle lacks the authoritative OpenAPI schema or generated type, the example is limited to the verified operation inventory and does not fabricate a body.

26. Basic Authentication

26.1 Overview

This example illustrates the minimal sequence required to authenticate an existing Identity using the Login Profile.

The objective is to demonstrate the interaction between the application and the SDK without exposing protocol internals.

26.2 Scenario

The application needs to authenticate a previously enrolled user before granting access to protected functionality.

The application delegates continuity verification to the SDK and applies its own authorization policy using the returned Trust Evaluation.

26.3 Integration Workflow

Figure 17. Basic Authentication Example

sequenceDiagram

participant App as Application
participant Client as Client
participant Session as Session
participant SDK as QVTM SDK

App->>Client: createSession()
Client->>Session: Session

App->>Session: authenticate()

Session->>SDK: Execute Authentication

SDK-->>Session: Authentication Result
SDK-->>Session: Trust Evaluation

Session-->>App: AuthenticationResult
Session-->>App: TrustEvaluation

26.4 Published API Sequence

Step	Published operation	HTTP contract
Start login	startLoginSession	POST /api/v1/admin/control-plane/login/start
Issue challenge	issueAuthChallenge	POST /api/v1/admin/control-plane/auth/challenges
Read status	getAuthChallengeStatus	GET /api/v1/admin/control-plane/auth/challenges/{challenge_id}/status
Verify approval	verifyAuthChallenge	POST /api/v1/admin/control-plane/auth/challenges/{challenge_id}/verify
Consume session handle	consumeAuthChallengeSessionHandle	POST /api/v1/admin/control-plane/auth/challenges/{challenge_id}/session-handle
Verify trust evidence	verifyTrustEvidence	POST /api/v1/trust/evidence/verify

The corresponding OpenAPI schemas are authoritative for every request and response body.

The SDK authenticates the Identity.

The application decides whether access should be granted.

26.5 Observable Result

A successful authentication typically returns:

- Authentication Result;
- Identity reference;
- Trust Evaluation;
- Continuity Evidence when required;
- CEP references when required.

The exact observable objects depend on the active Deployment Profile.

26.6 Error Handling

Typical observable failures include:

- Authentication Error;
- Identity Error;
- Session Error;

- Runtime Error.

Applications SHOULD process these errors using the common SDK error model described in Chapter 22.

26.7 Discussion

This example illustrates one of the central design principles of the QVTM SDK.

The SDK verifies continuity.

The application determines authorization.

This separation allows applications to implement any authorization policy while remaining independent of protocol evolution.

26.8 Related Chapters

This example builds upon:

- Chapter 6 — Client
- Chapter 7 — Session
- Chapter 11 — Trust Evaluation
- Chapter 13 — Authentication
- Chapter 17 — Login Profile

The underlying protocol behavior is defined by the QVTM Protocol Specification.

27. File Protection

27.1 Overview

This example illustrates the canonical workflow for protecting a digital asset using the Vault Profile.

The objective is to demonstrate how an application delegates asset protection to the SDK while remaining independent of the underlying protocol and cryptographic implementation.

27.2 Scenario

An application needs to protect a document before storing or transferring it.

The application provides the asset to the SDK.

The SDK performs the required protocol operations and returns the protected asset together with any observable continuity objects generated during the operation.

27.3 Integration Workflow

Figure 18. File Protection Example

```
sequenceDiagram
    participant App as Application
    participant Client as Client
    participant Session as Session
    participant SDK as QVTM SDK

    App->>Client: createSession()
    Client->>Session: Session

    App->>Session: protect(asset)

    Session->>SDK: Execute Vault Operation

    SDK-->>Session: Protected Asset
    SDK-->>Session: Continuity Evidence
    SDK-->>Session: CEP

    Session-->>App: ProtectionResult
```

27.4 Published Surface

The supplied public catalog does not publish Vault HTTP operations or a Vault OpenAPI schema. Consequently, this RC documents the Vault behavioral contract in Chapters 16 and 18 but does not invent callable methods or JSON bodies. Concrete Vault examples must be added from the released Vault SDK or its OpenAPI source before freeze.

The application delegates asset protection to the SDK.

The SDK coordinates all continuity-related protocol operations.

27.5 Observable Result

A successful protection operation MAY return:

- protected asset;
- Continuity Evidence;
- CEP references;
- Trust Evaluation when required;
- profile-specific metadata.

Applications SHOULD preserve the returned SDK objects according to the selected Deployment Profile.

27.6 Error Handling

Typical observable failures include:

- Protection Error;
- Identity Error;
- Session Error;
- Runtime Error.

Applications SHOULD process these errors using the common SDK error model.

27.7 Discussion

The protected asset becomes part of the application's protected data lifecycle.

The SDK preserves continuity independently of the application's storage implementation.

Applications remain responsible for storage location, retention policy, and access control.

27.8 Related Chapters

This example builds upon:

- Chapter 6 — Client
- Chapter 7 — Session
- Chapter 16 — Vault Operations
- Chapter 18 — Vault Profile

The underlying protection semantics are defined by the QVTM Protocol Specification.

28. Secure Messaging

28.1 Overview

This example illustrates how an application integrates the Secure Messaging Profile to exchange protected messages while preserving Trust Continuity.

The SDK provides continuity-aware messaging services.

The application remains responsible for conversation management, message presentation, and delivery infrastructure.

28.2 Scenario

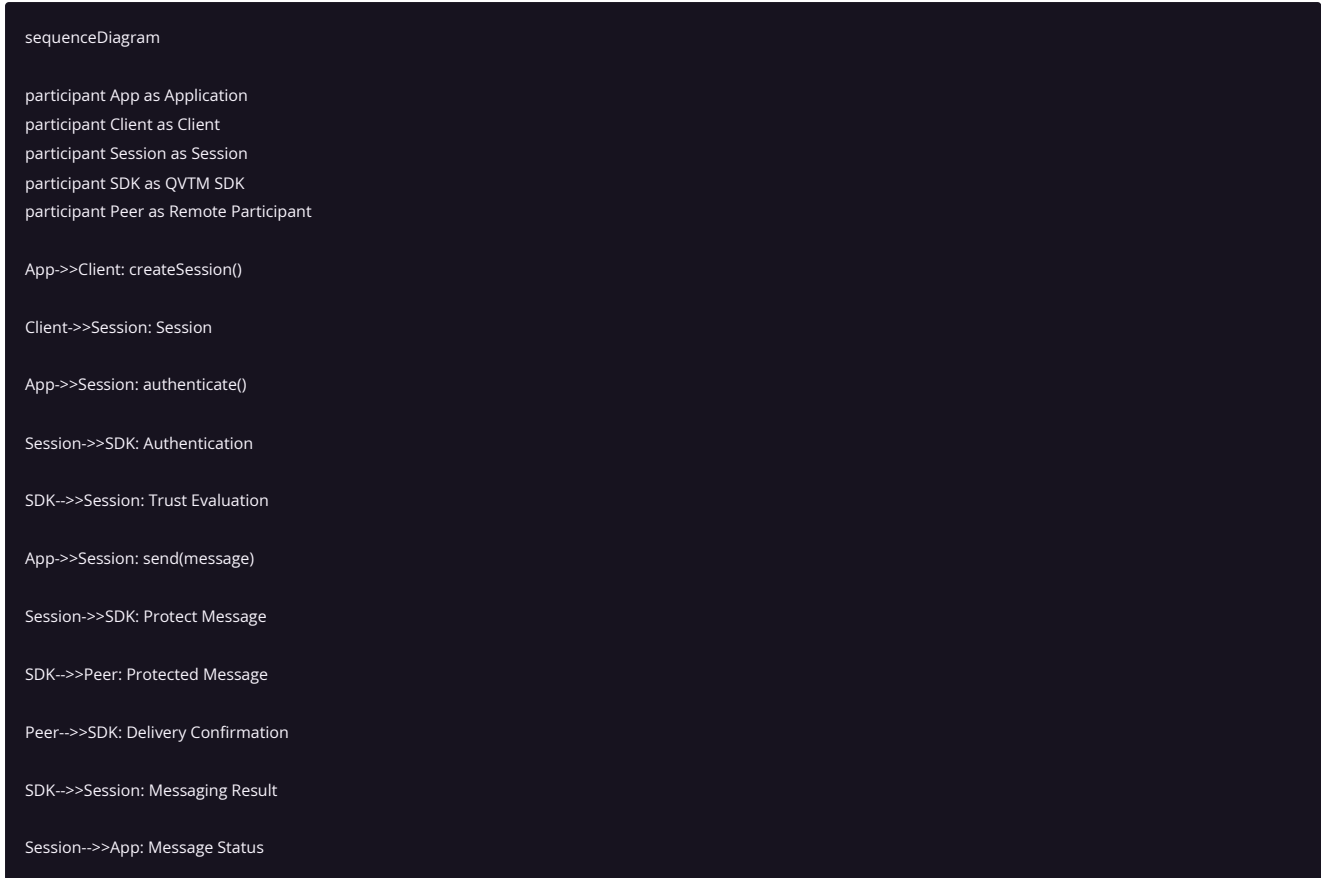
Two previously enrolled participants exchange protected messages.

The application authenticates the active Identity, establishes a messaging session, delegates message protection to the SDK, and consumes the observable results returned by the SDK.

The transport mechanism is outside the scope of the SDK.

28.3 Integration Workflow

Figure 19. Secure Messaging Example



The SDK coordinates continuity-aware messaging.

The application coordinates the conversation.

28.4 Published API Sequence

Step	Published operation	HTTP contract
Register device	registerRuntimeDevice	POST /devices
Start handshake	startRuntimeHandshake	POST /handshake/start
Finish handshake	finishRuntimeHandshake	POST /handshake/finish
Send protected message	sendRuntimeMessage	POST /messages/send
Read transparency state	getRuntimeTransparencyState	GET /transparency/{device_id}/state

The released runtime OpenAPI schema is authoritative for request and response JSON.

