Universal Manifest v0.4 Specification

PREVIEW

Working Draft — Preview

10 June 2026

This version: https://universalmanifest.net/spec/v0.4-preview/

Latest stable version: https://universalmanifest.net/spec/v0.3/ (same as previous version)

History: https://universalmanifest.net/spec/ (version index and changelog)

Editors: Universal Manifest Working Group

Working Draft — Preview. This document is not stable. It incorporates all normative content from v0.3 and adds sections for features under active development. Sections marked PREVIEW are not final and may change substantially. Working-group input is requested on each preview section.

For the current stable specification, see Universal Manifest v0.3.

Coming from v0.3? See Changes from v0.3 for a summary of what v0.4 adds: the manifest-class model and class registry, the `actorState` session-state field, the two-component `devices` model (`deviceAttestation` + `deviceCapability`), the `trustWeight` field and category-trust claim split, derived-variant sensor-consent vocabulary with purpose binding, the first-class Receipt manifest class, credential-binding verification sub-steps, the JWE baseline algorithm pair, the statusRef resolution protocol, the cryptographic-requirements table, the selective-disclosure dual-track model, and preview sections for ZKP proof profiles, multi-party ceremonies, bilateral sessions, agent delegation scope registry, profile registration, post-quantum signatures, and federation.

Coming from v0.1 or v0.2? See Changes from v0.2 for the v0.2-to-v0.3 migration path, then see Changes from v0.3 for v0.4 additions.

Abstract

This specification defines the **Universal Manifest**, a portable state capsule. A Universal Manifest is defined by an abstract data model — a set of types, properties, and semantics that exist

independently of any single serialization — together with *production rules* that specify how that abstract model is written into a concrete wire format. JSON-LD is the reference encoding used throughout this document and is the default format that conforming implementations are expected to interoperate with; CBOR-LD is defined as a second, compact encoding to demonstrate format independence. Formulated as a hybrid synthesis of web publication metadata and web application parameters, the Universal Manifest provides developers and holders with a standardized envelope to convey linked-data identity references, role permissions, device registrations, and pointers to canonical data sources.

The Universal Manifest is specifically designed for local-first environments (e.g., venue edges, public displays) where evaluators must tolerate partial connectivity and rely on cached, verifiable state. Using this standard, user agents, smart displays, and network edges can securely interoperate without requiring a continuous cloud connection, facilitating seamless cross-context experiences.

Conformance Requirements

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC 2119] [RFC 8174] when, and only when, they appear in ALL CAPITALS, as shown here.

Status of This Document

This section describes the status of this document at the time of its publication.

This is a Working Draft — Preview. It is not stable. Sections marked PREVIEW are under active development and may change substantially. Working-group input is sought on every preview section.

Universal Manifest v0.4 builds on the stable v0.3 specification, which established the evaluation contract, the six-stage evaluation sequence, encrypted inline facets, structured receipts, and the normative baseline for cross-DID binding, `requiredTrustTier`, and agent delegation. Version 0.4 is the production-candidate milestone. Features deferred from v0.3 land in two forms: JWE algorithm constraints (Section 2.4) and statusRef resolution (Section 3.4) arrive as new normative content, while Tier 2 ZKP proof profiles, Tier 3 multi-party ceremonies, the bilateral session model, the agent delegation scope registry, profile registration, post-quantum signatures, and federation are added as preview sections.

Version 0.4 introduces new normative content beyond v0.3 — credential binding (Section 6.6, Section 6.4.9), JWE algorithm constraints (Section 2.4), and the statusRef resolution protocol (Section 3.4) — and adds PREVIEW sections for features under working-group review. All v0.3 normative content is otherwise carried forward. Requirements in PREVIEW sections become final only after working-group review.

Because Universal Manifest is still in the v0.x line, minor-version bumps may still include breaking changes when those changes are reflected by a new version folder. Implementors should review the Changes from v0.3 section and the Changes from v0.2 section for the complete evolution of the specification.

For implementers: the v0.3 specification remains the stable baseline. Use the v0.3 spec, run the Standalone Conformance Suite, and review Conformance v0.3. The preview sections in this document are provided for working-group discussion and forward planning.

Working-group decisions built into this preview — flag to revise.

To give the working group a complete draft to react to rather than a list of open questions, each cross-cutting and wire-shape decision below has been **built into this preview using the editors' recommended default**. Every item is marked PREVIEW in its section and remains fully revisable: the working group can accept, modify, or remove any of them before the v0.4 schema is locked. This list records the choice made and where it lives, so the decisions are transparent. Wire-shape choices are the most time-sensitive, because the v0.4 wire format is not yet fielded — revising one after the schema lock would force a breaking change in a later version.

- **Manifest classes and polymorphic envelope.** *Built on default: add.* A normative section names the polymorphic envelope and the discriminator rule (the facet set determines the class), and an informative appendix lists the manifest classes already in use by integration profiles. See Section 1.0.1 and Appendix A. Non-breaking; every v0.3 manifest stays conformant. *Flag to revise.*
- **Root-level `actorState` field.** *Built on default: add as* OPTIONAL, RECOMMENDED *when `um:agentDelegation` is present.* An optional `actorState` object declares who operates a session (a `principal` plus a human/agent/hybrid `executor`), bridging the `um:agentDelegation` pointer (Section 6.5) to session-state semantics. See Section 1.4.7. *Wire-shape — flag to revise.*
- **`devices` array definition.** *Built on default: define in v0.4.* The `devices` array (Section 1.4.6) is lifted out of its reserved state with a two-component split: `deviceAttestation`

(long-lived, manufacturer-signed) and `deviceCapability` (session-scoped, user-signed, with an ephemeral signing key). *Wire-shape (most consequential) — flag to revise.*

- **Category-trust claim split and `trustWeight`.** *Built on default: adopt `trustWeight` in v0.4.* The single self-declared category field is replaced by four category-trust claim objects (operator identity, service-category claim, category attestation, urgency claim), and the `interpretedAs: "hint-only"` enum is replaced by a typed `trustWeight` field. See Section 6.8. *Wire-shape — flag to revise.*
- **Derived-variant sensor consent vocabulary.** *Built on default: add purpose binding and the derived-variant vocabulary in v0.4.* The `consents` vocabulary (Section 1.4.4) is extended with per-sensor derived-variant keys and per-consent purpose binding using the W3C Data Privacy Vocabulary. See Section 1.4.4.1. Small and non-breaking. *Wire-shape (small) — flag to revise.*
- **Receipt as a first-class manifest class.** *Built on default: promote in v0.4.* The structured receipt (Section 3.3) is promoted to a first-class manifest class with chain-integrity rules (`seq` + `prevHash` + session-scoped signing key), a typed event-class vocabulary, and an optional transparency-log anchoring profile. See Section 3.3.2. Additive. *Flag to revise.*
- **`effectiveTrustTier` receipt field status.** *Built on default: conditional.* `effectiveTrustTier` (Section 3.3.1.1) is a **SHOULD** field conditional on credential-binding support rather than required on every receipt. *Flag to revise (could be made required).*
- **Threshold protocol citations.** *Built on default: cite as non-normative only.* FROST [RFC9591] and threshold BBS+ are cited as non-normative guidance in the Tier 3 ceremony section (Section 6.4.8). *Flag to revise.*
- **Selective-disclosure dual-track model.** *Built on default: SD-JWT baseline, BBS+ for privacy-critical deployments.* The privacy section now defines an explicit dual-track selective-disclosure model. See Section 7.1. *Flag to revise.*
- **Profile placement of preview sections.** *Built on default: kept in core, revisit at lock.* The bilateral session model (Section 3.5) and federation (Section 8) remain in the core specification for now; both are protocol-layer concerns that may move to companion specifications. *Flag to revise at v0.4 lock.*
- **Signature portability across encodings.** *Built on default: per-production signing.* The integrity proof is computed over the bytes of one production, so re-encoding a manifest requires re-signing (Section 1.7.5). Whether a future profile should instead sign a canonicalization of the abstract data model — making one signature portable across encodings — is the strategic open question of the format-independence design. *Flag to revise.*

Per-section preview questions (algorithm pairs, proof-system selection, status-list mechanisms, scope naming, and similar) are raised inline in the relevant sections below, each with the default the editors built in. The list above collects only the cross-cutting and wire-shape decisions that benefit from being reviewed together before the v0.4 schema is locked.

Changes from v0.3

The following additions are made in v0.4. All v0.3 normative content is otherwise carried forward, with the single exception of the deprecated `interpretedAs` member, which is removed and replaced by the typed `trustWeight` field (Section 6.8.1); a v0.3 manifest carrying `interpretedAs` degrades safely (the member is ignored and `trustWeight` defaults to `"hint"`). v0.3 manifests remain parseable by v0.4 evaluators.

Credential binding (new normative content):

- **`holderBinding` on claims** (Section 6.6.1): Cryptographic binding of claims to the manifest subject via SD-JWT key binding, BBS+ holder commitment, or reciprocal DID control. REQUIRED for Tier 1+.
- **`presentationProof`** (Section 6.6.2): Proof-of-possession at verification time, binding the manifest to a specific verifier and moment. Prevents manifest replay.
- **`livenessAttestation`** (Section 6.6.3): Evidence of recent interactive human authentication with four freshness classes.
- **Claim proof process** (Section 6.4.9): End-to-end 7-step verification path for claim trustworthiness, including key-authorization checks against W3C DID Core verification relationships.
- **Credential binding receipt fields** (Section 3.3.1.1): The structured receipt records the outcome of binding verification via `holderBindingStatus`, `presentationProofStatus`, `livenessStatus`, `crossDidBindingStatus`, and `effectiveTrustTier`, making the gap between declared and verified trust explicit.

New normative content:

- **JWE algorithm constraints** (Section 2.4): MUST-implement algorithm pair (ECDH-ES+A256KW / A256GCM) for encrypted facets.
- **statusRef resolution schema** (Section 3.4): HTTP resolution protocol, response schema, status semantics, and error taxonomy for revocation endpoints.

- **Stage-2 credential-binding verification sub-steps** (Section 3.1.2): Sub-steps 2a–2d (holder-binding, presentation-proof, liveness, cross-DID binding) woven into the Verify stage of the evaluation sequence.
- **Cryptographic requirements summary** (Section 6.9): A consolidated three-tier table (mandatory-to-implement, recommended, optional, future) of all algorithms and cryptosuites referenced across the specification.

Envelope and wire-shape additions (PREVIEW — built on editors' recommended defaults):

- **Manifest classes and the polymorphic envelope** (Section 1.0.1, Appendix A): Names the polymorphic envelope, the facet-set discriminator rule, and the profile-extension policy, with an informative class-registry snapshot. Non-breaking; every v0.3 manifest stays conformant.
- **`actorState` member** (Section 1.4.7): Optional root field declaring the session principal and a human/agent/hybrid executor, bridging `um:agentDelegation` to session-state semantics.
- **Two-component `devices` model** (Section 1.4.6): Defines the reserved `devices` array as `deviceAttestation` (long-lived, manufacturer-signed) plus `deviceCapability` (session-scoped, user-signed, with an ephemeral signing key).
- **Derived-variant sensor consent vocabulary** (Section 1.4.4.1): Per-sensor derived-variant consent keys with per-consent purpose binding using the W3C Data Privacy Vocabulary.
- **Receipt as a first-class manifest class** (Section 3.3.2): Adds `seq`/`prevHash` chain integrity, a typed event vocabulary, a structured reason registry, and optional transparency-log anchoring.
- **Category trust and `trustWeight`** (Section 6.8): Replaces the `interpretedAs: "hint-only"` enum with a typed `trustWeight` field and splits the self-declared category into four category-trust claim objects.

Format independence (abstract data model):

- **Abstract data model and production rules** (Section 1.7): The manifest is now defined by a serialization-independent abstract data model plus production rules, following the W3C DID Core pattern. JSON-LD is reframed as the reference encoding (Section 1.7.3) rather than the sole format, and CBOR-LD is defined as a second, compact encoding (Section 1.7.4) to demonstrate format independence. The Abstract and Section 1 are reworded accordingly. No member definitions or wire shapes change for the JSON-LD encoding; v0.3 manifests remain valid JSON-LD productions.
- **Conformance to the abstract data model** (Section 4.5): Conformance is now defined against the abstract data model rather than the JSON-LD shape. Supporting the JSON-LD reference encoding remains sufficient for full conformance.

Expanded profiles (cryptographic and trust profiles):

- **Tier 2 ZKP proof profile** (Section 6.4.7): Dual-profile for cross-DID binding: BBS+ linked-secret proof (Profile 2A) and HD derivation proof via Groth16 (Profile 2B).
- **Tier 3 multi-party ceremony** (Section 6.4.8): Ceremony model with threshold attestation proof, signer-role taxonomy, and non-normative guidance on FROST and threshold BBS+.
- **Post-quantum signatures** (Section 6.7): Candidate algorithms (ML-DSA, SLH-DSA, FN-DSA), dual-signature migration path.
- **Selective-disclosure dual-track model** (Section 7.1): SD-JWT baseline (Track A) and BBS+ unlinkable derived proofs (Track B) for manifest-level selective disclosure.