The SDK manages continuity throughout the messaging workflow.

The application manages the conversation state.

28.5 Observable Result

A successful messaging operation MAY return:

- message status;
- Trust Evaluation;
- Continuity Evidence;
- CEP references;
- messaging metadata.

The returned objects depend on the active Deployment Profile.

28.6 Error Handling

Typical observable failures include:

- Authentication Error;
- Messaging Error;
- Synchronization Error;
- Recovery Error;
- Runtime Error.

Applications SHOULD process these failures using the common SDK error model.

28.7 Discussion

The Secure Messaging Profile intentionally separates messaging continuity from transport infrastructure.

The SDK does not prescribe:

- transport protocols;
- message queues;
- notification services;
- conversation storage.

These remain application responsibilities.

The SDK contributes continuity-aware protection while remaining transport independent.

28.8 Related Chapters

This example builds upon:

- Chapter 6 — Client
- Chapter 7 — Session
- Chapter 13 — Authentication
- Chapter 19 — Secure Messaging Profile

The continuity semantics governing messaging are defined by the QVTM Protocol Specification.

29. Synchronization

29.1 Overview

This example illustrates the canonical workflow for synchronizing continuity information between compatible QVTM participants.

Synchronization allows an application to exchange eligible continuity information without exposing protocol internals or modifying previously established continuity history.

The SDK coordinates the synchronization process while exposing only observable results.

29.2 Scenario

An application has accumulated new continuity information while operating independently.

When communication with a compatible participant becomes available, the application initiates a synchronization operation through the SDK.

The SDK determines which information is eligible for exchange.

29.3 Integration Workflow

Figure 20. Synchronization Example

```
sequenceDiagram
    participant App as Application
    participant Client as Client
    participant Session as Session
    participant SDK as QTM SDK
    participant Peer as Remote Participant

    App->>Client: createSession()
    Client->>Session: Session
    App->>Session: synchronize()
    Session->>SDK: Synchronization Request
    SDK->>Peer: Exchange Eligible CEPs
    Peer-->>SDK: Synchronization Response
    SDK-->>Session: Synchronization Result
    Session-->>App: Updated Continuity Context
```

The application initiates synchronization.

The SDK determines how synchronization is performed.

29.4 Published Surface

The current public catalog exposes runtime transparency history and trust artifact retrieval, but it does not publish a dedicated synchronization endpoint. Synchronization remains a protocol and SDK contract in Chapters 14 and 23. This document does not manufacture an HTTP mapping that is absent from the released catalog.

Applications interact only with the observable synchronization result.

The SDK manages protocol-specific processing.

29.5 Observable Result

A successful synchronization operation MAY return:

- synchronization status;
- accepted Continuity Evidence;
- accepted CEP references;
- updated Trust Evaluation when required;
- profile-specific synchronization metadata.

Applications SHOULD preserve any returned SDK-managed objects.

29.6 Error Handling

Typical observable failures include:

- Synchronization Error;
- Compatibility Error;
- Connectivity Error;
- Runtime Error.

Applications SHOULD process these failures using the common SDK error model.

29.7 Discussion

Synchronization is an incremental maintenance operation.

Applications SHOULD NOT assume that every local object participates in every synchronization cycle.

The SDK determines synchronization scope according to:

- the active Deployment Profile;
- protocol requirements;
- observable continuity context.

Applications SHOULD treat synchronization as a routine operational activity rather than an exceptional event.

29.8 Related Chapters

This example builds upon:

- Chapter 6 — Client
- Chapter 7 — Session
- Chapter 10 — Continuity Evidence Package (CEP)
- Chapter 14 — Synchronization

The synchronization semantics are defined by the QVTM Protocol Specification.

30. Recovery

30.1 Overview

This example illustrates the canonical workflow for restoring operational capability using the QVTM SDK while preserving the continuity established by previous protocol operations.

Recovery extends an existing continuity history.

It does not establish a new identity relationship.

Applications interact exclusively with the public SDK.

The SDK coordinates all recovery-specific protocol behavior.

30.2 Scenario

An operational Identity can no longer continue normal operation because the application is executing in a new operational environment.

The application invokes the Recovery operation using the appropriate recovery context supported by the selected Deployment Profile.

The SDK restores operational capability and returns the observable results.

30.3 Integration Workflow

Figure 21. Recovery Example

```
sequenceDiagram
    participant App as Application
    participant Client as Client
    participant Session as Session
    participant SDK as QTM SDK

    App->>Client: createSession()

    Client->>Session: Session

    App->>Session: recover(recoveryContext)

    Session->>SDK: Execute Recovery

    SDK-->>Session: Recovery Result
    SDK-->>Session: Identity
    SDK-->>Session: Trust Evaluation
    SDK-->>Session: Continuity Evidence

    Session-->>App: RecoveryResult
```

The SDK restores operational capability while preserving observable continuity.

30.4 Published Surface

The supplied public catalog does not expose a dedicated Recovery HTTP operation or the corresponding request and result schemas. Recovery is therefore documented as a protocol-backed SDK contract in Chapters 15 and 24. Binding-specific recovery calls and JSON must be imported from their released source before the document is frozen.

The application resumes normal operation only after successful Recovery.

30.5 Observable Result

A successful Recovery operation MAY return:

- Recovery Result;
- restored Identity;
- Trust Evaluation;
- Continuity Evidence;
- CEP references;
- profile-specific recovery metadata.

The exact observable objects depend on the selected Deployment Profile.

30.6 Error Handling

Typical observable failures include:

- Recovery Error;
- Recovery Context Error;
- Identity Error;
- Compatibility Error;
- Runtime Error.

Applications SHOULD process these failures using the common SDK error model described in Chapter 22.

30.7 Discussion

Recovery is a normal operational capability of the QVTM SDK.

Applications SHOULD design their lifecycle assuming that Recovery may occur at any point during the lifetime of an operational Identity.

Recovery preserves previously established continuity.

Historical Continuity Evidence remains unchanged.

New Continuity Evidence MAY be generated to represent the completed recovery operation.

30.8 Related Chapters

- This example builds upon:
- Chapter 6 — Client
 - Chapter 7 — Session
 - Chapter 8 — Identity
 - Chapter 9 — Continuity Evidence
 - Chapter 10 — Continuity Evidence Package (CEP)
 - Chapter 15 — Recovery

The recovery semantics are defined by the QVTM Protocol Specification.

Part VII — Published SDK API Reference

This part is derived from the public implementation surfaces supplied for this reference. It is the application-facing reference for the currently implemented TypeScript packages. The method names, request types, response types, and error classes shown here are implementation-derived rather than semantic placeholders.

JSON examples in this part are **implementation-derived examples of the published TypeScript types**. They illustrate the expected object shape. The TypeScript declaration remains authoritative for optionality and exact type constraints; the applicable OpenAPI document remains authoritative for HTTP serialization and status codes.

31. Published Package Inventory

Package	Documented version	Maturity	HTTP serialization	Public role
@qvtm/control-plane	0.1.0-sdk12	SDK boundary release	Yes	Tenant, login challenge, customer invite, package delivery, and device-trust administration.
@qvtm/trust-verifier	0.1.0-sdk12	SDK boundary release	Yes	Trust evidence inspection and verification, artifact package verification, and transparency export retrieval.
@qvtm/runtime-protocol	0.1.0-sdk12	SDK boundary release	Yes	Runtime device registration, handshake, protected message submission, and transparency state access.
@qvtm/vault	1.0.0	Public SDK release	No direct HTTP contract	Local Vault lifecycle, file references, device approval, synchronization, and event handling.

The supplied TypeScript package declarations are authoritative for public method names, parameters, return types, optionality, and binding-specific errors. Published OpenAPI descriptions remain authoritative for HTTP methods, paths, status codes, and wire serialization. The QVTM Protocol Specification remains authoritative for protocol semantics.

32. @qvtm/control-plane

Public methods

Method	Parameters	Returns
health	—	Promise<HealthResponse>
getRuntimeStatus	—	Promise<RuntimeStatusResponse>
runtimeStatus	—	Promise<RuntimeStatusResponse>
listAuthScopes	—	Promise<readonly string[]>
startLoginSession	body: StartLoginSessionRequest	Promise<LoginSessionResponse>
getLoginSessionStatus	loginSessionId: string	Promise<LoginSessionResponse>
issueAuthChallenge	body?: IssueAuthChallengeRequest	Promise<AuthChallengeResponse>
getAuthChallenge	challengeId: string	Promise<AuthChallengeResponse>
getAuthChallengeStatus	challengeId: string	Promise<AuthChallengeStatusResponse>
verifyAuthChallenge	challengeId: string, body: VerifyAuthChallengeRequest,	Promise<VerifyAuthChallengeResponse>
consumeAuthChallengeSessionHandle	challengeId: string, body?: SessionHandleConsumeRequest,	Promise<SessionHandleResponse>
consumeSessionHandle	challengeId: string, body?: SessionHandleConsumeRequest	Promise<SessionHandleResponse>
consumeAuthChallengeHandoff	challengeId: string, body?: SessionHandleConsumeRequest,	Promise<SessionHandleResponse>
getSessionDiagnostics	—	Promise<SessionDiagnosticsResponse>
revokeCurrentSession	—	Promise<AdminWriteResponse>
getChallengeCompatibilityPayload	challengeId: string	Promise<AuthChallengeResponse>
getChallengeCompatibilityStatus	challengeId: string	Promise<AuthChallengeStatusResponse>
verifyChallengeCompatibilityApproval	challengeId: string, body: VerifyAuthChallengeRequest,	Promise<VerifyAuthChallengeResponse>
consumeChallengeCompatibilitySessionHandle	challengeId: string, body?: SessionHandleConsumeRequest,	Promise<SessionHandleResponse>
consumeChallengeCompatibilityHandoff	challengeId: string, body?: SessionHandleConsumeRequest,	Promise<SessionHandleResponse>
listControlPlaneTenants	—	Promise<readonly TenantRecord[]>
upsertControlPlaneTenant	body: TenantUpsertRequest, idempotencyKey?: string	Promise<TenantRecord>
createTenant	body: TenantProvisioningRequest, idempotencyKey?: string	Promise<TenantRecord>
onboardControlPlaneTenant	body: TenantOnboardingRequest, idempotencyKey?: string,	Promise<TenantOnboardingResponse>
getControlPlaneTenant	tenantId: string	Promise<TenantRecord>
transitionControlPlaneTenantLifecycle	tenantId: string, body: TenantLifecycleTransitionRequest, idempotencyKey?: string,	Promise<AdminWriteResponse>
getControlPlaneTenantSummary	tenantId: string	Promise<TenantSummaryResponse>
listCustomerInvites	tenantId: string	Promise<readonly CustomerInviteResponse[]>

Method	Parameters	Returns
createCustomerInvite	tenantId: string, body: CustomerInviteCreateRequest, idempotencyKey?: string,	Promise<CustomerInviteResponse>
listPackageDeliveries	tenantId: string	Promise<readonly PackageDeliveryResponse[]>
createPackageDelivery	tenantId: string, body: PackageDeliveryCreateRequest, idempotencyKey?: string,	Promise<PackageDeliveryResponse>
revokePackageDelivery	tenantId: string, deliveryId: string, idempotencyKey?: string	Promise<AdminWriteResponse>
provisionTenantCustomerInvite	input: ProvisionTenantCustomerInviteWorkflowRequest,	Promise<ProvisionTenantCustomerInviteWorkflowResponse>
provisionTenantPackageDelivery	input: ProvisionTenantPackageDeliveryWorkflowRequest,	Promise<ProvisionTenantPackageDeliveryWorkflowResponse>
listDeviceTrustBundles	—	Promise<ListDeviceTrustBundlesResponse>
exportDeviceTrustBundleStatusList	—	Promise<DeviceTrustBundleStatusListResponse>
exportSignedDeviceTrustBundleStatusList	—	Promise<SignedDeviceTrustBundleStatusListResponse>
exportDeviceTrustBundleAuditTrail	—	Promise<DeviceTrustBundleAuditTrailResponse>
exportDeviceTrustBundleAuditTrailForBundle	bundleId: string	Promise<DeviceTrustBundleAuditTrailResponse>
exportDeviceTrustBundleEvidencePackage	bundleId: string	Promise<DeviceTrustBundleEvidencePackageResponse>
exportDeviceTrustBundleReadinessReport	bundleId: string	Promise<DeviceTrustBundleReadinessReportResponse>
exportDeviceTrustBundleDistributionPackage	bundleId: string	Promise<DeviceTrustBundleDistributionPackageResponse>
exportDeviceTrustBundleDistributionManifest	—	Promise<DeviceTrustBundleDistributionManifestResponse>
exportDeviceTrustDistributionClosureReport	—	Promise<DeviceTrustDistributionClosureReportResponse>
exportDeviceTrustPhase61FinalReadinessReport	—	Promise<DeviceTrustPhase61FinalReadinessReportResponse>
issueDeviceTrustBundle	body: IssueDeviceTrustBundleRequest, idempotencyKey?: string	Promise<DeviceTrustBundleResponse>
getDeviceTrustBundle	bundleId: string	Promise<DeviceTrustBundleResponse>
exportDeviceTrustBundle	bundleId: string	Promise<DeviceTrustBundleResponse>
revokeDeviceTrustBundle	bundleId: string, body: RevokeDeviceTrustBundleRequest, idempotencyKey?: string	Promise<DeviceTrustBundleResponse>
rotateDeviceTrustBundle	bundleId: string, body: RotateDeviceTrustBundleRequest, idempotencyKey?: string	Promise<DeviceTrustBundleResponse>
pollAuthChallengeStatus	challengeId: string, options?: AuthChallengePollOptions	Promise<AuthChallengePollResponse>

Public object contracts

QvtmControlPlaneClientOptions

Field	Type	Required
baseUrl	string	Yes
bearerToken	string	Yes
fetchImpl	typeof fetch	No

Expected JSON shape

```
{
  "baseUrl": "https://example.invalid/callback",
  "bearerToken": "example",
  "fetchImpl": "example"
}
```

QvtmApiErrorBody

Field	Type	Required
error	string	Yes
message	string	No
code	string	No
details	unknown	No

Expected JSON shape

```
{
  "error": "example",
  "message": "Operation completed.",
  "code": "example",
  "details": {}
}
```

HealthResponse

Field	Type	Required
status	string	Yes
service	string	Yes
version	string	No
deterministic	boolean	No

Expected JSON shape

```
{
  "status": "2026-07-10T12:00:00Z",
  "service": "qvtm-service",
  "version": "1.0.0",
  "deterministic": true
}
```

RuntimeStatusResponse

Field	Type	Required
status	string	Yes
service	string	Yes
runtime_mode	string	No
readiness	string	No

Expected JSON shape

```
{
  "status": "2026-07-10T12:00:00Z",
  "service": "qvtm-service",
  "runtime_mode": "2026-07-10T12:00:00Z",
  "readiness": "example"
}
```

StartLoginSessionRequest

Field	Type	Required
tenant_id	string	No
operator_ref	string	No
device_ref	string	No
redirect_uri	string	No
requested_scopes	readonly string[]	No

Expected JSON shape

```
{
  "tenant_id": "tenant-id-example",
  "operator_ref": "2026-07-10T12:00:00Z",
  "device_ref": "device-ref-example",
  "redirect_uri": "https://example.invalid/callback"
}
```

LoginSessionResponse

Field	Type	Required
login_session_id	string	Yes
challenge_id	string	No
status	ChallengeStatus	Yes
expires_at	string	No
next_action	string	No
session_handle_available	boolean	No

Expected JSON shape

```
{
  "login_session_id": "login-session-id-example",
  "challenge_id": "challenge-id-example",
  "status": "2026-07-10T12:00:00Z",
  "expires_at": "2026-07-10T12:00:00Z"
}
```

IssueAuthChallengeRequest

Field	Type	Required
tenant_id	string	No
operator_ref	string	No
device_ref	string	No

Field	Type	Required
requested_scopes	readonly string[]	No
expires_in_seconds	number	No

Expected JSON shape

```
{
  "tenant_id": "tenant-id-example",
  "operator_ref": "2026-07-10T12:00:00Z",
  "device_ref": "device-ref-example",
  "requested_scopes": [
    "example"
  ]
}
```

AuthChallengeResponse

Field	Type	Required
challenge_id	string	Yes
status	ChallengeStatus	Yes
tenant_id	string	No
operator_ref	string	No
device_ref	string	No
payload	Record<string, unknown>	No
expires_at	string	No
verification	ChallengeVerificationEvidence	No

Expected JSON shape

```
{
  "challenge_id": "challenge-id-example",
  "status": "2026-07-10T12:00:00Z",
  "tenant_id": "tenant-id-example",
  "operator_ref": "2026-07-10T12:00:00Z"
}
```

AuthChallengeStatusResponse

Field	Type	Required
challenge_id	string	Yes
status	ChallengeStatus	Yes
verified	boolean	Yes
expires_at	string	No
session_handle_available	boolean	No
evidence_ref	string	No

Expected JSON shape

```
{
  "challenge_id": "challenge-id-example",
  "status": "2026-07-10T12:00:00Z",
  "verified": true,
  "expires_at": "2026-07-10T12:00:00Z"
}
```

VerifyAuthChallengeRequest

Field	Type	Required
signature	string	Yes
public_key	string	Yes
algorithm	string	No
device_ref	string	No
operator_ref	string	No
signed_payload	Record<string, unknown>	No

Expected JSON shape

```
{
  "signature": "2026-07-10T12:00:00Z",
  "public_key": "base64url-public-key",
  "algorithm": "ML-DSA",
  "device_ref": "device-ref-example"
}
```

VerifyAuthChallengeResponse

Field	Type	Required
verified	boolean	Yes
challenge_id	string	Yes
status	ChallengeStatus	Yes
session_handle	string	No
expires_at	string	No
evidence_ref	string	No
verification	ChallengeVerificationEvidence	No

Expected JSON shape

```
{
  "verified": true,
  "challenge_id": "challenge-id-example",
  "status": "2026-07-10T12:00:00Z",
  "session_handle": "example"
}
```

ChallengeVerificationEvidence

Field	Type	Required
algorithm	string	No

Field	Type	Required
evidence_ref	string	No
verified_at	string	No
non_claims	readonly string[]	No

Expected JSON shape

```
{
  "algorithm": "ML-DSA",
  "evidence_ref": "evidence-ref-example",
  "verified_at": "2026-07-10T12:00:00Z",
  "non_claims": [
    "example"
  ]
}
```

SessionHandleConsumeRequest

Field	Type	Required
session_handle	string	No
expected_tenant_id	string	No
expected_operator_ref	string	No

Expected JSON shape

```
{
  "session_handle": "example",
  "expected_tenant_id": "expected-tenant-id-example",
  "expected_operator_ref": "2026-07-10T12:00:00Z"
}
```

SessionHandleResponse

Field	Type	Required
accepted	boolean	Yes
challenge_id	string	Yes
session_id	string	No
session_cookie_issued	boolean	No
evidence_ref	string	No
expires_at	string	No

Expected JSON shape

```
{
  "accepted": true,
  "challenge_id": "challenge-id-example",
  "session_id": "session-id-example",
  "session_cookie_issued": true
}
```

SessionDiagnosticsResponse

Field	Type	Required
authenticated	boolean	Yes
session_id	string	No
tenant_id	string	No
operator_ref	string	No
expires_at	string	No