Infrastructure (federation, registries, session model):

- **Bilateral session model** (Section 3.5): Session objects, paired receipt correlation, exchange identifiers, and session lifecycle states.
- **Agent delegation scope registry** (Section 6.5.3): Namespace convention, six core scope values, and registration procedure.
- **Profile registration mechanism** (Section 5.1): Five-step IANA-style registration process for new profiles.
- **Federation** (Section 8): Resolver coordination, status check distribution, cache invalidation, availability, and manifest forwarding.

Changes from v0.2

The following substantive changes were made between v0.2 and v0.3:

- **Evaluation sequence** (Section 3.1): Replaced the three-step manifest processing model with a normative six-stage evaluation sequence (Arrive, Verify, Project, Consent, Compose, Receipt).
- **Structured receipts** (Section 3.3): Defined a normative receipt schema as the output of the evaluation sequence, with four outcome categories.
- **Encrypted inline facets** (Section 2.3): Added a JWE inline encryption profile for facets, including key rotation and recipient revocation.
- **Cross-DID binding promoted** (Section 6.4.4): Promoted from non-normative convention to normative requirement.

- `requiredTrustTier` **promoted** (Section 6.4.5): Promoted from non-normative convention to normative requirement.
- **Agent delegation promoted** (Section 6.5): Promoted from non-normative convention to normative requirement.
- **Tier 2 defined** (Section 6.4.2): Tier 2 (cryptographic binding via zero-knowledge proof of cross-DID control) is now normatively defined. Marked at risk.
- **Evaluator behavior expanded** (Section 4.1): Added evaluation contract obligations referencing the evaluation sequence.
- **Privacy model expanded** (Section 7): Added sealed-entry and selective-disclosure privacy considerations for encrypted facets.
- **Signature limitations softened** (Section 6.1): Reduced v0.1 caveat language to reflect Signature Profile A delivery.
- **Signature integrity updated** (Section 1.5): Removed "reserved for subsequent iterations" language.
- **Informative references** (Section 9.2): Added URLs to all citations.

1. Universal Manifest

A **Universal Manifest** is a cross-platform data envelope defined by an abstract data model (Section 1.7) and serialized through a production rule for a concrete format. It blends the semantic linkability of a Web Publication Manifest with the applied processing lifecycle of a Web App Manifest. Unless stated otherwise, the examples and member definitions in this section are expressed in the JSON-LD reference encoding (Section 1.7.3); the same manifest MAY be produced in any other conformant encoding, such as CBOR-LD (Section 1.7.4), without changing its meaning.

1.0. Terminology

The following terms are used normatively throughout this specification.

- **manifest** — A Universal Manifest: the portable state capsule defined by this specification.
- **subject** — The primary entity a manifest describes, identified by the `subject` member (Section 1.3.2).

- **holder** — The party that assembles, signs, and presents a manifest. The holder controls selective disclosure (Section 3.1.3) and is the party bound by the `holderBinding` mechanisms (Section 6.6.1).
- **presenter** — The party transmitting a manifest to an evaluator in a given interaction; usually the holder.
- **evaluator** — The conformance class that consumes a manifest and applies the evaluation sequence (Section 3.1). Also called the *verifier*; it is the relying party applying the result. This specification uses "evaluator" uniformly for this single conformance class.
- **issuer** — The party that issued a credential carried as a claim or `claimProof` (Section 6.4.3).
- **attester** — A party that asserts a cross-DID binding or other attestation an evaluator may trust by policy (Section 6.4.4).
- **resolver** — A service that resolves manifest status or coordinates federated status (Section 3.4, Section 8).
- **status endpoint** — The HTTP endpoint a `signature.statusRef` resolves to (Section 3.4).
- **session** — A time-bounded interaction; for bilateral exchanges, the object of Section 3.5.
- **receipt** — The structured, machine-readable record an evaluator produces (Section 3.3).
- **production / consumption** — Serializing the abstract manifest into a wire encoding / parsing it back (Section 1.7.2).
- **sealed entry** — An encrypted facet an evaluator cannot decrypt; recorded as present but not read (Section 3.1.4).
- **trust tier** — One of the four claim-verification tiers (Section 6.4.2). The *achieved* tier (`effectiveTrustTier`, Section 3.3.1.1) is distinct from a declared `requiredTrustTier` floor and from the *interaction tier floor* negotiated in a bilateral exchange (Section 6.4.6).
- **interactive presentation** — A presentation made in response to a verifier-issued `challenge` (Section 6.6.2).

1.0.1. Manifest Classes and the Polymorphic Envelope PREVIEW

The Universal Manifest is a **polymorphic envelope**: a single fixed envelope shape carries many different kinds of payload. The envelope members defined in this specification (Section 1.2), the identity and lifespan members (Section 1.3), the structural state members (Section 1.4), and the integrity proof (Section 1.5) are common to every manifest. What a particular manifest *is* — an identity capsule, a device-capability descriptor, a consent record, a receipt, and so on — is determined by which facets, claims, consents, pointers, and reserved members

it carries, not by a distinct schema per kind. This is the same extensibility model that v0.3 already enables; v0.4 names it and states the rule explicitly.

1.0.1.1. Class Discriminator Rule

A **manifest class** is a named, profile-defined combination of structural members that an evaluator can recognize and act on. The discriminator for a class is its **characteristic facet set**: the presence of specific facet `@type` values, claim `@type` values, named structural members (for example, a populated `devices` array, or a receipt's `seq`/`prevHash` chain members), and any additional top-level `@type` values a class profile declares alongside `um:Manifest` (for example `um:Receipt`, Section 3.3.2). Evaluators **MUST NOT** rely on the manifest's top-level `@type` alone to determine a class beyond the required `um:Manifest` value; the discriminating members are the authoritative signal. A manifest **MAY** satisfy more than one class simultaneously (for example, an identity capsule that also carries a device-capability facet).

Because classes are discriminated by member presence rather than by a closed type enumeration, the envelope remains **non-breaking and forward-compatible**: an evaluator that does not recognize a class processes the members it understands and records the rest as present-but-unprocessed, exactly as it handles unknown facets, claims, and pointers elsewhere in this specification (Section 3.1.1). Every v0.3-conformant manifest is, by this rule, a valid v0.4 manifest of one or more classes.

1.0.1.2. Profile Extension Policy

New manifest classes are defined by **profile documents**, registered through the profile registration mechanism (Section 5.1) under the `domain` category. A manifest-class profile **MUST** specify: the class name and canonical profile identifier; the discriminating facet set (the members whose presence identifies the class); any class-specific required and optional members beyond the common envelope; and the evaluator obligations for the class. A class profile **MUST NOT** redefine, remove, or change the meaning of any common envelope member defined in this specification; it may only add class-specific members and constrain their use. An informative snapshot of the manifest classes currently surfaced by integration profiles is provided in Appendix A.

> **Preview:** Naming the polymorphic envelope and the class-discriminator rule is built into this preview on the editors' recommended default (add; non-breaking). The registry mechanism itself is a v0.4 deliverable. Working-group input is requested on whether the discriminator should remain member-presence-based or also admit an explicit optional `manifestClass` hint member.

1.1. Examples

The following is an example of a minimal Universal Manifest:

Example 1: Minimal Universal Manifest payload

```
{
  "@context": [
    "https://universalmanifest.net/ns/v0.4"
  ],
  "@id": "urn:uuid:123e4567-e89b-12d3-a456-426614174000",
  "@type": ["um:Manifest"],
  "manifestVersion": "0.4",
  "subject": "did:example:123",
  "issuedAt": "2026-03-01T00:00:00Z",
  "expiresAt": "2026-03-02T00:00:00Z",
  "signature": {
    "algorithm": "Ed25519",
    "canonicalization": "JCS-RFC8785",
    "keyRef": "did:key:z6MkExample#keys-1",
    "publicKeySpkiB64": "MCowBQYDK2VwAyEA...",
    "created": "2026-03-01T00:00:00Z",
    "value": "base64url-encoded-signature-bytes"
  }
}
```

> **Note:** The `signature` values above are illustrative. See Section 1.6.2 for the full signature shape definition and Section 1.6.3 for the signing input procedure.

1.2. Core Identity Members (JSON-LD Reference Encoding)

The three properties below — the manifest's **contextReference**, **id**, and **type** (Section 1.7.1) — identify and semantically anchor every manifest. They are defined here in the JSON-LD reference encoding (Section 1.7.3), where they are expressed using the JSON-LD keywords `@context`, `@id`, and `@type`. The requirements stated below apply to the corresponding abstract properties in any conformant encoding; a compact encoding such as CBOR-LD (Section 1.7.4) carries the same three properties in its own form (for example, the context reference is carried as a compression context identifier rather than an inline term list).

1.2.1. `@context` member (abstract property `contextReference`)

The `@context` member establishes the semantic definitions of terms used within the manifest. It **MUST** include the Universal Manifest namespace for the specification version being implemented (e.g., `https://universalmanifest.net/ns/v0.4` for this version).

The namespace URI is versioned. Manifests conforming to this specification **MUST** use `https://universalmanifest.net/ns/v0.4`. Evaluators that support an earlier version **MAY** process manifests declaring it; an evaluator processing a v0.3 manifest under this specification applies the v0.4 evaluation sequence with the compatibility rules of Section 6.6.7.

The `um:` prefix used throughout this specification (in `@type` values such as `um:Manifest` and in registry identifiers such as `um:reason:`) expands to the IRI `https://universalmanifest.net/ns/um#`. The full term definitions for a given version are fixed by that version's context document, served at the versioned namespace URI; this specification version's context document is reproduced, with its content hash, in Appendix B, and that pinned copy is normative for context integrity (Section 6.10).

1.2.2. `@id` member (abstract property `id`)

Holders **MUST** generate an `@id` member as a globally unique opaque identifier (e.g., `urn:uuid:<uuidv4>`). The `@id` value **MUST NOT** contain personally identifiable information.

1.2.3. `@type` member (abstract property `type`)

The `@type` member indicates the document type classifying the resource. It **MUST** include the value `um:Manifest`.

1.3. Identities & Lifespans

1.3.1. `manifestVersion` member

`manifestVersion` (string, **REQUIRED**): The version of the Universal Manifest specification this manifest conforms to. For v0.4 conformant manifests, the value **MUST** be `"0.4"`. Evaluators **MUST** check that they support the declared version before processing the manifest through the evaluation sequence. "Support the declared version" means having a defined processing path for it — either native support or, for v0.3, the compatibility path of Section 6.6.7.

1.3.2. `subject` member

The `subject` member is **REQUIRED**. It specifies the primary entity (user, app, venue) the manifest describes. It **MUST** contain a stable identifier URI (e.g., a Decentralized Identifier / DID).

1.3.3. `issuedAt` and `expiresAt` members

Both `issuedAt` and `expiresAt` are **REQUIRED**. They formulate the bounding constraints (TTL) for the manifest's validity. All date-time values in this specification **MUST** conform to the RFC 3339 [RFC3339] `date-time` production and **SHOULD** use UTC (`Z`).

1.4. Structural State Members

The manifest structure relies on domain-specific members akin to Web Publication linkages.

1.4.1. `facets` member

The `facets` member organizes extended functional blocks (`um:Facet`), packaging specific verifiable capabilities, metadata subsets, or configuration modules. The abstract `facets` property is a list of facet objects; when present, it **MUST** be expressed as a JSON array in the JSON-LD reference encoding (and as the corresponding list in any other production rule).

1.4.2. Structural State Arrays (`claims`, `consents`, `pointers`, `devices`)

These arrays group specific operational contexts representing permissions, deployed hardware targets, and external data reference pointers connected to the `subject`. Each array is a top-level

structural member of the manifest. The following subsections define the base schema for each array's entries.

All structural state members (`facets`, `claims`, `consents`, `devices`, `pointers`) are **OPTIONAL**. When absent, evaluators **MUST** treat them as empty arrays.

The manifest **MAY** also include the following top-level member:

- `requiredTrustTier` — An integer (0–3) specifying the minimum trust tier an evaluator **MUST** satisfy before acting on this manifest. This field is **OPTIONAL**; when absent, the default is 0 (no minimum required). See Section 6.4.5 for the trust-tier evaluation algorithm.

1.4.3. `claims` Array Schema

The `claims` array contains zero or more claim objects. Each claim object represents a statement about the manifest `subject`, issued by the manifest signer or by an external party. Evaluators process claims according to the tiered trust model (Section 6.4.2) and the evaluation sequence (Section 3.1).

A base claim object **MUST** contain the following fields:

- `@type` — A string identifying the claim type. **MUST** be present. Specialized claim types (e.g., `"identity.crossDidBinding"`) use this field to declare their schema.
- `issuer` — A DID or URI identifying the entity that asserts the claim. **MUST** be present. When the manifest signer is the issuer, this field **MUST** match the manifest `subject` or the signing key's controller.