Expected JSON shape

```
{
  "authenticated": true,
  "session_id": "session-id-example",
  "tenant_id": "tenant-id-example",
  "operator_ref": "2026-07-10T12:00:00Z"
}
```

TenantUpsertRequest

Field	Type	Required
tenant_id	string	Yes
display_name	string	Yes
lifecycle_state	TenantLifecycleState	No
region	string	No
plan_ref	string	No
metadata	Record<string, unknown>	No

Expected JSON shape

```
{
  "tenant_id": "tenant-id-example",
  "display_name": "Example",
  "lifecycle_state": "2026-07-10T12:00:00Z",
  "region": "example"
}
```

TenantRecord

Field	Type	Required	
tenant_id	string	Yes	
display_name	string	Yes	
lifecycle_state	TenantLifecycleState	string	Yes
region	string	No	
plan_ref	string	No	
created_at	string	No	
updated_at	string	No	
metadata	Record<string, unknown>	No	

Expected JSON shape

```
{
  "tenant_id": "tenant-id-example",
  "display_name": "Example",
  "lifecycle_state": "2026-07-10T12:00:00Z",
  "region": "example"
}
```

TenantOnboardingRequest

Field	Type	Required
tenant_id	string	Yes
display_name	string	Yes
owner_operator_ref	string	No
owner_display_name	string	No
plan_ref	string	No
region	string	No

Expected JSON shape

```
{
  "tenant_id": "tenant-id-example",
  "display_name": "Example",
  "owner_operator_ref": "2026-07-10T12:00:00Z",
  "owner_display_name": "Example"
}
```

TenantOnboardingResponse

Field	Type	Required
tenant_id	string	Yes
status	string	Yes
tenant	TenantRecord	No
owner_membership_ref	string	No
required_next_steps	readonly string[]	No

Expected JSON shape

```
{
  "tenant_id": "tenant-id-example",
  "status": "2026-07-10T12:00:00Z",
  "tenant": "example",
  "owner_membership_ref": "owner-membership-ref-example"
}
```

TenantLifecycleTransitionRequest

Field	Type	Required
target_state	TenantLifecycleState	Yes
reason	string	No

Field	Type	Required
actor_ref	string	No

Expected JSON shape

```
{
  "target_state": "2026-07-10T12:00:00Z",
  "reason": "example",
  "actor_ref": "actor-ref-example"
}
```

TenantSummaryResponse

Field	Type	Required	
tenant_id	string	Yes	
display_name	string	Yes	
lifecycle_state	TenantLifecycleState	string`	Yes
active_members	number	No	
pending_customer_invites	number	No	
open_package_deliveries	number	No	

Expected JSON shape

```
{
  "tenant_id": "tenant-id-example",
  "display_name": "Example",
  "lifecycle_state": "2026-07-10T12:00:00Z",
  "active_members": 1
}
```

CustomerInviteCreateRequest

Field	Type	Required
customer_ref	string	Yes
customer_display_name	string	Yes
customer_contact_ref	string	No
portal_base_url	string	No
delivery_channel	string	No
expires_in_seconds	number	No

Expected JSON shape

```
{
  "customer_ref": "customer-ref-example",
  "customer_display_name": "Example",
  "customer_contact_ref": "customer-contact-ref-example",
  "portal_base_url": "https://example.invalid/callback"
}
```

CustomerInviteResponse

Field	Type	Required				
invite_ref	string	Yes				
tenant_id	string	Yes				
customer_ref	string	Yes				
customer_display_name	string	No				
customer_contact_ref	string	No				
claim_url	string	Yes				
expires_at	string	Yes				
status	'pending'	'claimed'	'expired'	'revoked'	string`	No
required_claim_steps	readonly string[]	No				
non_claims	readonly string[]	No				

Expected JSON shape

```
{
  "invite_ref": "invite-ref-example",
  "tenant_id": "tenant-id-example",
  "customer_ref": "customer-ref-example",
  "customer_display_name": "Example",
  "claim_url": "https://example.invalid/callback",
  "expires_at": "2026-07-10T12:00:00Z"
}
```

PackageDeliveryCreateRequest

Field	Type	Required				
package_ref	string	Yes				
recipient_type	`customer'	'operator'	'verifier'	'auditor'	string`	Yes
recipient_ref	string	Yes				
delivery_channel	string	No				
package_type	string	No				
evidence_ref	string	No				
expires_in_seconds	number	No				
metadata	Record<string, unknown>	No				

Expected JSON shape

```
{
  "package_ref": "package-ref-example",
  "recipient_type": "customer",
  "recipient_ref": "recipient-ref-example",
  "delivery_channel": "example"
}
```

PackageDeliveryResponse

Field	Type	Required					
delivery_id	string	Yes					
tenant_id	string	Yes					
package_ref	string	Yes					
recipient_type	string	Yes					
recipient_ref	string	Yes					
delivery_channel	string	No					
package_type	string	No					
evidence_ref	string	No					
status	`pending`	'delivered'	'claimed'	'revoked'	'expired'	string`	Yes
expires_at	string	No					
created_at	string	No					
non_claims	readonly string[]	No					

Expected JSON shape

```
{
  "delivery_id": "delivery-id-example",
  "tenant_id": "tenant-id-example",
  "package_ref": "package-ref-example",
  "recipient_type": "example",
  "recipient_ref": "recipient-ref-example",
  "status": "pending"
}
```

DeviceTrustBundleRecord

Field	Type	Required	
bundle_id	string	Yes	
device_id	string	Yes	
domain_ref	string	Yes	
device_ref	string	Yes	
allowed_capabilities	readonly string[]	Yes	
policy_version	string	Yes	
issued_at_bucket	string	Yes	
expires_at_bucket	string	Yes	
issuer_signature_alg	string	Yes	
status	DeviceTrustBundleStatus	Yes	
revoked_at_bucket	`string`	null`	No
revocation_reason	`string`	null`	No
rotated_from_bundle_id	`string`	null`	No
rotated_to_bundle_id	`string`	null`	No

Field	Type	Required
audit_event_ids	readonly string[]	Yes

Expected JSON shape

```
{
  "bundle_id": "bundle-id-example",
  "device_id": "device-id-example",
  "domain_ref": "domain-ref-example",
  "device_ref": "device-ref-example",
  "allowed_capabilities": [
    "example"
  ],
  "policy_version": "1.0.0",
  "issued_at_bucket": "2026-07-10T12:00:00Z",
  "expires_at_bucket": "2026-07-10T12:00:00Z",
  "issuer_signature_alg": "2026-07-10T12:00:00Z",
  "status": "2026-07-10T12:00:00Z",
  "audit_event_ids": [
    "example"
  ]
}
```

IssueDeviceTrustBundleRequest

Field	Type	Required
device_id	string	Yes
domain_ref	string	Yes
device_ref	string	Yes
allowed_capabilities	readonly string[]	Yes
policy_version	string	Yes
issued_at_bucket	string	Yes
expires_at_bucket	string	Yes

Expected JSON shape

```
{
  "device_id": "device-id-example",
  "domain_ref": "domain-ref-example",
  "device_ref": "device-ref-example",
  "allowed_capabilities": [
    "example"
  ],
  "policy_version": "1.0.0",
  "issued_at_bucket": "2026-07-10T12:00:00Z",
  "expires_at_bucket": "2026-07-10T12:00:00Z"
}
```

RevokeDeviceTrustBundleRequest

Field	Type	Required
revoked_at_bucket	string	Yes
reason	string	Yes

Expected JSON shape

```
{
  "revoked_at_bucket": "2026-07-10T12:00:00Z",
  "reason": "example"
}
```

RotateDeviceTrustBundleRequest

Field	Type	Required
issued_at_bucket	string	Yes
expires_at_bucket	string	Yes

Expected JSON shape

```
{
  "issued_at_bucket": "2026-07-10T12:00:00Z",
  "expires_at_bucket": "2026-07-10T12:00:00Z"
}
```

DeviceTrustBundleResponse

Field	Type	Required
status	string	Yes
service	string	Yes
bundle	DeviceTrustBundleRecord	Yes

Expected JSON shape

```
{
  "status": "2026-07-10T12:00:00Z",
  "service": "qvtm-service",
  "bundle": "example"
}
```

ListDeviceTrustBundlesResponse

Field	Type	Required
status	string	Yes
service	string	Yes
bundles	readonly DeviceTrustBundleRecord[]	Yes

Expected JSON shape

```
{
  "status": "2026-07-10T12:00:00Z",
  "service": "qvtm-service",
  "bundles": []
}
```

DeviceTrustBundleStatusSnapshot

Field	Type	Required	
bundle_id	string	Yes	
status	DeviceTrustBundleStatus	Yes	
expires_at_bucket	string	Yes	
rotated_to_bundle_id	`string	null`	No
revocation_reason	`string	null`	No

Expected JSON shape

```
{
  "bundle_id": "bundle-id-example",
  "status": "2026-07-10T12:00:00Z",
  "expires_at_bucket": "2026-07-10T12:00:00Z",
  "rotated_to_bundle_id": "2026-07-10T12:00:00Z"
}
```

DeviceTrustBundleStatusListResponse

Field	Type	Required	
status	string	Yes	
service	string	Yes	
format	`qvtm.vault.device-trust-bundle-status-list.v1`	string`	Yes
generated_at_bucket	string	Yes	
snapshots	readonly DeviceTrustBundleStatusSnapshot[]	Yes	

Expected JSON shape

```
{
  "status": "2026-07-10T12:00:00Z",
  "service": "qvtm-service",
  "format": "qvtm.vault.device-trust-bundle-status-list.v1",
  "generated_at_bucket": "2026-07-10T12:00:00Z",
  "snapshots": []
}
```

DeviceTrustBundleStatusListSignature

Field	Type	Required	
issuer_key_ref	string	Yes	
signature_alg	string	Yes	
signature_ref	string	Yes	
signed_at_bucket	string	Yes	
transparency_checkpoint_ref	`string	null`	No
audit_event_ids	readonly string[]	Yes	