A base claim object **MAY** contain the following fields:

- `@id` — An opaque identifier for this claim entry. **RECOMMENDED** when receipts or warnings reference the claim (e.g., `affectedClaims`, `claimStatuses` in Section 3.3.1).
- `subject` — DID or URI of the entity the claim is about. When absent, the manifest-level `subject` is implied.
- `issuedAt` — RFC 3339 date-time when the claim was issued.
- `expiresAt` — RFC 3339 date-time when the claim expires. Evaluators **SHOULD** reject expired claims.
- `claimProof` — A Verifiable Presentation, attestation proof object, URI reference, or array of proof entries enabling Tier 1 verification (see Section 6.4.3). When an array, each entry is

independently verifiable and **MAY** carry its own `proofType`, `proofPurpose`, `verificationMethod`, and `statusRef` fields.

- `holderBinding` — An object declaring how this claim is cryptographically bound to the manifest subject. **REQUIRED** for Tier 1 and above; **OPTIONAL** for Tier 0. The `mode` field **MUST** be one of `"sd-jwt-kb"`, `"bbs-holder-commitment"`, or `"reciprocal-control"`. See Section 6.6 for the binding verification procedure and mode-specific required fields.
- `requiredTrustTier` — Integer (0–3) declaring the minimum trust tier for this claim (see Section 6.4.5).

Specialized claim types extend the base schema by adding type-specific fields. The `@type` field determines which additional fields are expected. Evaluators encountering an unrecognized `@type` **MUST** treat the claim as unprocessable (present but unverifiable at any tier above Tier 0). The `identity.crossDidBinding` claim type is fully defined in Section 6.4.4.

Example 2: Base claim object

```
{
  "claims": [
    {
      "@type": "membership.organization",
      "issuer": "did:web:example-org.com",
      "subject": "did:key:z6MkAlice",
      "issuedAt": "2026-03-01T00:00:00Z",
      "expiresAt": "2027-03-01T00:00:00Z"
    },
    {
      "@type": "identity.crossDidBinding",
      "issuer": "did:web:verify.example",
      "boundDids": ["did:key:z6MkAlice", "did:plc:alice-bsky"],
      "attester": "did:web:verify.example",
      "attestationMethod": "AT Protocol handle resolution",
      "attestedAt": "2026-03-15T10:30:00Z",
      "claimProof": "https://verify.example/attestations/abc123"
    }
  ]
}
```

1.4.4. `consents` Array Schema

The `consents` array contains zero or more consent entry objects. Each consent entry records a permission grant governing how an evaluator may act on specific facets. The Consent stage of the evaluation sequence (Section 3.1.4) uses these entries to determine whether processing a facet is authorized.

A consent entry **MUST** contain the following fields:

- `@id` — A URI uniquely identifying this consent entry. **MUST** be present. Consent entries are identified by `@id` for receipt recording.
- `@type` — **MUST** include `"um:Consent"`; **MAY** be a bare string when it is the sole type (per the `@type` shape convention in Section 2.1).
- `facetRef` — A string matching the `@id` of the facet this consent governs. This is the linking mechanism between a consent entry and its target facet. An evaluator determines which consent entry governs a facet by matching `facetRef` to the facet's `@id`.
- `scope` — An array of scope strings defining what operations are authorized (e.g., `"read"`, `"display"`, `"cache"`, `"process"`, `"forward"`). These scope strings are deployment-defined; this specification does not enumerate a closed set. Evaluators **MUST NOT** perform any operation that is not literally present in `scope` (fail closed on unrecognized or absent scopes).
- `purpose` — A string declaring the stated purpose for the data use (e.g., `"session-personalization"`, `"age-verification"`, `"access-control"`). Evaluators **MUST** verify that their intended use falls within the declared purpose before processing the facet. Purpose comparison is exact string equality unless both parties support a shared vocabulary (for example the W3C Data Privacy Vocabulary [DPV]), in which case an evaluator **MAY** treat a narrower stated use as falling within a broader granted purpose per that vocabulary's hierarchy; deployments doing so **MUST** document it in their conformance claim.
- `grantedAt` — RFC 3339 date-time when the consent was granted.
- `expiresAt` — RFC 3339 date-time when the consent expires. Evaluators **MUST** treat expired consent as absent consent.

A consent entry **MAY** contain the following fields:

- `grantor` — DID or URI of the entity who granted consent. When absent, the manifest `subject` is implied.
- `withdrawnAt` — RFC 3339 date-time when consent was withdrawn. When this field is present, the consent is no longer active regardless of `expiresAt`. Evaluators **MUST** treat a consent entry with a `withdrawnAt` value as absent consent.

- `conditions` — An array of condition strings imposing additional constraints (e.g., `"offline-only"`, `"no-third-party-sharing"`). Condition strings are deployment-defined. Evaluators **MUST NOT** process a facet governed by a consent entry carrying a condition they do not recognize or cannot enforce (fail closed, consistent with `scope` handling).

When a facet has no matching consent entry in the `consents` array (i.e., no entry whose `facetRef` matches the facet's `@id`), the evaluator **MUST** record the facet status as `"consent-missing"` in the receipt and **MUST NOT** process the facet's data.

When the `consents` array is empty or absent, no facets carry consent records. Evaluators **MUST** treat all facets as lacking consent in this case, unless the manifest's deployment context operates under a consent model external to the manifest (e.g., a pre-negotiated bilateral agreement). Such external consent models are outside the scope of this specification.

Example 3: Consent entry

```
{
  "consents": [
    {
      "@id": "urn:uuid:consent-public-profile-001",
      "@type": "um:Consent",
      "facetRef": "urn:uuid:facet-public-profile-001",
      "scope": ["read", "display", "cache"],
      "purpose": "session-personalization",
      "grantedAt": "2026-04-01T09:00:00Z",
      "expiresAt": "2026-04-01T18:00:00Z"
    },
    {
      "@id": "urn:uuid:consent-health-data-002",
      "@type": "um:Consent",
      "facetRef": "urn:uuid:facet-health-data-002",
      "scope": ["read"],
      "purpose": "age-verification",
      "grantor": "did:key:z6MkAlice",
      "grantedAt": "2026-04-01T09:00:00Z",
      "expiresAt": "2026-04-01T12:00:00Z",
      "conditions": ["no-third-party-sharing"]
    }
  ]
}
```

1.4.4.1. Derived-Variant Sensor Consent Vocabulary PREVIEW

Spatial-computing sensors expose data at very different levels of sensitivity depending on how much processing is applied. A single boolean "eye-tracking allowed" consent cannot distinguish "raw gaze vector stream" from "region-of-interest hit only." v0.4 extends the existing `consents` vocabulary with **derived-variant** consent keys: each sensor class is qualified by a derivation tier so that holders consent to the minimum-sensitivity variant a use case actually needs. This models sensor consent at a granularity no other privacy-credential specification currently addresses, and it is non-breaking: the v0.1 boolean keys remain valid and are interpreted as the most-permissive variant only when a deployment explicitly opts in; absent an explicit grant for a derived variant, evaluators MUST fail closed.

A derived-variant consent key uses dot-separated `sensor.<class>.<signal>.<derivation>` naming, aligned with the Khronos OpenXR sensor classes [OPENXR]. Eight sensor classes (eye, hand, face, body, depth, audio, rgb-camera, and environment) are each qualified by a derivation tier from `raw` (least processed, most sensitive) through progressively more-derived variants (for example `roi-derived`, `event-derived`, `presence-derived`). Examples:

- `sensor.eye.gaze.raw` — raw gaze vector stream.
- `sensor.eye.gaze.roi-derived` — region-of-interest hit-test results only, no raw gaze.
- `sensor.hand.pose.raw` — full hand skeleton stream.
- `sensor.hand.pose.gesture-derived` — recognized discrete gestures only.

Each derived-variant consent entry MUST carry the `purpose` field defined in Section 1.4.4, binding the grant to a stated purpose so that a grant for one purpose cannot be silently reused for another (consent-creep prevention, per GDPR Article 5(1)(b) and ISO/IEC 27560:2023). The `purpose` value SHOULD be drawn from the W3C Data Privacy Vocabulary (DPV) where an applicable term exists, so that purposes are machine-comparable across deployments; deployment-specific purposes MAY be used where DPV has no suitable term.

An evaluator MUST match the most specific derived-variant key granted. A grant for `sensor.eye.gaze.roi-derived` MUST NOT be read as granting `sensor.eye.gaze.raw`. A grant for a less-derived (more sensitive) variant MAY be treated by policy as also authorizing strictly more-derived variants of the same signal, but a deployment that does so MUST document the implication in its conformance claim.

Example 4: Derived-variant sensor consent with purpose binding

```
{
  "consents": [
    {
      "@id": "urn:uuid:consent-gaze-roi-001",
      "@type": "um:Consent",
      "facetRef": "urn:uuid:facet-eye-roi-001",
      "scope": ["read", "process"],
      "sensorConsent": "sensor.eye.gaze.roi-derived",
      "purpose": "dpv:UserInterfacePersonalisation",
      "grantedAt": "2026-06-09T09:00:00Z",
      "expiresAt": "2026-06-09T18:00:00Z"
    }
  ]
}
```

Preview: Per-consent purpose binding and the derived-variant sensor vocabulary are built into this preview on the editors' recommended default (add in v0.4; small, non-breaking). Working-group input is requested on the full derivation-tier enumeration per sensor class and on whether the sensor key is best carried as the dedicated `sensorConsent` field shown above or folded into `scope`.

1.4.5. `pointers` Array Schema

The `pointers` array contains zero or more pointer objects. Each pointer references an external data source, delegation relationship, or canonical resource connected to the manifest `subject`. Pointers are typed; the `@type` field determines the pointer's semantics and expected fields.

A base pointer object **MUST** contain the following fields:

- `@type` — A string identifying the pointer type. **MUST** be present. Defines which additional fields are expected and how the evaluator processes the pointer.
- `target` — A URI referencing the external resource, endpoint, or entity this pointer connects to. **MUST** be present.

Pointer types **MAY** define type-specific fields that replace the base `target` requirement. When a pointer type declares such a replacement, its type-specific fields take precedence.

A base pointer object MAY contain the following fields:

- `@id` — An opaque identifier for this pointer entry.
- `label` — A human-readable label describing the pointer's purpose.
- `createdAt` — RFC 3339 date-time when the pointer was created.
- `expiresAt` — RFC 3339 date-time when the pointer expires. Evaluators SHOULD ignore expired pointers.

The following pointer types are defined in this specification:

- `um:agentDelegation` — Declares delegated session authority. See Section 6.5 for the complete schema. The `um:agentDelegation` type replaces the base `target` requirement with the type-specific fields defined in Section 6.5.1.

Evaluators encountering an unrecognized pointer `@type` MUST record the pointer as present in the receipt but MUST NOT act on it. Future versions of this specification or extension profiles MAY define additional pointer types.

1.4.6. `devices` Array Schema PREVIEW

The `devices` array registers hardware endpoints (e.g., XR headsets, NFC readers, smart displays, wearable sensors) associated with the manifest `subject`. v0.3 reserved this member without defining its entries. v0.4 defines device entries as a **two-component split** that mirrors the layered device-attestation models of FIDO, TPM 2.0, WebAuthn Level 3 [WEBAUTHN], Android Hardware Attestation, and Apple DeviceCheck: a long-lived hardware provenance component and a session-scoped capability component. Separating them lets a session-only manifest advertise device capabilities without leaking a long-lived hardware identifier.

Each entry in the `devices` array is a device object that MAY carry a `deviceAttestation` component, a `deviceCapability` component, or both. A device object MUST include a `@type` of `"um:Device"` and an `@id` identifying the entry within the manifest. The `deviceAttestation` and `deviceCapability` sub-objects are device *components*, not `um:Facet` objects (Section 2.1); the term "facet" is reserved for top-level `um:Facet` objects.

1.4.6.1. `deviceAttestation` Component (long-lived, manufacturer-signed)

The `deviceAttestation` component carries hardware provenance that changes rarely and is signed by a manufacturer or platform attestation authority. When present it MUST contain:

- `deviceClass` — A string identifying the hardware category (e.g., `"xr-headset"`, `"nfc-reader"`, `"wearable-sensor"`).
- `modelHash` — A hash identifying the device model/firmware baseline, suitable for matching against a manufacturer registry without revealing a per-unit serial.
- `attestation` — A Verifiable Credential or platform attestation object signed by the manufacturer or attestation authority, conveying the hardware-rooted attestation (e.g., a WebAuthn/FIDO device attestation [WEBAUTHN] or a TPM 2.0 [TPM2] quote). Evaluators **MUST** verify this attestation against a configurable manufacturer trust anchor before relying on the device's hardware claims.

The `deviceAttestation` component **MAY** additionally carry `manufacturer` (DID or URI) and `attestedAt` (RFC 3339). Because hardware identifiers are long-lived and potentially correlatable, holders **SHOULD** omit the `deviceAttestation` component from session-only or pseudonymous manifests and carry only `deviceCapability` (see Section 7).