Expected JSON shape

```
{
  "issuer_key_ref": "issuer-key-ref-example",
  "signature_alg": "2026-07-10T12:00:00Z",
  "signature_ref": "2026-07-10T12:00:00Z",
  "signed_at_bucket": "2026-07-10T12:00:00Z",
  "audit_event_ids": [
    "example"
  ]
}
```

SignedDeviceTrustBundleStatusListResponse

Field	Type	Required
status	string	Yes
service	string	Yes
status_list	DeviceTrustBundleStatusListResponse	Yes
signature	DeviceTrustBundleStatusListSignature	Yes

Expected JSON shape

```
{
  "status": "2026-07-10T12:00:00Z",
  "service": "qvtm-service",
  "status_list": "2026-07-10T12:00:00Z",
  "signature": "2026-07-10T12:00:00Z"
}
```

DeviceTrustBundleAuditEvent

Field	Type	Required	
audit_event_id	string	Yes	
bundle_id	string	Yes	
action	DeviceTrustBundleAuditAction	Yes	
occurred_at_bucket	string	Yes	
actor_ref	string	Yes	
evidence_ref	string	Yes	
previous_bundle_id	`string`	null`	No
next_bundle_id	`string`	null`	No
status_after	DeviceTrustBundleStatus	Yes	

Expected JSON shape

```
{
  "audit_event_id": "audit-event-id-example",
  "bundle_id": "bundle-id-example",
  "action": "example",
  "occurred_at_bucket": "2026-07-10T12:00:00Z",
  "actor_ref": "actor-ref-example",
  "evidence_ref": "evidence-ref-example",
  "status_after": "2026-07-10T12:00:00Z"
}
```

DeviceTrustBundleAuditTrailResponse

Field	Type	Required	
status	string	Yes	
service	string	Yes	
format	`qvtm.vault.device-trust-bundle-audit-trail.v1`	string`	Yes
generated_at_bucket	string	Yes	
bundle_id	`string	null`	No
events	readonly DeviceTrustBundleAuditEvent[]	Yes	

Expected JSON shape

```
{
  "status": "2026-07-10T12:00:00Z",
  "service": "qvtm-service",
  "format": "qvtm.vault.device-trust-bundle-audit-trail.v1",
  "generated_at_bucket": "2026-07-10T12:00:00Z",
  "events": []
}
```

DeviceTrustBundleEvidenceReceipt

Field	Type	Required	
receipt_id	string	Yes	
bundle_id	string	Yes	
action	DeviceTrustBundleEvidenceAction	Yes	
issued_at_bucket	string	Yes	
issuer_key_ref	string	Yes	
signature_alg	string	Yes	
signature_ref	string	Yes	
audit_event_id	string	Yes	
transparency_checkpoint_ref	`string	null`	No
previous_bundle_id	`string	null`	No
next_bundle_id	`string	null`	No
status_after	DeviceTrustBundleStatus	Yes	

Expected JSON shape

```
{
  "receipt_id": "receipt-id-example",
  "bundle_id": "bundle-id-example",
  "action": "example",
  "issued_at_bucket": "2026-07-10T12:00:00Z",
  "issuer_key_ref": "issuer-key-ref-example",
  "signature_alg": "2026-07-10T12:00:00Z",
  "signature_ref": "2026-07-10T12:00:00Z",
  "audit_event_id": "audit-event-id-example",
  "status_after": "2026-07-10T12:00:00Z"
}
```

DeviceTrustBundleEvidencePackageResponse

Field	Type	Required	
status	string	Yes	
service	string	Yes	
format	`qvtm.vault.device-trust-bundle-evidence-package.v1`	string`	Yes
generated_at_bucket	string	Yes	
bundle	DeviceTrustBundleRecord	Yes	
signed_status_list	SignedDeviceTrustBundleStatusListResponse	Yes	
audit_trail	DeviceTrustBundleAuditTrailResponse	Yes	
receipts	readonly DeviceTrustBundleEvidenceReceipt[]	Yes	

Expected JSON shape

```
{
  "status": "2026-07-10T12:00:00Z",
  "service": "qvtm-service",
  "format": "qvtm.vault.device-trust-bundle-evidence-package.v1",
  "generated_at_bucket": "2026-07-10T12:00:00Z",
  "bundle": "example",
  "signed_status_list": "2026-07-10T12:00:00Z",
  "audit_trail": "example",
  "receipts": []
}
```

DeviceTrustBundleReadinessCheck

Field	Type	Required
check_id	string	Yes
passed	boolean	Yes
severity	string	Yes
message	string	Yes

Expected JSON shape

```
{
  "check_id": "check-id-example",
  "passed": true,
  "severity": "example",
  "message": "Operation completed."
}
```

DeviceTrustBundleReadinessReportResponse

Field	Type	Required	
status	string	Yes	
service	string	Yes	
format	`qvtm.vault.device-trust-bundle-readiness-report.v1`	string`	Yes
generated_at_bucket	string	Yes	

Field	Type	Required
bundle_id	string	Yes
readiness_status	DeviceTrustBundleReadinessStatus	Yes
checks	readonly DeviceTrustBundleReadinessCheck[]	Yes
evidence_package	DeviceTrustBundleEvidencePackageResponse	Yes

Expected JSON shape

```
{
  "status": "2026-07-10T12:00:00Z",
  "service": "qvtm-service",
  "format": "qvtm.vault.device-trust-bundle-readiness-report.v1",
  "generated_at_bucket": "2026-07-10T12:00:00Z",
  "bundle_id": "bundle-id-example",
  "readiness_status": "2026-07-10T12:00:00Z",
  "checks": [],
  "evidence_package": "example"
}
```

DeviceTrustBundleDistributionPackageResponse

Field	Type	Required	
status	string	Yes	
service	string	Yes	
format	`qvtm.vault.device-trust-bundle-distribution-package.v1`	string`	Yes
generated_at_bucket	string	Yes	
bundle_id	string	Yes	
distribution_status	DeviceTrustBundleDistributionStatus	Yes	
evidence_package	DeviceTrustBundleEvidencePackageResponse	Yes	
readiness_report	DeviceTrustBundleReadinessReportResponse	Yes	

Expected JSON shape

```
{
  "status": "2026-07-10T12:00:00Z",
  "service": "qvtm-service",
  "format": "qvtm.vault.device-trust-bundle-distribution-package.v1",
  "generated_at_bucket": "2026-07-10T12:00:00Z",
  "bundle_id": "bundle-id-example",
  "distribution_status": "2026-07-10T12:00:00Z",
  "evidence_package": "example",
  "readiness_report": "example"
}
```

DeviceTrustBundleDistributionManifestResponse

Field	Type	Required	
status	string	Yes	
service	string	Yes	
format	`qvtm.vault.device-trust-bundle-distribution-manifest.v1`	string`	Yes

Field	Type	Required	
<code>generated_at_bucket</code>	<code>string</code>	Yes	
<code>package_count</code>	<code>number</code>	Yes	
<code>phase_readiness_status</code>	<code>DeviceTrustBundleDistributionStatus</code>	Yes	
<code>required_formats</code>	<code>readonly string[]</code>	Yes	
<code>packages</code>	<code>readonly DeviceTrustBundleDistributionPackageResponse[]</code>	Yes	

Expected JSON shape

```
{
  "status": "2026-07-10T12:00:00Z",
  "service": "qvtm-service",
  "format": "qvtm.vault.device-trust-bundle-distribution-manifest.v1",
  "generated_at_bucket": "2026-07-10T12:00:00Z",
  "package_count": 1,
  "phase_readiness_status": "2026-07-10T12:00:00Z",
  "required_formats": [
    "example"
  ],
  "packages": []
}
```

DeviceTrustDistributionClosureCheck

Field	Type	Required	
<code>check_id</code>	<code>string</code>	Yes	
<code>passed</code>	<code>boolean</code>	Yes	
<code>severity</code>	<code>string</code>	Yes	
<code>message</code>	<code>string</code>	Yes	

Expected JSON shape

```
{
  "check_id": "check-id-example",
  "passed": true,
  "severity": "example",
  "message": "Operation completed."
}
```

DeviceTrustDistributionClosureReportResponse

Field	Type	Required	
<code>status</code>	<code>string</code>	Yes	
<code>service</code>	<code>string</code>	Yes	
<code>format</code>	<code>`qvtm.vault.device-trust-distribution-closure-report.v1`</code>	<code>string`</code>	Yes
<code>generated_at_bucket</code>	<code>string</code>	Yes	
<code>closure_status</code>	<code>DeviceTrustDistributionClosureStatus</code>	Yes	
<code>manifest</code>	<code>DeviceTrustBundleDistributionManifestResponse</code>	Yes	
<code>checks</code>	<code>readonly DeviceTrustDistributionClosureCheck[]</code>	Yes	

Field	Type	Required
next_phase	string	Yes

Expected JSON shape

```
{
  "status": "2026-07-10T12:00:00Z",
  "service": "qvtm-service",
  "format": "qvtm.vault.device-trust-distribution-closure-report.v1",
  "generated_at_bucket": "2026-07-10T12:00:00Z",
  "closure_status": "2026-07-10T12:00:00Z",
  "manifest": "example",
  "checks": [],
  "next_phase": "example"
}
```

DeviceTrustPhase61ReadinessSignal

Field	Type	Required
signal_id	string	Yes
passed	boolean	Yes
severity	string	Yes
message	string	Yes

Expected JSON shape

```
{
  "signal_id": "signal-id-example",
  "passed": true,
  "severity": "example",
  "message": "Operation completed."
}
```

DeviceTrustPhase61FinalReadinessReportResponse

Field	Type	Required	
status	string	Yes	
service	string	Yes	
format	`qvtm.vault.device-trust-phase61-final-readiness.v1`	string`	Yes
generated_at_bucket	string	Yes	
phase	string	Yes	
readiness_status	DeviceTrustPhase61ReadinessStatus	Yes	
closure_report	DeviceTrustDistributionClosureReportResponse	Yes	
signals	readonly DeviceTrustPhase61ReadinessSignal[]	Yes	
residual_risks	readonly string[]	Yes	
next_phase	string	Yes	