1.4.6.2. `deviceCapability` Component (session-scoped, user-signed)

The `deviceCapability` component describes what the device can do for the current session and is signed by the subject (or a session-scoped key the subject controls), not by the manufacturer. When present it **MAY** contain:

- `sensors` — An array of sensor-class identifiers the device exposes (e.g., `"eye-tracking"`, `"hand-tracking"`, `"depth"`, `"rgb-camera"`), using the sensor-class naming aligned with the Khronos OpenXR sensor classes (see Section 1.4.4.1).
- `openxrExtensions` — An array of OpenXR extension identifiers the device/runtime supports, referencing the Khronos OpenXR extension registry.
- `positioningTier` — A string declaring the spatial-tracking capability (e.g., `"3dof"`, `"6dof"`, `"world-scale"`).
- `processingConstraints` — An object describing relevant compute/thermal/power limits an evaluator should respect.
- `privacyModes` — An array of privacy-mode identifiers the device can enforce (e.g., `"on-device-only"`, `"roi-derived-only"`).
- `sessionSigningKey` — A session-scoped ephemeral public key (JWK or DID URL) used to sign per-session device assertions. This key **MUST NOT** be a long-lived hardware identifier; it is generated per session to avoid cross-session linkage.

Evaluators **MUST NOT** treat `deviceCapability` claims as hardware-attested unless a corresponding verified `deviceAttestation` component is present on the same device entry. `deviceCapability` is a self-declared, session-scoped assertion (analogous to a hint-weight claim; see Section 6.8) absent such attestation.

Example 5: Device entry with both components

```
{
  "devices": [
    {
      "@id": "urn:uuid:device-headset-1",
      "@type": "um:Device",
      "deviceAttestation": {
        "deviceClass": "xr-headset",
        "modelHash": "sha256:9f2c...",
        "manufacturer": "did:web:headset-vendor.example",
        "attestedAt": "2026-06-01T00:00:00Z",
        "attestation": { "@type": "VerifiableCredential" }
      },
      "deviceCapability": {
        "sensors": ["eye-tracking", "hand-tracking", "depth"],
        "openxrExtensions": ["XR_EXT_eye_gaze_interaction"],
        "positioningTier": "6dof",
        "privacyModes": ["roi-derived-only"],
        "sessionSigningKey": "did:key:z6MkSessionEphemeral#keys-1"
      }
    }
  ]
}
```

The `devices` array is part of the signed payload: holders include it when signing, and evaluators **MUST** include it when recomputing the signing input (Section 1.6.3) and **MUST NOT** discard it before verification. Evaluators **MUST** carry it through the evaluation sequence without modification, and **MUST NOT** discard the `devices` array during the Arrive stage's unknown-field handling, because it is a named structural member. Evaluators that do not implement the device components **MUST** still preserve the array and record device entries as present-but-unprocessed.

> **Preview:** The two-component device split is built into this preview on the editors' recommended default (define in v0.4), because the wire shape is not yet fielded and the split is the most consequential wire-shape decision in this version. Attestation transparency logs for device attestations are out of scope for v0.4 and flagged for v0.5. Working-group input is requested on the sensor-class and OpenXR-extension vocabularies and on the session-key generation requirements.

1.4.7. `actorState` member PREVIEW

The `actorState` member is an **OPTIONAL** top-level object declaring who is operating the session the manifest is presented in: the human principal, a delegated agent, or a hybrid of the two. It bridges the `um:agentDelegation` pointer (Section 6.5), which declares that delegation *exists*, to the session-state semantics of who is *currently acting*. `actorState` is **RECOMMENDED** whenever an `um:agentDelegation` pointer is present.

When present, `actorState` **MUST** contain:

- `principal` — The DID of the human or legal entity the manifest represents. It **MUST** match the manifest `subject`. An evaluator that finds `actorState.principal` not equal to `subject` **MUST** record this as a verification failure and **MUST NOT** rely on the manifest for delegated-authority decisions.

When present, `actorState` **MAY** contain an `executor` object describing who is operating on the principal's behalf:

- `type` — One of `"human"`, `"agent"`, or `"hybrid"`. **MUST** be present when `executor` is present.
- `delegateId` — The DID of the operating agent. **REQUIRED** when `type` is `"agent"` or `"hybrid"`.
- `delegationRef` — A reference to the `um:agentDelegation` pointer (by `@id`) that authorizes this executor. When present, evaluators **MUST** confirm the referenced pointer exists, is unexpired, and names `delegateId` as its delegate.
- `lastVerifiedAt` — RFC 3339 date-time of the last interactive verification of the principal. Meaningful only when `type` is `"human"` or `"hybrid"`; relates to the liveness model in Section 6.6.3.

When `executor` is absent, evaluators **MUST** treat the session as principal-operated (`type: "human"` with no delegate). `actorState` is additive and non-breaking: v0.3 manifests omit it, and evaluators that do not implement it record it as present-but-unprocessed.

Example 6: actorState with an agent executor

```
{
  "subject": "did:key:z6MkAlice",
  "actorState": {
    "principal": "did:key:z6MkAlice",
    "executor": {
      "type": "agent",
      "delegateId": "did:key:z6MkAgentBot",
      "delegationRef": "urn:uuid:pointer-delegation-1",
      "lastVerifiedAt": "2026-06-09T09:00:00Z"
    }
  }
}
```

Preview: `actorState` is built into this preview on the editors' recommended default (add as **OPTIONAL**, **RECOMMENDED** with `um:agentDelegation`). It is a wire-shape addition. Working-group input is requested on whether `actorState` should become **REQUIRED** when any delegation pointer is present.

1.5. Signature Integrity

1.5.1. `signature` member

The `signature` member carries cryptographic proof that the manifest payload has not been tampered with since issuance. Every v0.4 conformant manifest **MUST** include a `signature` member conforming to the JCS + Ed25519 Signature Profile (Signature Profile A, Section 1.6). Future spec versions **MAY** introduce additional normative profiles; evaluators **MUST** reject manifests whose signature profile they do not support.

Unsigned manifests **MAY** exist as development artifacts but are not v0.4 conformant and **MUST** be rejected by conformant evaluators.

1.6. Signature Profile A: JCS + Ed25519

Signature Profile A is the baseline normative signature profile. This profile constrains the `signature` member to a deterministic, portable format suitable for local-first verification on constrained devices.

> **Note:** This profile was introduced in v0.2 and remains the required baseline in v0.3 and v0.4.

Signature profiles are additive. Future versions MAY introduce additional profiles (e.g., post-quantum algorithms or W3C Data Integrity proofs). Evaluators verify the profiles they support; supplementary proof material from an unknown profile carried alongside a supported signature (for example, a `postQuantumSignature` during the dual-signature migration, Section 6.7.3) is safely ignored. A manifest whose only signature uses an unsupported profile is rejected (Section 1.5.1, Section 1.6.5).

1.6.1. Canonicalization and Algorithm

Signature Profile A uses **JSON Canonicalization Scheme** (JCS, RFC 8785) for canonicalization and **Ed25519** [RFC8032] for signing. Signature values are encoded as **base64url** [RFC4648].

This combination provides deterministic signing input without requiring JSON-LD expansion or RDF canonicalization. Signature Profile A is defined against the JSON-LD reference encoding (Section 1.7.3); it signs the canonical bytes of that production, in keeping with the specification's "state capsule" usage. Other production rules either reuse a profile defined against the abstract model or specify their own byte-level signing rule (Section 1.7.5).

1.6.2. Signature Shape

The `signature` object MUST contain the following fields for this profile:

- `signature.algorithm` — MUST be `"Ed25519"`.
- `signature.canonicalization` — MUST be `"JCS-RFC8785"`.
- `signature.keyRef` — URI reference to verification key material (recommended: DID URL or HTTPS URL). MUST be present.
- `signature.value` — base64url-encoded Ed25519 signature over the canonical bytes.

The following fields are OPTIONAL:

- `signature.publicKeySpkiB64` — base64-encoded SPKI DER public key bytes for offline/fixture/local-first verification.
- `signature.created` — RFC 3339 date-time indicating when the signature was produced.
- `signature.statusRef` — URI to status material for this manifest instance (the manifest identified by its `@id`, echoed as `manifestId` in status responses; see Section 3.4). It conveys the status of the manifest, not of any key.
- `signature.revocationCursor` — monotonic status cursor/version string for cache-aware revocation checks.

Additional fields **MAY** exist for future profiles, but evaluators **SHOULD** rely on `algorithm` + `canonicalization` to decide whether they can verify a given signature.

The `signature` property is **not included** in the signing input to avoid circularity. The fields `statusRef` and `revocationCursor`, when present, are metadata for revocation-aware policy checks and do not alter the signing input.

Example 7: Signature Profile A object

```
{
  "signature": {
    "algorithm": "Ed25519",
    "canonicalization": "JCS-RFC8785",
    "keyRef": "did:key:z6MkAlice#keys-1",
    "publicKeySpkiB64": "MCowBQYDK2VwAyEA...",
    "created": "2026-04-01T10:00:00Z",
    "value": "base64url-encoded-signature-bytes"
  }
}
```

1.6.3. Signing Input Procedure

To compute the signature for this profile:

1. Start with the complete JSON-LD production of the manifest (the reference encoding, Section 1.7.3).
2. Remove the `signature` property entirely. Also remove the `presentationProof` property if present (it is a verification-time proof computed over the signing-input hash and is therefore

excluded from the signing input; see Section 6.6.2), and the `postQuantumSignature` property if present (both integrity proofs are computed over the same signing input; see Section 6.7.3, PREVIEW).

3. Canonicalize the remaining object using JCS (RFC 8785), producing a UTF-8 byte sequence.
4. Compute the Ed25519 signature over those bytes.
5. Set the `signature` property on the manifest with the fields defined above.

This yields a stable, portable verification input for any implementation that supports JCS + Ed25519.

1.6.4. Evaluator Checklist

An evaluator implementing this profile **MUST**:

1. Confirm the document is a v0.x Universal Manifest (required fields present and `@type` includes `um:Manifest`).
2. Enforce TTL: reject for use if `now > expiresAt`; sanity-check `issuedAt <= expiresAt`.
3. Determine signature profile support: require `signature.algorithm === "Ed25519"`, `signature.canonicalization === "JCS-RFC8785"`, and a non-empty `signature.value`.
4. Resolve the verification key. The evaluator **MUST** attempt to resolve `keyRef` (method-specific; note that some methods, e.g. `did:key`, resolve locally without network access).
   a. If resolution succeeds and `publicKeySpkiB64` is present, the resolved key **MUST** be byte-identical to the decoded `publicKeySpkiB64`; on mismatch, reject with outcome `rejected` and `signatureCheck: "invalid"`.
   b. If resolution succeeds and `publicKeySpkiB64` is absent, use the resolved key.
   c. If resolution is unavailable (offline or endpoint unreachable) and `publicKeySpkiB64` is present, the evaluator **MAY** verify against the embedded key but **MUST** record `keyRefResolution: "unresolved"` in the receipt, **MUST NOT** grant trust above Tier 0 on the basis of `keyRef`'s identity, and **SHOULD** re-validate when connectivity returns.
   d. If resolution is unavailable and `publicKeySpkiB64` is absent, the manifest cannot be verified and **MUST** be rejected for use (and **MAY** be retained for retry).
5. Recompute the signing input (remove `signature`, JCS canonicalize).
6. Verify the Ed25519 signature over the canonical bytes.

If verification fails, the manifest **MUST** be rejected for use (but **MAY** be retained for debugging).

> **Security Note:** Without the key-matching check in step 4(a), a malicious holder can bundle a `keyRef` pointing to a high-reputation DID while supplying their own key material in `publicKeySpkiB64`, enabling key substitution attacks. The offline path 4(c) does not skip this protection — it caps identity assurance rather than blocking verification: an evaluator that cannot resolve `keyRef` can still confirm payload integrity against the embedded key, but **MUST NOT** attribute the `keyRef` identity to that key (recording `keyRefResolution: "unresolved"`) until resolution confirms the binding. This preserves local-first verification (Abstract) without reintroducing the substitution vector as an identity claim.

1.6.5. Profile Identification

Evaluators **MUST** treat the pair `signature.algorithm` + `signature.canonicalization` as the explicit profile identity.

- If the pair is unsupported, the manifest **MUST** be rejected, recording `signatureCheck: "unsupported-profile"` in the receipt (Section 3.3.1).
- Evaluators **MUST NOT** reinterpret unknown pairs as the baseline profile.

1.6.6. Revocation-Aware Verification Extension

For evaluators claiming revocation-aware verification:

1. If `signature.statusRef` is present, resolve status from that URI (or a configured equivalent).
2. If `signature.revocationCursor` is present, use it to prevent stale-status acceptance and to drive cache revalidation policy.
3. If revocation status cannot be determined and policy requires active status, the manifest **MUST** be rejected for use.

Evaluators that do not implement revocation-aware verification **MUST** report revocation status as `unchecked` and **MUST NOT** claim revocation-aware conformance.

1.7. Abstract Data Model and Production Rules

A Universal Manifest is defined in two layers. The **abstract data model** specifies the manifest's types, properties, and semantics independently of any serialization. A **production rule** specifies how that abstract model is written into — and read back from — a concrete wire format. This

separation, modeled on the W3C Decentralized Identifiers (DID) Core architecture [DID-CORE], allows the Universal Manifest to gain new encodings over time without changing what a manifest *means*.

Throughout the rest of this specification, member definitions and examples are presented in the JSON-LD reference encoding. This is an editorial convenience, not a normative restriction: every requirement stated in terms of a JSON-LD member applies to the corresponding abstract property in any conformant encoding.

1.7.1. Abstract Data Model

The abstract data model describes a manifest as a *map* of abstract properties to values, using the format-agnostic core data types of [INFRA]: maps, lists, sets, datetimes, strings, integers, booleans, and null. The following abstract properties constitute a Universal Manifest. The names below are the canonical abstract property names; each production rule defines how a name is expressed in its format.

- **type** (set of strings, REQUIRED) — The document's type classification. MUST include the manifest type `um:Manifest`. Expressed as `@type` in JSON-LD.
- **id** (string, REQUIRED) — A globally unique opaque identifier for the manifest instance (see Section 1.2.2). Expressed as `@id` in JSON-LD.
- **contextReference** (list, REQUIRED) — A reference to the term definitions that fix the meaning of the manifest's properties for the specification version in use (see Section 1.2.1). Expressed as `@context` in JSON-LD; carried as a compression context reference in CBOR-LD.
- **manifestVersion** (string, REQUIRED) — The specification version the manifest conforms to (see Section 1.3.1).
- **subject** (string, REQUIRED) — A stable identifier URI for the primary entity the manifest describes (see Section 1.3.2).
- **issuedAt**, **expiresAt** (datetime, REQUIRED) — The validity bounds (TTL) of the manifest (see Section 1.3.3).
- **facets** (list, OPTIONAL) — Composable functional blocks (see Section 1.4.1 and Section 2).
- **claims**, **consents**, **pointers**, **devices** (lists, OPTIONAL) — The structural state arrays (see Section 1.4.2).
- **requiredTrustTier** (integer, OPTIONAL) — The minimum trust tier an evaluator must satisfy (see Section 6.4.5).

- **signature** (map, REQUIRED) — Cryptographic integrity proof over the manifest payload (see Section 1.5).
- **presentationProof** (map, OPTIONAL) — Proof-of-possession binding the manifest to a specific verifier and moment (see Section 6.6.2).
- **livenessAttestation** (map, OPTIONAL) — Evidence of recent interactive human authentication (see Section 6.6.3).
- **actorState** (map, OPTIONAL, PREVIEW) — The session principal and a human/agent/hybrid executor (see Section 1.4.7).
- **postQuantumSignature** (map, OPTIONAL, PREVIEW) — A supplementary post-quantum integrity proof during the dual-signature migration period (see Section 6.7.3).

The semantics of these properties — the evaluation sequence (Section 3.1), the tiered trust model (Section 6.4.2), consent matching, signature verification, and all conformance obligations — are defined against this abstract data model. They do not depend on which production rule was used to serialize a given manifest.

Extensions and future versions that define new root-level members MUST define them as abstract properties with a representation in each production rule they are used with; unknown members encountered in a representation are preserved as unprocessed entries (Section 3.1.1).

> **Note:** The integrity proof is computed over the bytes of a specific production (see Section 1.7.5), so a signature is bound to one encoding. Converting a signed manifest from one encoding to another requires re-signing under the target production. The *meaning* of the manifest is preserved across encodings; the *signature bytes* are not.

1.7.2. Production Rules

A **production rule** (or simply *production*) is a bidirectional mapping between the abstract data model and a concrete byte representation. *Production* serializes an abstract manifest into a representation; *consumption* parses a representation back into the abstract manifest. A production rule MUST:

1. Define a media type that identifies the encoding.
2. Specify how each abstract property in Section 1.7.1 is represented, including required and optional properties.
3. Guarantee that consumption of a produced representation yields an abstract manifest equal to the original (lossless round-trip) for all properties defined by this specification.

This specification defines two production rules: JSON-LD (the reference encoding) and CBOR-LD (a compact encoding). Future versions or extension profiles **MAY** define additional production rules (for example, a plain-JSON or COSE-based encoding) without altering the abstract data model. Evaluators **MUST NOT** assume that the absence of a production rule from this list means a manifest is non-conformant; they reject representations whose media type or encoding they do not support, exactly as they reject unsupported signature profiles (Section 1.6.5).

1.7.3. JSON-LD Production Rule (Reference Encoding)

The JSON-LD production rule is the reference encoding for the Universal Manifest and the default format for interoperation. A manifest in this encoding is a JSON-LD document [JSON-LD] as described in Section 1.2 through Section 1.6. Its media type is `application/um+ld+json` (a manifest is also valid `application/ld+json`).

The mapping from the abstract data model to this encoding is direct:

- Abstract **type**, **id**, and **contextReference** are expressed as the JSON-LD keywords `@type`, `@id`, and `@context`.
- All other abstract properties are expressed as JSON members of the same name (`manifestVersion`, `subject`, `issuedAt`, `expiresAt`, `facets`, `claims`, `consents`, `pointers`, `devices`, `requiredTrustTier`, `signature`, `presentationProof`, `livenessAttestation`, `actorState`, `postQuantumSignature`), with the types defined in Sections 1.2–1.4.
- Abstract datetimes are expressed as RFC 3339 [RFC3339] `date-time` strings; integers, strings, lists, and maps map to their JSON counterparts.
- JSON-LD permits `@context` to be either a string or an array. A lone context string is consumed as a one-element `contextReference` list; producers **SHOULD** emit the array form. This normalization does not affect the signature, which covers the concrete bytes of the consumed production (Section 1.7.5).

Because the abstract model defines this as the reference encoding, every example elsewhere in this specification is a JSON-LD production. Example 1 shows a minimal manifest in this encoding.

1.7.4. CBOR-LD Production Rule (Compact Encoding)

The CBOR-LD production rule is a second, compact binary encoding defined to demonstrate that the abstract data model is genuinely format-independent. It is intended for constrained channels —

QR codes, NFC tags, and low-bandwidth or embedded transports — where the verbosity of JSON-LD is costly. Its media type is `application/um+cbor-ld`.

CBOR-LD [CBOR-LD] compresses a JSON-LD document into Concise Binary Object Representation (CBOR, [RFC8949]) by using the manifest's JSON-LD context to replace string term names with compact integer tokens. Production and consumption are defined as follows:

1. **Production (encode).** Begin with the JSON-LD production of the abstract manifest (Section 1.7.3). Using the Universal Manifest context identified by **contextReference**, map each defined term to its integer token, then emit the document as CBOR. The **contextReference** is carried as a context identifier (not as inlined term strings) so the decoder can reconstruct the term table.
2. **Consumption (decode).** Read the context identifier, load the corresponding Universal Manifest context, reverse the integer-to-term mapping, and reconstruct the JSON-LD document. Interpreting that document under Section 1.7.3 yields the abstract manifest.

This mapping is bidirectional and semantically lossless for all abstract properties defined by this specification: a manifest produced in CBOR-LD and then consumed MUST yield the same abstract manifest as the JSON-LD production from which it was derived. Conformance is therefore identical across the two encodings; only the bytes on the wire differ.

CBOR-LD decoding requires the manifest's JSON-LD context to be available to the decoder (whether fetched, cached, or pre-registered). Deployments using the CBOR-LD encoding on offline or constrained channels SHOULD pre-provision the Universal Manifest context for the relevant specification version, since the context cannot be assumed to be retrievable at decode time. Term tokenization is deterministic only when both parties use the same context version, which the versioned namespace (Section 1.2.1) makes explicit.

> **Preview:** The CBOR-LD production rule is provided to validate format independence and to anchor the abstract-data-model architecture. A complete CBOR-LD profile — including a registered context-to-token table, a fixed signature production for the binary encoding (see Section 1.7.5), and conformance fixtures — will be finalized after working-group review. Working-group input is requested on whether CBOR-LD is the right compact encoding to standardize first.

1.7.5. Signing and Production

The integrity proof defined in Section 1.5 and Section 1.6 is computed over the canonical bytes of a specific production. Signature Profile A (Section 1.6) is defined against the JSON-LD production:

the signing input is the JCS-canonicalized [RFC 8785] JSON-LD document with the signing-input exclusions of Section 1.6.3 removed (the **signature** property and, when present, **presentationProof** and **postQuantumSignature**).

Each production rule MUST either reuse a profile defined against the abstract model or specify how the integrity proof is computed over its own canonical bytes, so that a verifier processing that encoding can recompute the signing input deterministically. Because a signature is bound to the bytes of one production, a manifest re-encoded under a different production rule MUST be re-signed under that production's signing rule. Abstract semantics are preserved across encodings; signatures are not portable across encodings.

> **Note:** The working group is evaluating whether a future signature profile should be defined over the abstract data model itself rather than over production bytes, which would make a single integrity proof portable across encodings at the cost of requiring a canonicalization of the abstract model. This draft builds in per-production signing as the default. Feedback is requested.

2. Entities and Facets

Universal Manifest adopts a compositional pattern allowing nested structures (`facets` mapping to specific `entities`), drawing from semantic web standards for deeply interlinked resources.

2.1. um:Facet Module

A facet is a composable part grouped within a manifest's envelope. A facet object MUST contain the following fields:

- `@id` — A URI uniquely identifying this facet within the manifest. MUST be present. The consent mechanism (Section 1.4.4) uses `consents[].facetRef` to match against `facets[].@id`, and the receipt (Section 3.3.1) records facet status by `@id`.
- `@type` — MUST include `"um:Facet"`.

A facet object SHOULD contain the following field:

- `entity` — A `um:Entity` object (Section 2.2) holding the facet's payload parameters. A facet without an `entity` is structurally valid but carries no payload.

A facet object MAY contain the following fields:

- `name` — A human-readable display label for the facet.
- `ref` — URI routing to the facet's authoritative source.
- `requiredTrustTier` — Integer (0–3) overriding the manifest-level trust tier for this facet. Can only raise the floor, not lower it. See Section 6.4.5.

Facets are identified by `@id` for consent matching and receipt recording. The `name` field is a display label and **MUST NOT** be used as a unique identifier.

`@type` shape convention: Throughout this specification, a `@type` member **MUST** include the type value required for its object kind (for example `um:Facet`, `um:Consent`, `um:Manifest`) and **MAY** be expressed as a bare string when that is the sole type, or as an array when multiple type values apply. Where an object's definition elsewhere says its `@type` "MUST be" a single value, read it as "MUST include" under this convention.

Internationalization: Human-readable strings (such as `name`, `label`, `displayName`, `reason`, and `rotationReason`) **SHOULD** be language-tagged using JSON-LD language maps where multilingual display is expected. Evaluators **MUST NOT** use display fields for matching (the `@id`-not-`name` rule above generalizes to all display fields).

Example 8: Plaintext facet with all fields

```
{
  "@id": "urn:uuid:facet-public-profile",
  "@type": "um:Facet",
  "name": "publicProfile",
  "ref": "https://example.com/profiles/alice",
  "entity": {
    "@id": "urn:uuid:entity-alice-profile",
    "@type": ["um:Entity", "um:IdentityProfile"],
    "displayName": "Alice Example",
    "avatarUrl": "https://example.com/avatars/alice.png"
  }
}
```

2.2. um:Entity Base

The `um:Entity` acts as the base classification for all embedded configurations, representations, or localized states. An entity object **MUST** contain the following fields:

- `@id` — A URI uniquely identifying this entity. **MUST** be present.
- `@type` — An array of type strings. **MUST** be present and **MUST** include at least one type value.

> **Note:** The `um:Entity` base requires only `@id` and `@type`. Domain profiles extend the entity with additional fields specific to their use case.

All other fields on an entity are profile-extensible. Evaluators **MUST** ignore unknown entity fields for processing purposes but **MUST NOT** strip them before signature verification.

Example 9: Plaintext entity

```
{
  "@id": "urn:uuid:entity-alice-profile",
  "@type": ["um:Entity", "um:IdentityProfile"],
  "displayName": "Alice Example",
  "avatarUrl": "https://example.com/avatars/alice.png",
  "preferredLanguage": "en"
}
```

2.3. Encrypted Facets (JWE Inline Profile)

A facet **MAY** carry an encrypted entity payload using the JWE inline encryption profile. This enables the holder to include sensitive data that is readable only by designated recipients while remaining a sealed entry to all other evaluators. Encrypted facets support the sealed-entry principle: evaluators acknowledge encrypted facets as present but cannot read their contents without the appropriate decryption key.

2.3.1. Declaration

To declare an encrypted facet, the facet **MUST** include the field `encryptionProfile` with the value `"jwe-inline-v1"`. When `encryptionProfile` is present, the `entity` field **MUST** contain a JWE JSON Serialization object instead of a plain `um:Entity`.

2.3.2. JWE Structure

The `entity` value for an encrypted facet **MUST** conform to the following structure (a JWE JSON Serialization object per [RFC7516]):

- `protected` — base64url-encoded JWE Protected Header. **MUST** specify `alg` (key agreement algorithm) and `enc` (content encryption algorithm).
- `recipients` — Array of recipient objects. Each recipient object **MUST** contain a `header` object with a `kid` field (key identifier, typically a DID URL) and an `encrypted_key` field (base64url-encoded wrapped content encryption key).
- `iv` — base64url-encoded initialization vector.
- `ciphertext` — base64url-encoded encrypted payload.
- `tag` — base64url-encoded authentication tag.