Expected JSON shape

```
{
  "status": "2026-07-10T12:00:00Z",
  "service": "qvtm-service",
  "format": "qvtm.vault.device-trust-phase61-final-readiness.v1",
  "generated_at_bucket": "2026-07-10T12:00:00Z",
  "phase": "example",
  "readiness_status": "2026-07-10T12:00:00Z",
  "closure_report": "example",
  "signals": [],
  "residual_risks": [
    "example"
  ],
  "next_phase": "example"
}
```

AdminWriteResponse

Field	Type	Required
accepted	boolean	Yes
tenant_id	string	Yes
action	string	Yes
evidence_ref	string	No
required_follow_up	readonly string[]	No

Expected JSON shape

```
{
  "accepted": true,
  "tenant_id": "tenant-id-example",
  "action": "example",
  "evidence_ref": "evidence-ref-example"
}
```

QvtmWorkflowIdempotencyKeys

Field	Type	Required
tenant	string	No
customerInvite	string	No
packageDelivery	string	No

Expected JSON shape

```
{
  "tenant": "example",
  "customerInvite": "example",
  "packageDelivery": "example"
}
```

ProvisionTenantCustomerInviteWorkflowRequest

Field	Type	Required
tenant	TenantProvisioningRequest	Yes
customerInvite	CustomerInviteCreateRequest	Yes

Field	Type	Required
idempotencyKeys	QvtmWorkflowIdempotencyKeys	No

Expected JSON shape

```
{
  "tenant": "example",
  "customerInvite": "example",
  "idempotencyKeys": "example"
}
```

ProvisionTenantCustomerInviteWorkflowResponse

Field	Type	Required
tenant	TenantRecord	Yes
customerInvite	CustomerInviteResponse	Yes
required_next_steps	readonly string[]	Yes

Expected JSON shape

```
{
  "tenant": "example",
  "customerInvite": "example",
  "required_next_steps": [
    "example"
  ]
}
```

AuthChallengePollOptions

Field	Type	Required
intervalMs	number	No
timeoutMs	number	No
signal	AbortSignal	No

Expected JSON shape

```
{
  "intervalMs": 1,
  "timeoutMs": 1,
  "signal": "example"
}
```

Published enums and unions

Type	Values / expression
QvtmSdkStatus	'stable' | 'experimental' | 'internal' | 'legacy' | 'deprecated'
ChallengeStatus	'issued' | 'pending' | 'verified' | 'expired' | 'revoked' | 'failed' | string
TenantLifecycleState	'draft' | 'active' | 'suspended' | 'archived'
DeviceTrustBundleStatus	'Active' | 'Revoked' | 'Expired' | 'Rotated' | string

Type	Values / expression
DeviceTrustBundleAuditAction	'Issued' | 'Revoked' | 'RotatedFrom' | 'RotatedTo' | 'StatusListExported' | 'SignedStatusListExported' | string
DeviceTrustBundleEvidenceAction	'Issued' | 'Revoked' | 'RotatedFrom' | 'RotatedTo' | string
DeviceTrustBundleReadinessStatus	'Ready' | 'Blocked' | string
DeviceTrustBundleDistributionStatus	'ReadyForOfflineUse' | 'Blocked' | string
DeviceTrustDistributionClosureStatus	'ReadyToClosePhase61' | 'Blocked' | string
DeviceTrustPhase61ReadinessStatus	'ReadyForPhase62' | 'Blocked' | string

Error classes

- QvtmControlPlaneError

33. @qvtm/trust-verifier

Public methods

Method	Parameters	Returns
capabilities	—	Promise<TrustCapabilitiesResponse>
inspectEvidence	body: TrustEvidenceInspectRequest	Promise<TrustEvidenceInspectResponse>
verifyEvidence	body: TrustEvidenceVerifyRequest	Promise<TrustEvidenceVerifyResponse>
getArtifactPackage	packageId: string	Promise<TrustArtifactPackageResponse>
verifyArtifactPackage	packageId: string, body?: TrustArtifactPackageVerifyRequest,	Promise<TrustArtifactPackageVerifyResponse>
getTransparencyExportEnvelope	exportDigest: string	Promise<TransparencyExportEnvelopeResponse>

Public object contracts

QvtmTrustVerifierClientOptions

Field	Type	Required
baseUrl	string	Yes
bearerToken	string	No
fetchImpl	typeof fetch	No

Expected JSON shape

```
{
  "baseUrl": "https://example.invalid/callback",
  "bearerToken": "example",
  "fetchImpl": "example"
}
```

TrustHealthResponse

Field	Type	Required
status	string	Yes
service	string	No

Field	Type	Required
version	string	No

Expected JSON shape

```
{
  "status": "2026-07-10T12:00:00Z",
  "service": "qvtm-service",
  "version": "1.0.0"
}
```

TrustCapabilitiesResponse

Field	Type	Required
capabilities	readonly string[]	Yes
algorithms	readonly string[]	No
evidence_types	readonly string[]	No

Expected JSON shape

```
{
  "capabilities": [
    "example"
  ],
  "algorithms": [
    "example"
  ],
  "evidence_types": [
    "example"
  ]
}
```

TrustEvidenceInspectRequest

Field	Type	Required
evidence	unknown	Yes
expected_tenant_id	string	No
expected_operator_ref	string	No
expected_device_ref	string	No

Expected JSON shape

```
{
  "evidence": {},
  "expected_tenant_id": "expected-tenant-id-example",
  "expected_operator_ref": "2026-07-10T12:00:00Z",
  "expected_device_ref": "expected-device-ref-example"
}
```

TrustEvidenceInspectResponse

Field	Type	Required
accepted	boolean	Yes

Field	Type	Required
evidence_type	string	No
evidence_ref	string	No
claims	Record<string, unknown>	No
warnings	readonly string[]	No
non_claims	readonly string[]	No

Expected JSON shape

```
{
  "accepted": true,
  "evidence_type": "example",
  "evidence_ref": "evidence-ref-example",
  "claims": {}
}
```

TrustArtifactPackageResponse

Field	Type	Required
package_id	string	Yes
status	string	Yes
evidence_refs	readonly string[]	No
manifest	Record<string, unknown>	No

Expected JSON shape

```
{
  "package_id": "package-id-example",
  "status": "2026-07-10T12:00:00Z",
  "evidence_refs": [
    "example"
  ],
  "manifest": {}
}
```

TrustArtifactPackageVerifyRequest

Field	Type	Required
expected_tenant_id	string	No
expected_customer_ref	string	No
require_all_evidence	boolean	No

Expected JSON shape

```
{
  "expected_tenant_id": "expected-tenant-id-example",
  "expected_customer_ref": "expected-customer-ref-example",
  "require_all_evidence": true
}
```

TrustArtifactPackageVerifyResponse

Field	Type	Required
package_id	string	Yes
verified	boolean	Yes
verification_ref	string	No
failed_checks	readonly string[]	No
warnings	readonly string[]	No

Expected JSON shape

```
{
  "package_id": "package-id-example",
  "verified": true,
  "verification_ref": "2026-07-10T12:00:00Z",
  "failed_checks": [
    "example"
  ]
}
```

TransparencyExportEnvelopeResponse

Field	Type	Required
export_digest	string	Yes
status	string	No
envelope	Record<string, unknown>	No

Expected JSON shape

```
{
  "export_digest": "sha256:example",
  "status": "2026-07-10T12:00:00Z",
  "envelope": {}
}
```

Error classes

- QvtmTrustVerifierError

34. @qvtm/runtime-protocol

Public methods

Method	Parameters	Returns
version	—	Promise<RuntimeVersionResponse>
registerDevice	body: RuntimeDeviceRegistrationRequest	Promise<RuntimeDeviceRecord>
getDevice	deviceId: string	Promise<RuntimeDeviceRecord>
startHandshake	body: RuntimeHandshakeStartRequest	Promise<RuntimeHandshakeResponse>
finishHandshake	body: RuntimeHandshakeFinishRequest	Promise<RuntimeHandshakeResponse>
sendMessage	body: RuntimeMessageSendRequest	Promise<RuntimeMessageSendResponse>

Method	Parameters	Returns
getTransparencyKey	deviceId: string	Promise<RuntimeTransparencyKeyResponse>
getTransparencyState	deviceId: string	Promise<RuntimeTransparencyStateResponse>
listTransparencyHistory	deviceId: string	Promise<RuntimeTransparencyHistoryResponse>

Public object contracts

QvtmRuntimeClientOptions

Field	Type	Required
baseUrl	string	Yes
bearerToken	string	No
fetchImpl	typeof fetch	No

Expected JSON shape

```
{
  "baseUrl": "https://example.invalid/callback",
  "bearerToken": "example",
  "fetchImpl": "example"
}
```

RuntimeHealthResponse

Field	Type	Required
status	string	Yes
service	string	No
version	string	No

Expected JSON shape

```
{
  "status": "2026-07-10T12:00:00Z",
  "service": "qvtm-service",
  "version": "1.0.0"
}
```

RuntimeVersionResponse

Field	Type	Required
version	string	Yes
protocol_version	string	No
capabilities	readonly string[]	No

Expected JSON shape

```
{
  "version": "1.0.0",
  "protocol_version": "1.0.0",
  "capabilities": [
    "example"
  ]
}
```

RuntimeDeviceRegistrationRequest

Field	Type	Required
device_id	string	No
device_ref	string	No
operator_ref	string	No
tenant_id	string	No
public_key	string	No
algorithm	string	No
metadata	Record<string, unknown>	No