Evaluators that do not possess a decryption key for any recipient entry **MUST** treat the facet as a sealed entry. Evaluators **MUST NOT** reject a manifest solely because it contains encrypted facets they cannot decrypt.

Example 10: Facet with JWE inline encryption

```
{
  "@id": "urn:uuid:facet-medical-records",
  "@type": "um:Facet",
  "name": "privateHealth",
  "encryptionProfile": "jwe-inline-v1",
  "entity": {
    "protected": "eyJhbGciOiJFQ0RILUVTK0EyNTZLVyIsImVuYyI6IkEyNTZHQ00ifQ",
    "recipients": [
      {
        "header": {
          "kid": "did:example:clinic#key-agree-1"
        },
        "encrypted_key": "base64url-encrypted-key-for-k1"
      }
    ],
    "iv": "base64url-iv-1",
    "ciphertext": "base64url-ciphertext-1",
    "tag": "base64url-tag-1"
  }
}
```

2.3.3. Key Rotation

When a recipient's key agreement key is rotated, the holder **MUST** re-encrypt the content encryption key under the new key and update the `recipients` array. The JWE entity object **MAY** include the following fields to signal key rotation:

- `previousKid` — the `kid` of the replaced key.
- `rotationReason` — human-readable description of the rotation cause.

Key rotation changes the manifest's signing input. The holder **MUST** re-sign the manifest after any key rotation operation.

2.3.4. Recipient Revocation

To revoke a recipient's access, the holder **MUST** remove the recipient from the `recipients` array, re-encrypt the payload with a new content encryption key, and re-issue the manifest. The JWE entity object **MAY** include the following fields to signal revocation:

- `revokedRecipientKid` — the `kid` of the revoked recipient.
- `revocationAction` — human-readable description of the revocation action taken.

When all recipients are revoked, the `recipients` array **MUST** be an empty array. The encrypted payload remains present but is not decryptable by any party.

2.4. JWE Algorithm Constraints

v0.3 supports sealed entries via the JWE inline encryption profile (Section 2.3) but does not constrain which algorithms are permitted. v0.4 defines a mandatory-to-implement algorithm pair that establishes baseline interoperability for encrypted facets across evaluator implementations.

2.4.1. Baseline Algorithm Pair

Evaluators claiming v0.4 conformance with encrypted facet support **MUST** implement the following algorithm pair:

- **Key agreement algorithm (`alg`):** `ECDH-ES+A256KW` -- Elliptic Curve Diffie-Hellman Ephemeral Static key agreement with AES-256 Key Wrap, per [RFC7518] Section 4.6.
- **Content encryption algorithm (`enc`):** `A256GCM` -- AES-256 in Galois/Counter Mode, per [RFC7518] Section 5.3.

This pair is drawn from the examples in v0.3 and provides 256-bit symmetric security with authenticated encryption.

2.4.2. Evaluation Contract

Holders **MUST** use the baseline algorithm pair for encrypted facets unless both parties have negotiated an alternative out-of-band. Evaluators **MUST** support the baseline algorithm pair and **MAY** support additional algorithm pairs. Additional algorithm pairs **MAY** be registered through the profile registration mechanism (Section 5.1).

When an evaluator encounters a JWE header specifying an unsupported algorithm pair, the evaluator **MUST** record the facet status as `"opaque"` in the receipt with reason `um:reason:crypto:unsupported-algorithm` (Section 3.3.2.3). The evaluator **MUST NOT** reject the manifest; the facet is treated as a sealed entry.

2.4.3. Algorithm Negotiation

For bilateral exchanges (Section 3.5), participants **MAY** negotiate alternative algorithm pairs by including a `supportedAlgorithms` array in the session object. When no negotiation occurs, the baseline pair applies. Algorithm negotiation details are defined by profile documents; this specification requires only that all conformant evaluators support the baseline pair.

> **Note:** The working group may consider adding `ECDH-ES / A128CBC-HS256` as an additional should-implement pair for environments where library support for AES-GCM is limited. Feedback is requested on whether broader library support justifies a second required pair.

> **Note:** The working group is evaluating whether v0.4 should define a negotiation mechanism for algorithm selection in bilateral exchanges or whether out-of-band negotiation is sufficient.

3. Manifest Lifecycle and Caching

Parallel to the Web Application Manifest lifecycle, the Universal Manifest must be systematically processed, applied, and occasionally evicted from client edges.

3.1. Evaluation Sequence

When a user agent, smart edge, or any evaluating platform encounters a Universal Manifest, it **MUST** process the manifest through a six-stage evaluation sequence. Each stage produces a defined output that feeds the next stage. Implementations **MAY** short-circuit the sequence at any stage by emitting a rejection receipt (see Section 3.3).

3.1.1. Stage 1: Arrive

The manifest is received and its envelope structure becomes visible to the evaluator. The evaluator **MUST** consume the representation through a production rule it supports (Section 1.7.2) to obtain the

abstract manifest, then confirm the existence of the required abstract properties (**contextReference**, **id**, **type**, `manifestVersion`, `subject`, `issuedAt`, `expiresAt`, `signature`) and verify that **type** includes `um:Manifest` (per Section 1.2.3). In the JSON-LD reference encoding these properties appear as `@context`, `@id`, and `@type` with the remaining members alongside them. The evaluator **MUST** ignore unknown properties for processing purposes but **MUST NOT** remove them from the payload prior to signature verification. The signing input is the canonical bytes of the consumed production with the signing-input exclusions removed — the `signature` property and, when present, `presentationProof` and `postQuantumSignature` (for the JSON-LD production, the JCS-canonicalized document per Section 1.6.3; in general, see Section 1.7.5). After verification succeeds, unrecognized properties have no normative semantics and **MUST NOT** affect evaluation sequence outcomes. If consumption or structural validation fails, the evaluation sequence terminates with a `rejected` receipt.

> **Note:** This ensures extension fields survive the verification boundary while having no effect on evaluation-sequence behavior. Forward compatibility is preserved because evaluators do not act on unknown fields, but signature integrity is maintained because those fields remain in the signing input.

3.1.2. Stage 2: Verify

The evaluator **MUST** verify the manifest's cryptographic integrity and freshness. This stage includes:

1. Signature verification over the signing input defined by the production rule and declared signature profile (Section 1.7.5; for the JSON-LD reference encoding under Signature Profile A, the JCS-canonicalized document per Section 1.6.3).
2. Freshness enforcement: reject if `now > expiresAt` or if `issuedAt > expiresAt`.
3. Revocation status resolution, if `signature.statusRef` is present and the evaluator implements revocation-aware verification.

If signature verification fails, the manifest **MUST** be rejected. The evaluator **MAY** retain the manifest for debugging purposes.

Credential-binding verification sub-steps. When the manifest relies on the credential-binding mechanisms of Section 6.6 for Tier 1 or higher assurance, the evaluator **MUST** additionally perform the following sub-steps as part of the Verify stage. Each sub-step records its outcome in the credential-binding receipt fields (Section 3.3.1.1). Evaluators that do not implement credential binding skip these sub-steps and record the corresponding statuses as `"absent"`.

1. **(2a) Holder-binding verification.** For each claim carrying `holderBinding` (Section 6.6.1), verify the binding according to its `mode` (`sd-jwt-kb`, `bbs-holder-commitment`, or `reciprocal-control`). Record `holderBindingStatus`. A claim relied upon for assurance whose holder binding fails or is absent **MUST NOT** be granted trust above Tier 0.
2. **(2b) Presentation-proof verification.** If the manifest carries a `presentationProof` (Section 6.6.2), verify that the `challenge` matches the verifier-issued nonce, the `audience` matches this verifier, and the `proofValue` validates over the signing-input hash, `challenge`, `audience`, and `created`. Record `presentationProofStatus`. A failed presentation proof **MUST** cause the manifest to be treated as replay-suspect and rejected for interactive verification. When the evaluator issued a challenge for this presentation (an interactive presentation) and no `presentationProof` is present, the evaluator **MUST** treat the absence identically to a failed proof, record `presentationProofStatus: "missing-required"`, and reject the manifest for interactive verification.
3. **(2c) Liveness-attestation verification.** If a `livenessAttestation` (Section 6.6.3) is present, verify its proof and compute its `freshnessClass` from `attestedAt`. Record `livenessStatus`. Evaluators **MUST NOT** treat a `stale` or `unknown` freshness class as current human presence.
4. **(2d) Cross-DID binding-proof verification.** For each `identity.crossDidBinding` claim (Section 6.4.4), verify the attester authorization (when the binding is attester-asserted) and any Tier 2 `bindingProof` or Tier 3 `ceremonyProof` per the claim-proof process (Section 6.4.9). Record each result in `crossDidBindingStatus`.

> **Note:** The Stage-2 credential-binding sub-steps (2a–2d) are the normative record of how the binding mechanics of Section 6.6 are woven into the evaluation sequence. They are new normative content in v0.4 (see Changes from v0.3) and operationalize the credential-binding requirements of Section 6.6; the binding modes and proof procedures they invoke are defined there.

Evaluators **SHOULD** allow a clock-skew tolerance of no more than 60 seconds for `issuedAt` and `expiresAt` comparisons. Manifests with `issuedAt` more than 60 seconds in the future relative to the evaluator's clock **SHOULD** be rejected with outcome `rejected` and `freshnessCheck: "stale"`. Deployments in environments without NTP (constrained devices, air-gapped venues) **MAY** configure wider tolerances but **MUST** document the tolerance in their conformance claim.

3.1.3. Stage 3: Project

The evaluator **MUST** extract only the facets, claims, pointers, and device entries relevant to its processing context. Selective disclosure is holder-controlled: the manifest holder determines which

facets are included in a given manifest instance. The evaluator **MUST NOT** assume that the manifest contains the complete set of the subject's facets. Facets not included in the manifest are not absent — they are not projected for this interaction.

3.1.4. Stage 4: Consent

The evaluator **MUST** evaluate per-facet consent records before acting on facet data. For each projected facet, the evaluator matches the facet's `@id` against `consents[].facetRef` values (see Section 1.4.4). For each facet:

- If a `consents` entry governs the facet (i.e., a consent entry exists whose `facetRef` matches the facet's `@id`), the evaluator **MUST** verify that consent scope, purpose, and expiry are satisfied. Specifically: the evaluator's intended operation **MUST** appear in the consent's `scope` array, the evaluator's intended use **MUST** fall within the consent's declared `purpose`, and the current time **MUST** be within the consent's validity window (at or after `grantedAt`, before `expiresAt`). If a `withdrawnAt` field is present, the consent is treated as absent.
- If a facet is encrypted (see Section 2.3) and the evaluator lacks a decryption key, the evaluator **MUST** acknowledge the facet as a sealed entry. Sealed entries are recorded as present but not read.
- If no consent entry governs the facet and the facet is not a sealed entry, the evaluator **MUST** record the facet status as `"consent-missing"`.

Evaluators **MUST NOT** process facet data when required consent is absent, expired, or withdrawn.

Consent-evaluation outcomes map to receipt statuses as follows. No consent entry matches the facet → facet status `"consent-missing"`. A matching entry exists but the intended operation is not in `scope`, the intended use is not within `purpose`, or a declared condition is violated or unenforceable → facet status `"consent-denied"` (with the consent status `"scope-mismatch"`, `"purpose-mismatch"`, or `"condition-violated"` as applicable). A matching entry is expired or withdrawn → facet status `"consent-denied"` with consent status `"expired"` or `"withdrawn"`.

3.1.5. Stage 5: Compose

The evaluator composes the processing result into one of four outcome categories:

- `accepted` — all projected facets processed, all consent requirements satisfied, signature valid.
- `accepted-with-warnings` — manifest accepted but one or more non-critical conditions were noted (e.g., revocation status could not be checked).

- `accepted-partial` — some facets were processed; others were rejected, sealed (encrypted and undecryptable), or lacked consent.
- `rejected` — the manifest failed a mandatory check (signature, freshness, structural validity, or required consent).

The composed result **MUST** be machine-readable and **MUST** include per-facet status.

3.1.6. Stage 6: Receipt

The evaluator **MUST** produce a structured receipt (see Section 3.3) that honestly records what the evaluator actually did. The receipt captures the outcome of each preceding stage. Evaluators **MUST NOT** omit failed checks or suppress negative outcomes from the receipt.

3.2. Caching Formulation

For constrained devices and public displays:

1. **TTL Ejection**: Caches **MUST** immediately evict or reject payloads where the system clock surpasses `expiresAt`.
2. **Telemetry Minimization**: Centralized logging platforms **SHOULD** stream only the `@id` string (and potentially a content hash), bypassing the full manifest payload to conserve bandwidth.
3. **Identifier Rotation**: Identifiers (`@id`) **SHOULD** be rotated on each issuance (a fresh random `@id` per manifest instance) to avert heuristic tracking.

3.3. Structured Receipts

A structured receipt is the terminal output of the evaluation sequence (Section 3.1). It provides an honest, machine-readable record of what the evaluator did with the manifest. Evaluators **MUST** produce a receipt for every manifest processed through the evaluation sequence.

3.3.1. Receipt Fields

A receipt **MUST** include the following fields:

- `@type` — **MUST** include `"um:Receipt"`.
- `manifestId` — the `@id` of the processed manifest.

- `outcome` — one of `"accepted"`, `"accepted-with-warnings"`, `"accepted-partial"`, or `"rejected"`.
- `signatureCheck` — result of signature verification: `"valid"`, `"invalid"`, `"unsupported-profile"`, or `"not-evaluated"`.
- `freshnessCheck` — result of TTL enforcement: `"fresh"`, `"expired"`, `"stale"`, or `"not-evaluated"`. The `"stale"` value indicates that the manifest's `issuedAt` is in the future relative to the evaluator's clock (beyond the permitted clock-skew tolerance).

A manifest rejected during the Arrive stage (structural failure) terminates before signature or freshness checks run; stages not reached before termination record `"not-evaluated"` for the corresponding field.

A receipt **MUST** include a `facetStatuses` array; when the manifest contains zero facets, it **MUST** be an empty array. Each entry in the array is a per-facet status object containing a `facetId` (matching the facet's `@id`), `status` (`"processed"`, `"opaque"`, `"consent-denied"`, `"consent-missing"`, `"trustTierUnsupported"`, or `"not-projected"`), an **OPTIONAL** `name` (the facet's display label, if present), and an **OPTIONAL** `reason` string. The `"trustTierUnsupported"` status records a facet withheld because its `requiredTrustTier` cannot be verified (Section 6.4.5).

The `"not-projected"` status indicates that the evaluator's local policy expected a specific facet (identified by type or `@id`) that is absent from the presented manifest. This status is **OPTIONAL** and applies only when the evaluator has prior knowledge of the subject's manifest schema. Evaluators without such knowledge **MUST NOT** produce `"not-projected"` entries.

A receipt **SHOULD** include the following fields when applicable:

- `revocationStatus` — `"active"`, `"revoked"`, `"suspended"`, or `"unchecked"`.
- `revocationReason` — **OPTIONAL** registry-coded string (Section 3.3.2.3) recording why `revocationStatus` has its value (for example `um:reason:status:endpoint-unavailable` when status could not be resolved).
- `keyRefResolution` — `"resolved"` or `"unresolved"`, recording whether the verification `keyRef` was resolved during the Verify stage (Section 1.6.4 step 4). `"unresolved"` caps `keyRef`-derived identity assurance at Tier 0.
- `consentStatuses` — array of per-consent status objects. Each entry **MUST** contain: `facetId` (the `@id` of the governed facet), `consentRef` (reference to the consent entry; omitted when `status` is `"missing"`, since no entry exists to reference), `status` (one of `"valid"`, `"expired"`, `"withdrawn"`, `"missing"`, `"scope-mismatch"`, `"purpose-mismatch"`, or `"condition-violated"`), and `checkedAt` (RFC 3339 date-time when the consent status was evaluated).

- `claimStatuses` — array of per-claim status objects recording the outcome of claim evaluation. Each entry **MUST** contain a `claimRef` (the claim's `@id`, or its zero-based index in the `claims` array when no `@id` is present) and a `status` (one of `"verified"`, `"unverified"`, `"failed"`, `"unprocessable"`, or `"trustTierUnsupported"`), and **MAY** contain `tier` (the integer trust tier at which the claim was verified) and a `reason` string. This field is where the per-claim outcomes required by Section 1.4.3 (unrecognized claim types), Section 6.4.5 (`"trustTierUnsupported"`), and Section 6.4.9 step 7 are recorded; `identity.crossDidBinding` outcomes additionally use `crossDidBindingStatus` (Section 3.3.1.1).
- `unprocessedEntries` — array recording structural entries the evaluator preserved but did not act on, as required by Section 1.4.5 (unrecognized pointer types), Section 1.4.6 (unimplemented device entries), and Section 3.1.1. Each entry **MUST** contain `member` (the structural member name, e.g., `"pointers"` or `"devices"`) and `entryRef` (the entry's `@id`, or its zero-based index when no `@id` is present), and **MAY** contain a `reason` string.
- `processedAt` — RFC 3339 date-time when the receipt was produced.
- `warnings` — array of warning objects for the `accepted-with-warnings` outcome. Each entry **MUST** contain a `code` drawn from the structured reason registry (Section 3.3.2.3) and a human-readable `message` string, and **MAY** contain reference members (e.g., `affectedClaims`, `affectedFacets`). A bare string **MAY** be used only for an unregistered condition that has no applicable `code`.
- `exchangeId` — **OPTIONAL** correlation identifier; **REQUIRED** on receipts produced in a bilateral session (Section 3.5.3).
- `receiptId` — unique identifier for the receipt (**SHOULD** be an opaque URI, e.g., `urn:uuid:...`).
- `evaluatorId` — DID or identifier of the evaluator that produced the receipt. **REQUIRED** on receipts exchanged in a bilateral session (Section 3.5.3).
- `receiptSignature` — optional signature over the receipt using the evaluator's key, following the same profile as manifest signatures (Signature Profile A).

Receipt signing is **RECOMMENDED** for accountability use cases but is not required for v0.4 conformance. Unsigned receipts are valid evaluation sequence outputs but provide weaker non-repudiation guarantees.

3.3.1.1. Credential Binding Status Fields

When the evaluator processes the credential-binding mechanisms defined in Section 6.6, the receipt **SHOULD** record their outcomes using the following fields. Evaluators that do not implement

credential binding omit these fields; their absence is equivalent to `"absent"`.

- `holderBindingStatus` — One of `"verified"`, `"failed"`, `"unsupported-mode"`, or `"absent"`. Records the result of holder-binding verification (Section 6.6.1). `"absent"` indicates the claim carried no `holderBinding`.
- `presentationProofStatus` — One of `"verified"`, `"failed"`, `"missing-required"`, or `"absent"`. Records the result of presentation-proof verification (Section 6.6.2). `"missing-required"` indicates an interactive presentation (verifier-issued challenge) in which the required `presentationProof` was absent; `"absent"` indicates a non-interactive presentation that legitimately carried none.
- `livenessStatus` — An object recording liveness-attestation evaluation (Section 6.6.3), containing `freshnessClass` (one of `"live"`, `"recent"`, `"stale"`, or `"unknown"`) and, when present, the `method` and `userVerified` values copied from the attestation.
- `crossDidBindingStatus` — An array of per-binding status objects, each containing the bound DID set and a `status` (one of `"verified"`, `"attester-asserted"`, `"failed"`, or `"trustTierUnsupported"`), summarizing the verification of each `identity.crossDidBinding` claim (Section 6.4.4, Section 6.4.9).
- `effectiveTrustTier` — The highest trust tier the evaluator actually verified for the claims it relied on, per Section 6.4.2. A claim without a verified `holderBinding` contributes at most Tier 0 (Section 6.6.1). `effectiveTrustTier` is independent of the declared `requiredTrustTier`; the evaluator **MUST NOT** act on any item whose `requiredTrustTier` exceeds the `effectiveTrustTier` it achieved (Section 6.4.5). Recording both makes the gap between declared and verified trust explicit.

> **Preview decision:** These receipt fields are the normative record of credential-binding outcomes (Section 6.6). One open working-group decision remains on their status: whether `effectiveTrustTier` should be a **REQUIRED** field of every v0.4 receipt (recommended direction: required, with non-binding evaluators recording Tier 0 and the binding statuses absent) or remain a **SHOULD** field conditional on credential-binding support, as built here. Flag to revise.

Example 11: Structured receipt

```
{
  "@type": "um:Receipt",
  "manifestId": "urn:uuid:123e4567-e89b-12d3-a456-426614174000",
  "outcome": "accepted-partial",
  "signatureCheck": "valid",
  "freshnessCheck": "fresh",
  "revocationStatus": "unchecked",
  "facetStatuses": [
    { "facetId": "urn:uuid:facet-public-profile", "name": "publicProfile", "
    { "facetId": "urn:uuid:facet-private-health", "name": "privateHealth", "
  ],
  "processedAt": "2026-05-19T12:00:00Z"
}
```

3.3.2. Receipt as a First-Class Manifest Class PREVIEW

Beyond its role as the terminal output of a single evaluation (Section 3.3.1), a receipt is itself a **first-class manifest class** (Section 1.0.1): a signed, portable record that can be chained, retained, and independently verified. A receipt manifest carries the common envelope members (Section 1.2) with `@type` including both `um:Manifest` and `um:Receipt`, plus the receipt fields below. This promotion is additive; an evaluator that only emits inline receipts remains conformant, and the chain-integrity members are OPTIONAL unless a receipt is part of a sequence.

3.3.2.1. Chain Integrity

Receipts in a session or audit sequence form a hash-linked chain. A chained receipt MUST include:

- `chainId` — A URI identifying the chain this receipt belongs to. REQUIRED on every chained receipt; for a bilateral session it MUST equal the `sessionId` (Section 3.5). It is the stable identifier that ties a sequence of receipts together (`exchangeId` is per exchange round and does not serve this purpose).
- `seq` — A monotonically increasing non-negative integer giving the receipt's position in the chain. The first receipt in a chain has `seq` 0.

- `prevHash` — The hash of the immediately preceding receipt's canonical signing bytes (Section 1.6.3), encoded as a multibase/multihash string [MULTIFORMATS]. The multihash function **MUST** be SHA-256 for v0.4; other functions **MAY** be used additionally. For the first receipt (`seq` 0), `prevHash` **MUST** be omitted.

A chained receipt **SHOULD** be signed with a **session-scoped signing key** (for example, the `deviceCapability.sessionSigningKey` of Section 1.4.6.2, or another ephemeral key bound to the session) rather than a long-lived identity key, so that a receipt chain does not become a long-lived correlator. Because a session-scoped key cannot be authorized through the subject's DID document the way Section 6.4.9 authorizes manifest-signing keys, a session-scoped receipt-signing key **MUST** be introduced by a `sessionKeyAuthorization` — a statement naming the session key, the `chainId`, and a validity window, signed by a key authorized for the evaluator's DID — carried in or referenced from the first receipt of the chain, so that the chain is attributable to a known evaluator. Evaluators verifying a receipt chain **MUST** confirm that each `seq` increments by one without gaps and that each `prevHash` matches the prior receipt; a broken link **MUST** be reported and the chain after the break treated as unverified.

3.3.2.2. Typed Event Vocabulary

A receipt manifest **MAY** carry an `events` array recording typed lifecycle events. Each event object **MUST** carry an `eventType` from the following registry and an `at` (RFC 3339) timestamp; it **MAY** carry an event-specific `reason` drawn from the structured reason registry (Section 3.3.2.3) and a `subjectRef` identifying the affected facet, claim, consent, or device. The recognized event classes are:

- `manifest-arrived`, `manifest-verified`, `manifest-rejected` — envelope lifecycle.
- `facet-processed`, `facet-sealed`, `facet-consent-denied` — facet outcomes.
- `consent-granted`, `consent-withdrawn`, `consent-expired` — consent transitions.
- `claim-verified`, `claim-failed`, `binding-verified`, `binding-failed` — trust outcomes.
- `avatar-retrieved`, `avatar-substituted` — presence/avatar disclosure outcomes (presence is a locator, not authorization).
- `session-completed` — session terminal event.

This is a registry, not a closed enumeration: new event classes are added through the profile registration mechanism (Section 5.1). Evaluators encountering an unrecognized `eventType` **MUST** preserve the event but **MUST NOT** act on it.

3.3.2.3. Structured Reason Registry

Receipt `reason` values (on facet statuses, consent statuses, and events) **SHOULD** be drawn from a structured reason registry using `um:reason:<category>:<code>` naming (for example `um:reason:consent:withdrawn`, `um:reason:crypto:no-decryption-key`, `um:reason:trust:tier-unsupported`). Structured reasons make receipts machine-comparable across implementations. Free-text reasons remain valid where no registry code applies. New reason codes are registered through the profile registration mechanism (Section 5.1).

3.3.2.4. Transparency-Log Anchoring (optional)

A receipt manifest **MAY** be anchored to an append-only transparency log following the Certificate Transparency 2.0 model [RFC9162]. When anchored, the receipt **MAY** carry a `transparencyAnchor` object with `logId` (the log's identifier), `inclusionProof` (a Merkle inclusion proof), and `anchoredAt` (RFC 3339). Anchoring is **OPTIONAL** and does not increase per-event cost for deployments that do not use it; it provides tamper-evident, independently auditable history for accountability use cases. Evaluators that do not implement transparency anchoring **MUST** preserve the `transparencyAnchor` field without acting on it.