Expected JSON shape

```
{
  "device_id": "device-id-example",
  "device_ref": "device-ref-example",
  "operator_ref": "2026-07-10T12:00:00Z",
  "tenant_id": "tenant-id-example"
}
```

RuntimeHandshakeStartRequest

Field	Type	Required
initiator_device_id	string	Yes
recipient_device_id	string	Yes
context	Record<string, unknown>	No

Expected JSON shape

```
{
  "initiator_device_id": "2026-07-10T12:00:00Z",
  "recipient_device_id": "recipient-device-id-example",
  "context": {}
}
```

RuntimeHandshakeFinishRequest

Field	Type	Required
handshake_id	string	Yes
device_id	string	Yes
signed_payload	Record<string, unknown>	No
signature	string	No

Expected JSON shape

```
{
  "handshake_id": "handshake-id-example",
  "device_id": "device-id-example",
  "signed_payload": {},
  "signature": "2026-07-10T12:00:00Z"
}
```

RuntimeHandshakeResponse

Field	Type	Required
handshake_id	string	Yes
status	string	Yes
expires_at	string	No
transcript_ref	string	No
next_action	string	No

Expected JSON shape

```
{
  "handshake_id": "handshake-id-example",
  "status": "2026-07-10T12:00:00Z",
  "expires_at": "2026-07-10T12:00:00Z",
  "transcript_ref": "transcript-ref-example"
}
```

RuntimeMessageSendRequest

Field	Type	Required
sender_device_id	string	Yes
recipient_device_id	string	Yes
ciphertext	string	Yes
content_type	string	No
envelope	Record<string, unknown>	No

Expected JSON shape

```
{
  "sender_device_id": "sender-device-id-example",
  "recipient_device_id": "recipient-device-id-example",
  "ciphertext": "example",
  "content_type": "example"
}
```

RuntimeMessageSendResponse

Field	Type	Required
message_id	string	Yes
status	string	Yes
delivered_at	string	No

Field	Type	Required
evidence_ref	string	No

Expected JSON shape

```
{
  "message_id": "message-id-example",
  "status": "2026-07-10T12:00:00Z",
  "delivered_at": "2026-07-10T12:00:00Z",
  "evidence_ref": "evidence-ref-example"
}
```

RuntimeTransparencyKeyResponse

Field	Type	Required
device_id	string	Yes
public_key	string	No
algorithm	string	No
transparency_ref	string	No
state	string	No

Expected JSON shape

```
{
  "device_id": "device-id-example",
  "public_key": "base64url-public-key",
  "algorithm": "ML-DSA",
  "transparency_ref": "transparency-ref-example"
}
```

RuntimeTransparencyStateResponse

Field	Type	Required
device_id	string	Yes
state	string	Yes
current_digest	string	No
updated_at	string	No

Expected JSON shape

```
{
  "device_id": "device-id-example",
  "state": "2026-07-10T12:00:00Z",
  "current_digest": "sha256:example",
  "updated_at": "2026-07-10T12:00:00Z"
}
```

RuntimeTransparencyHistoryResponse

Field	Type	Required
device_id	string	Yes

Field	Type	Required
entries	readonly Record<string, unknown>[]	Yes

Expected JSON shape

```
{
  "device_id": "device-id-example",
  "entries": {}
}
```

Error classes

- QvtmRuntimeProtocolError

35. @qvtm/vault

Public methods

Method	Parameters	Returns	
listDevices	—	Promise<readonly QvtmVaultDevice[]>	
createVault	request: QvtmVaultCreateRequest	Promise<QvtmVaultOperationReceipt>	
openVault	request: QvtmVaultOpenRequest	Promise<QvtmVaultOperationReceipt>	
importVault	request: QvtmVaultImportRequest	Promise<QvtmVaultOperationReceipt>	
exportVaultCopy	request: QvtmVaultExportCopyRequest	Promise<QvtmVaultOperationReceipt>	
removeLocalVault	vaultId: string	Promise<QvtmVaultOperationReceipt>	
requestIdentityApproval	request: QvtmVaultIdentityApprovalRequest	Promise<QvtmVaultOperationReceipt>	
syncVault	request: QvtmVaultSyncRequest	Promise<QvtmVaultOperationReceipt>	
onEvent	listener: (event: QvtmVaultEvent) => void): () => void { if (typeof listener !== 'function') { throw new QvtmVaultError('invalid-request', 'Vault event listener must be a function.')} return this.provider.onEvent?.(listener) ?? () => undefined; } } export function createQvtmVaultClient(options: QvtmVaultClientOptions	QvtmVaultClient	
assertVaultFile	ref; return ref; } export function assertQvtmVaultFileRef(file: QvtmVaultFileRef	void	
assertVaultFile	file); } function assertProvider(provider: QvtmVaultProvider	void	
assertSafeVaultFileName	file.displayName); if (file.locationHint !== undefined) assertSafeLabel(file.locationHint, 'locationHint', MAX_LOCATION_HINT_LENGTH); } function assertSafeVaultFileName(displayName: string	void	
assertSafeLabel	`displayName, 'displayName', MAX_DISPLAY_NAME_LENGTH); if (displayName.trim().length < 2) { throw new QvtmVaultError('invalid-request', 'Vault display name is required.')} } } function assertEnvironment(environment: QvtmVaultEnvironment	undefined`	void
assertSafeLabel	value, field, 128); if (value.trim().length === 0) { throw new QvtmVaultError('invalid-request', `\${field} is required. '); } } function assertSafeLabel(value: string, field: string, maxLength: number	void	

Public object contracts

QvtmVaultFileRef

Field	Type	Required
kind	'vault-file'	Yes

Field	Type	Required
displayName	string	Yes
extension	typeof QVTM_VAULT_FILE_EXTENSION	Yes
locationHint	string	No

Expected JSON shape

```
{
  "kind": "vault-file",
  "displayName": "Example",
  "extension": "example",
  "locationHint": "2026-07-10T12:00:00Z"
}
```

QvtmVaultDescriptor

Field	Type	Required
vaultId	string	Yes
displayName	string	Yes
file	QvtmVaultFileRef	No
health	QvtmVaultHealth	Yes
deviceState	QvtmVaultDeviceState	Yes
syncState	QvtmVaultSyncState	No
lastOpenedLabel	string	No

Expected JSON shape

```
{
  "vaultId": "vaultId-example",
  "displayName": "Example",
  "file": "example",
  "health": "example",
  "deviceState": "2026-07-10T12:00:00Z"
}
```

QvtmVaultSession

Field	Type	Required
sessionId	string	Yes
vaultId	string	Yes
state	QvtmVaultSessionState	Yes
openedAtLabel	string	No
offlineAvailable	true	Yes
remoteOpenAllowed	false	Yes

Expected JSON shape

```
{
  "sessionId": "sessionId-example",
  "vaultId": "vaultId-example",
  "state": "2026-07-10T12:00:00Z",
  "openedAtLabel": "2026-07-10T12:00:00Z",
  "offlineAvailable": "example",
  "remoteOpenAllowed": "example"
}
```

QvtmVaultDevice

Field	Type	Required
deviceId	string	Yes
label	string	Yes
current	boolean	Yes
state	QvtmVaultDeviceState	Yes
lastSeenLabel	string	No

Expected JSON shape

```
{
  "deviceId": "deviceId-example",
  "label": "Example",
  "current": true,
  "state": "2026-07-10T12:00:00Z"
}
```

QvtmVaultCreateRequest

Field	Type	Required
displayName	string	Yes
destination	QvtmVaultFileRef	No
environment	QvtmVaultEnvironment	No

Expected JSON shape

```
{
  "displayName": "Example",
  "destination": "2026-07-10T12:00:00Z",
  "environment": "example"
}
```

QvtmVaultOpenRequest

Field	Type	Required
file	QvtmVaultFileRef	Yes
requireCurrentDevice	boolean	No

Expected JSON shape

```
{
  "file": "example",
  "requireCurrentDevice": true
}
```

QvtmVaultImportRequest

Field	Type	Required
file	QvtmVaultFileRef	Yes
rememberOnThisDevice	boolean	No

Expected JSON shape

```
{
  "file": "example",
  "rememberOnThisDevice": true
}
```

QvtmVaultExportCopyRequest

Field	Type	Required
vaultId	string	Yes
destination	QvtmVaultFileRef	No

Expected JSON shape

```
{
  "vaultId": "vaultId-example",
  "destination": "2026-07-10T12:00:00Z"
}
```

QvtmVaultSyncRequest

Field	Type	Required		
vaultId	string	Yes		
mode	``manual-upload``	'manual-download'	'status-only``	Yes

Expected JSON shape

```
{
  "vaultId": "vaultId-example",
  "mode": "manual-upload"
}
```

QvtmVaultIdentityApprovalRequest

Field	Type	Required		
reason	``trust-current-device``	'recover-access'	'review-device``	Yes
userVisibleLabel	string	No		

Expected JSON shape

```
{
  "reason": "trust-current-device",
  "userVisibleLabel": "Example"
}
```

QvtmVaultOperationReceipt

Field	Type	Required
operationId	string	Yes
ok	true	Yes
message	string	Yes
vault	QvtmVaultDescriptor	No
session	QvtmVaultSession	No
syncState	QvtmVaultSyncState	No

Expected JSON shape

```
{
  "operationId": "2026-07-10T12:00:00Z",
  "ok": "example",
  "message": "Operation completed.",
  "vault": "example"
}
```

QvtmVaultEvent

Field	Type	Required
name	QvtmVaultEventName	Yes
atLabel	string	No
vaultId	string	No
session	QvtmVaultSession	No
message	string	No

Expected JSON shape

```
{
  "name": "Example",
  "atLabel": "2026-07-10T12:00:00Z",
  "vaultId": "vaultId-example",
  "session": "example"
}
```

QvtmVaultClientOptions

Field	Type	Required
provider	QvtmVaultProvider	Yes

Expected JSON shape

```
{
  "provider": "example"
}
```

Published enums and unions

Type	Values / expression
<code>QtmVaultEnvironment</code>	'personal' | 'enterprise'
<code>QtmVaultHealth</code>	'ready' | 'needs-device-approval' | 'recovery-required' | 'locked'
<code>QtmVaultDeviceState</code>	'trusted' | 'needs-review' | 'revoked' | 'unknown'
<code>QtmVaultSyncState</code>	'offline' | 'local-only' | 'pending' | 'synced' | 'conflict'
<code>QtmVaultSessionState</code>	'closed' | 'opening' | 'open' | 'locked'

Error classes

- `QtmVaultError`

36. Operation-to-Response Traceability

The following rule applies throughout this reference:

1. A public method identifies its request type and return type in the package method table.
2. The corresponding public object contract defines every published field and its required status.
3. The JSON block beneath that contract shows the expected response shape.
4. Errors are represented by the package-specific public error class and, for HTTP packages, the published API error body.
5. An application must not treat transport success as Trust Continuity or authorization success unless the returned contract explicitly carries that result.

For overloaded or convenience methods, the return contract is identical to the underlying published operation. Aliases such as `health()` and `getControlPlaneHealth()` are documented separately because both are public, but they return the same published type.

36.1 Cross-Cutting Response Invariants

Observable condition	Required interpretation
Transport or method call completed successfully	The SDK operation completed at its transport or binding boundary; this does not by itself establish Trust Continuity or application authorization.
<code>status</code> is <code>completed</code>	The bounded operation completed; the application must still inspect any Trust Evaluation, conditions, warnings, and continuity effect.
CEP verification succeeded	The CEP passed the applicable syntax, integrity, compatibility, and semantic verification boundary; this does not by itself establish a continuous relationship.
Synchronization material was accepted	The material was accepted into the applicable synchronization context; acceptance does not imply application authorization.
<code>recoverable</code> is <code>true</code>	A corrective or retry path exists; immediate repetition is safe only when the operation is idempotent or the documentation explicitly permits it.
<code>continuityEffect</code> is <code>none</code>	No new Continuity Evidence was committed by the operation. The operation may still have returned useful application data.
<code>continuityEffect</code> is <code>extended</code>	New evidence extended existing history after the applicable commit boundary; historical evidence was not rewritten.
An open or string-extensible value is returned	The application must preserve forward compatibility and must not reject unknown values unless local policy explicitly requires a closed set.

36.2 Public Data-Handling Labels

Field tables and binding-specific references may use the following labels where the implementation or profile exposes the distinction.

Label	Developer obligation
Secret	Must remain inside the trusted boundary and must never be logged or placed in ordinary application metadata.
Sensitive	May cross an approved boundary but must be protected in storage, transport, diagnostics, and telemetry.
Opaque	May be stored and returned through documented APIs but must not be parsed or reconstructed by the application.
Public identifier	May be used for correlation according to tenant, privacy, and retention policy; it is not automatically safe for unrestricted publication.

37. Public Documentation Boundary

This reference publishes only application-facing types and methods found in the supplied SDK implementation. It does not publish:

- private request helpers;
- internal protocol scheduling;
- cryptographic key material;
- private storage layouts;
- proprietary Trust Evaluation logic;
- Graph Puzzle internals;
- test-only fixtures;
- generated build artifacts as independent contracts.

Where a public response contains `unknown`, open extension fields, or string-extensible status values, applications must preserve forward compatibility and avoid assuming undocumented structure.

Appendices

The appendices provide reference material intended to support SDK integration.

Unlike the previous chapters, which describe concepts and workflows, the appendices serve as quick-reference material for developers during implementation and maintenance.

They are informative rather than normative.

Appendix A. Error Codes

A.1 Overview

Every SDK implementation exposes a stable set of observable error codes.

Error codes allow applications to implement consistent diagnostics and recovery strategies while remaining independent of SDK implementation details.

Individual SDK implementations MAY introduce additional implementation-specific error codes.

Such extensions SHOULD NOT modify the semantics of the common error categories defined by this document.

A.2 Error Code Structure

Conceptually, every SDK error consists of:

Field	Description
Category	High-level error classification.
Code	Stable implementation-independent identifier.
Message	Human-readable diagnostic description.
Recoverable	Indicates whether the operation may be retried.

Applications SHOULD rely primarily on the error category and code.

Human-readable messages are intended for diagnostics.

A.3 Common Error Codes

Error Code	Category	Recoverable
SDK_CONFIGURATION_ERROR	Configuration Error	No
SDK_INITIALIZATION_ERROR	Initialization Error	No
SDK_SESSION_ERROR	Session Error	Yes
SDK_IDENTITY_ERROR	Identity Error	Depends
SDK_ENROLLMENT_ERROR	Enrollment Error	Depends
SDK_AUTHENTICATION_ERROR	Authentication Error	Depends
SDK_SYNCHRONIZATION_ERROR	Synchronization Error	Yes
SDK_RECOVERY_ERROR	Recovery Error	Depends
SDK_VAULT_ERROR	Vault Error	Depends
SDK_COMPATIBILITY_ERROR	Compatibility Error	No
SDK_RUNTIME_ERROR	Runtime Error	Depends

These identifiers represent conceptual SDK error codes.

Individual SDK implementations MAY adapt naming conventions while preserving observable semantics.

A.4 Error Handling Recommendations

Applications SHOULD:

- process every error explicitly;
- distinguish recoverable and non-recoverable conditions;
- avoid exposing internal diagnostics to end users;
- preserve diagnostic information for operational troubleshooting;
- avoid deriving protocol state from error codes.

Additional error handling guidance is provided in Chapter 22.

Appendix B. Common Integration Patterns

B.1 Overview

This appendix summarizes common application integration patterns using the QVTM SDK.

These patterns are informative.

They illustrate recommended architectural approaches without prescribing implementation details.

B.2 Long-Lived Client Pattern

Create a single Client during application startup.

Reuse the Client throughout the operational lifetime of the application.

Create independent Sessions for individual protocol operations.

This minimizes initialization overhead while simplifying lifecycle management.

B.3 Short-Lived Session Pattern

Treat Sessions as temporary execution contexts.

A recommended workflow is:

Applications SHOULD avoid using Sessions as long-lived application state.

B.4 Offline-First Pattern

Applications supporting disconnected environments SHOULD continue normal operation while offline.

When connectivity becomes available:

The SDK coordinates synchronization automatically through documented public APIs.

B.5 Capability Discovery Pattern

Applications SHOULD discover available functionality dynamically.

Capability discovery provides greater compatibility than explicit version checks.

B.6 Authentication Pattern

Authentication SHOULD follow the sequence:

Applications remain responsible for authorization decisions.

B.7 Vault Pattern

Typical Vault integrations follow:

Applications SHOULD delegate all continuity management to the SDK.

B.8 Synchronization Pattern

Synchronization SHOULD occur opportunistically.

Typical triggers include:

- application startup;
- connectivity restoration;
- completion of protected operations;
- scheduled maintenance.

Applications SHOULD avoid continuous synchronization unless operationally required.

B.9 Recovery Pattern

Applications SHOULD treat Recovery as part of the normal lifecycle.

Recovery extends continuity.

It does not recreate continuity history.

B.10 Multi-Profile Pattern

Applications MAY combine multiple Deployment Profiles.

For example:

All profiles reuse the same Core SDK object model.

B.11 Recommended Architecture

Applications SHOULD organize responsibilities as follows:

Business policy remains outside the SDK.

Protocol implementation remains below the SDK boundary.

Appendix C. Migration Guidelines

C.1 Overview

The QVTM SDK is designed to support long-term evolution while preserving stable public integration contracts.

This appendix summarizes recommended migration strategies for applications adopting newer SDK versions.

C.2 SDK Version Upgrades

Applications SHOULD:

- update to supported SDK releases;
- review release notes;
- verify capability availability;
- execute integration tests before deployment.

Minor SDK upgrades SHOULD preserve public API compatibility.

C.3 Major Version Upgrades

Major SDK versions MAY introduce breaking API changes.

Applications SHOULD:

- migrate incrementally;
 - validate Deployment Profiles;
 - review deprecated APIs;
 - verify compatibility with supported Protocol Specification versions.
-

C.4 Deployment Profile Migration

Applications MAY migrate between Deployment Profiles when supported.

Typical migrations include:

- Login Vault
- Login Secure Messaging
- Vault Secure Upload

The Core SDK object model remains unchanged.

Only profile-specific workflows evolve.

C.5 Protocol Version Migration

Applications SHOULD rely on SDK compatibility mechanisms rather than implementing protocol migration themselves.

Protocol negotiation remains the responsibility of the SDK.

Applications SHOULD consume only observable compatibility results.

C.6 Capability Migration

When new capabilities become available:

- detect capabilities dynamically;
- enable optional features;
- preserve compatibility with older deployments.

Applications SHOULD avoid assuming universal capability availability.

C.7 Deprecated APIs

SDK implementations MAY deprecate APIs before removing them.

Applications SHOULD migrate away from deprecated APIs before upgrading to a future major SDK version.

Deprecated APIs SHOULD NOT be used for new development.

C.8 Testing Recommendations

Before deploying an upgraded SDK, applications SHOULD verify:

- Enrollment;
 - Authentication;
 - Synchronization;
 - Recovery;
 - Deployment Profile behavior;
 - error handling;
 - compatibility with existing deployments.
-

C.9 Backward Compatibility

Applications maximize compatibility by:

- relying only on documented SDK contracts;
 - avoiding implementation-specific behavior;
 - using capability discovery;
 - treating unknown optional metadata as informational.
-

C.10 Summary

Successful SDK migrations follow four principles:

1. Prefer capability discovery over version checks.
2. Depend only on documented public APIs.
3. Validate integrations before deployment.
4. Allow the SDK to manage protocol evolution.

Following these recommendations minimizes migration effort while preserving long-term interoperability across QVTM SDK implementations.