Example 12: Chained receipt manifest with typed events


```
{
  "@context": ["https://universalmanifest.net/ns/v0.4"],
  "@id": "urn:uuid:receipt-0002",
  "@type": ["um:Manifest", "um:Receipt"],
  "manifestVersion": "0.4",
  "subject": "did:key:z6MkVenueEdge",
  "issuedAt": "2026-06-09T12:00:05Z",
  "expiresAt": "2026-06-10T12:00:05Z",
  "manifestId": "urn:uuid:123e4567-e89b-12d3-a456-426614174000",
  "outcome": "accepted",
  "signatureCheck": "valid",
  "freshnessCheck": "fresh",
  "facetStatuses": [
    { "facetId": "urn:uuid:facet-public-profile", "status": "processed" }
  ],
  "chainId": "urn:uuid:chain-7c1f",
  "seq": 2,
  "prevHash": "uEiD9f2c...",
  "events": [
    { "eventType": "manifest-verified", "at": "2026-06-09T12:00:05Z" },
    { "eventType": "facet-processed", "at": "2026-06-09T12:00:05Z", "subject
  ],
  "signature": {
    "algorithm": "Ed25519",
    "canonicalization": "JCS-RFC8785",
    "keyRef": "did:key:z6MkSessionEphemeral#keys-1",
    "value": "base64url-encoded-signature-bytes"
  }
}
```


Preview: Promoting the Receipt to a first-class manifest class — with the `seq`/`prevHash` chain, the typed event vocabulary, the structured reason registry, and optional CT-style transparency anchoring — is built into this preview on the editors' recommended default (promote in v0.4; additive). Working-group input is requested on the canonical event-class and reason-code registries and on the hash/multibase encoding for `prevHash`.

3.4. statusRef Resolution Schema

This section defines the resolution protocol for the `signature.statusRef` field (Section 1.6.6), specifying how evaluators resolve revocation and status information from the URI provided in a manifest's signature.

`signature.statusRef` resolves the status of the **manifest instance** identified by `manifestId`. It does **not** convey the revocation status of any key. Key revocation is determined separately, through the key's DID document verification-method state (for DID-based keys) or through the status mechanism of the credential carrying it (a `claimProof` entry's own `statusRef`, Section 6.4.3). These are distinct objects of revocation and **MUST NOT** be conflated.

v0.3 provides the `statusRef` extension point but does not define the response format. v0.4 closes this gap with a normative response schema and evaluator procedure.

3.4.1. Resolution Procedure

When `signature.statusRef` is present and the evaluator implements revocation-aware verification (Section 1.6.6, Section 4.7), the evaluator **MUST** resolve the status using the following procedure:

1. Issue an HTTP `GET` request to the `statusRef` URI.
2. If the evaluator has a cached response, the evaluator **SHOULD** include its most recently stored response `cursor` for this manifest in an `If-None-Match` header to enable conditional requests (subject to the `revocationCursor` floor below).
3. The status endpoint **MUST** respond with content type `application/json`.
4. The evaluator **MUST** parse the response and evaluate the `status` field.

`statusRef` URIs **MUST** use the `https` scheme. Status responses carry no independent signature, so authenticated transport is the only integrity protection for status information; evaluators **MUST NOT** resolve a `statusRef` over plaintext transport and **MUST** treat a non-HTTPS `statusRef` as an unreachable endpoint (Section 3.4.4).

Status endpoints **SHOULD** return an `ETag` equal to the current `cursor`; a `304 Not Modified` response means the status is unchanged since the cursor the evaluator supplied. The holder-embedded `signature.revocationCursor` is the issuance-time floor: an evaluator whose cached response is older than `revocationCursor` **MUST** revalidate, and the response `cursor` supersedes `revocationCursor` thereafter.

3.4.2. Response Schema

A status endpoint MUST return a JSON object containing the following fields:

- `manifestId` -- The `@id` of the manifest whose status is being queried. MUST match the manifest's `@id`.
- `status` -- One of `"active"`, `"revoked"`, or `"suspended"`. MUST be present.
- `updatedAt` -- RFC 3339 timestamp of the last status change. MUST be present.

A status endpoint MAY include:

- `reason` -- A human-readable string describing the reason for the current status.
- `cursor` -- An opaque string that evaluators SHOULD store and use in subsequent conditional requests.
- `nextCheck` -- An ISO 8601 duration (e.g., `"PT1H"`) recommending when the evaluator should next poll. Evaluators SHOULD respect this value.

3.4.3. Status Semantics

- `"active"` -- The manifest's signature and credentials remain valid. The evaluator MUST continue evaluation.
- `"revoked"` -- The manifest has been permanently invalidated by the holder. The evaluator MUST reject the manifest with receipt outcome `"rejected"` and `revocationStatus: "revoked"`.
- `"suspended"` -- The manifest has been temporarily suspended. The evaluator SHOULD treat the manifest as `"accepted-with-warnings"` and record `revocationStatus: "suspended"` in the receipt. Evaluators MAY apply local policy to decide whether to process suspended manifests.

3.4.4. Error Handling

When the status endpoint is unreachable or returns an error, evaluators MUST apply the following behavior:

- **HTTP 404 (Not Found):** The manifest is unknown to the status provider. Record `revocationStatus: "unchecked"` with reason `um:reason:status:endpoint-unknown-manifest`.

- **HTTP 503 (Service Unavailable) or network failure:** The status endpoint is temporarily unavailable. Record `revocationStatus: "unchecked"` with reason `um:reason:status:endpoint-unavailable`. The evaluator **MUST NOT** reject the manifest solely because the status endpoint is unreachable.
- **HTTP 4xx/5xx (other):** Record `revocationStatus: "unchecked"` with the HTTP status code in the reason field.

Evaluators operating in offline mode **MUST** record `revocationStatus: "unchecked"` with reason `um:reason:status:offline`.

Example 13: statusRef response

```
{
  "manifestId": "urn:uuid:123e4567-e89b-12d3-a456-426614174000",
  "status": "active",
  "updatedAt": "2026-05-22T10:00:00Z",
  "cursor": "v3",
  "nextCheck": "PT1H"
}
```

Deployments **MAY** use a W3C Bitstring Status List endpoint [VC-STATUS] as the `statusRef` target, provided the evaluator can parse the response into the status semantics defined above; this is a **SHOULD**-level alternative status mechanism that mitigates the status-resolution correlation risk (Section 7).

> **Note:** The working group is evaluating whether temporarily unreachable status endpoints should result in manifest rejection for high-stakes deployments. Feedback is requested.

3.5. Bilateral Session Model PREVIEW

This section defines the protocol-layer session model for bilateral exchanges, extending the manifest-level bilateral exchange defined in Section 6.4.6 with session objects, paired receipt correlation, and exchange identifiers.

3.5.1. Session Object

A bilateral session is a time-bounded interaction between two participants who each present and evaluate the other's Universal Manifest. Implementations **MAY** use the session object defined here to manage bilateral exchanges at the protocol layer.

A session object **MUST** contain the following fields:

- `@type` -- **MUST** be `"um:BilateralSession"`.
- `sessionId` -- A globally unique URI identifying this session (e.g., `urn:uuid:...`). **MUST** be present.
- `exchangeId` -- A correlation identifier shared by both parties in a single exchange round. Both parties include the same `exchangeId` in their receipts. **MUST** be present.
- `participants` -- An array of participant objects, each containing a `did` (participant's DID) and `role` (one of `"initiator"` or `"responder"`). **MUST** contain exactly two entries.
- `initiatedAt` -- RFC 3339 timestamp when the session was created.
- `expiresAt` -- RFC 3339 timestamp when the session expires. Evaluators **MUST** reject session operations after this time.
- `state` -- The current session lifecycle state.

A session comprises exactly one exchange round: the session object carries a single `exchangeId`, and a new exchange between the same parties is a new session with a new `sessionId` and `exchangeId`. `sessionId` and `exchangeId` **MUST** each be generated with at least 128 bits of entropy (for example, a UUIDv4) and **MUST NOT** encode party identifiers, since both values are correlation tokens the two parties rely on.

3.5.2. Session Lifecycle

A bilateral session transitions through the following states:

1. `"initiated"` -- The initiator has created the session and sent the `sessionId` and `exchangeId` to the responder.
2. `"manifests-exchanged"` -- Both parties have presented their manifests. Each manifest **SHOULD** reference the `exchangeId` in its presentation context.
3. `"receipts-exchanged"` -- Both parties have produced receipts and shared them. Each receipt **MUST** include the `exchangeId`.

4. `"completed"` -- Both parties have acknowledged receipt of the other's receipt. The session is complete.
5. `"expired"` -- The session's `expiresAt` time has passed without reaching `"completed"`. Evaluators **MUST** treat expired sessions as terminated.

Transitions are strictly forward. A session **MUST NOT** return to a previous state. The `"expired"` state **MAY** be entered from any active state when the session TTL elapses.

3.5.3. Paired Receipt Correlation

When a receipt is produced as part of a bilateral session, the receipt **MUST** include an `exchangeId` field matching the session's `exchangeId`. This enables pairing: the receipt produced by Party A (evaluating Party B's manifest) can be correlated with the receipt produced by Party B (evaluating Party A's manifest) via the shared identifier. A receipt exchanged in a bilateral session **MUST** include `evaluatorId` and **SHOULD** carry `receiptSignature`, so that the exchanged receipts are attributable and tamper-evident.

3.5.4. Transport Independence

The bilateral session model is transport-agnostic. Sessions **MAY** be conducted over any transport that can carry manifest payloads in a supported production rule (Section 1.7.2) — the JSON-LD reference encoding or, once its profile is finalized (Section 1.7.4), a compact encoding such as CBOR-LD on constrained channels: local transports (NFC, BLE, QR scan), network transports (HTTPS, WebSocket), or hybrid combinations. Transport-specific bindings **MAY** be defined in profile documents. The session object defined in this section **MUST NOT** assume any specific transport capability.

Example 14: Bilateral session object

```
{
  "@type": "um:BilateralSession",
  "sessionId": "urn:uuid:session-abc-123",
  "exchangeId": "urn:uuid:exchange-def-456",
  "participants": [
    { "did": "did:key:z6MkAlice", "role": "initiator" },
    { "did": "did:key:z6MkVenue", "role": "responder" }
  ],
  "initiatedAt": "2026-05-22T10:00:00Z",
  "expiresAt": "2026-05-22T10:05:00Z",
  "state": "manifests-exchanged"
}
```

Note: The working group is evaluating whether the session model should remain part of the core specification or be published as a separate companion specification. Bilateral sessions are protocol-layer concerns that may evolve independently of the manifest format. Feedback is requested.

4. Conformance

Conformance in this specification is defined against the abstract data model (Section 1.7.1), not against any single serialization. An implementation conforms by correctly handling the abstract properties and semantics of a manifest after consuming it through at least one production rule (Section 1.7.2); the JSON-LD reference encoding (Section 1.7.3) is the encoding implementations are expected to support for interoperation. The behavioral requirements below apply to the abstract manifest regardless of the encoding it was produced in. See Section 4.5 for encoding-specific conformance obligations.

4.1. Evaluator Behavior

An evaluator **MUST** consume a manifest through a production rule it supports (Section 1.7.2) to obtain the abstract manifest, then validate its abstract properties and securely ignore unknown elements without raising fatal invocation errors. Freshness (via `expiresAt` TTL constraints) is an absolute rejection gateway. Implementers **MUST** verify `issuedAt <= expiresAt`.

Evaluators claiming v0.4 conformance **MUST** implement the six-stage evaluation sequence defined in Section 3.1. Specifically, a conformant evaluator **MUST**:

1. Execute all six stages in order (Arrive, Verify, Project, Consent, Compose, Receipt).
2. Produce a structured receipt (Section 3.3) for every processed manifest.
3. Treat encrypted facets as sealed entries when the evaluator lacks a decryption key, recording them as present in the receipt.
4. Respect `requiredTrustTier` declarations at the manifest, claim, and facet levels.

4.2. Holder Behavior

Holders generating the manifest **MUST** assign a globally stable identifier URI for the `subject` (preferably an established DID) and a random URI for the manifest root (`@id`). To shield clients from unbounded trust windows, holders **MUST** strictly bound `expiresAt` to a sensible interaction lifetime (e.g., hours or days). Holders **MUST** sign every manifest prior to distribution using Signature Profile A (Section 1.6) or a subsequent normative profile.

For local-first deployments where evaluators may verify without connectivity, holders **SHOULD** use an offline-resolvable `keyRef` method (for example `did:key`) so that key resolution (Section 1.6.4 step 4) succeeds at the edge and identity assurance is not capped at Tier 0.

v0.4 adds the following holder obligations:

- Holders encrypting facets **MUST** use the baseline JWE algorithm pair (`ECDH-ES+A256KW` / `A256GCM`, Section 2.4.1) unless an alternative has been negotiated out-of-band.
- Holders presenting any claim intended to be relied upon at Tier 1 or above **MUST** carry a `holderBinding` on that claim (Section 6.6.1).
- When presenting a manifest in response to a verifier-issued challenge (an interactive presentation), the presenter **MUST** include a `presentationProof` (Section 6.6.2).

4.3. Bilateral Participant Behavior

A Conformant Bilateral Participant **MUST** implement both Evaluator Behavior (Section 4.1) and Holder Behavior (Section 4.2). In a bilateral exchange (see Section 6.4.6), both parties independently evaluate the other's manifest. The *interaction tier floor* for the exchange is the maximum of either party's `requiredTrustTier`. This negotiated floor is distinct from the

`effectiveTrustTier` each party records (Section 3.3.1.1), which is the tier that party actually verified.

4.4. Standalone Conformance Suite

Implementations validate conformance claims natively by testing against the official `conformance/` suite—which includes fixture validation (accepting valid stubs and correctly isolating/flagging malformed artifacts like missing contexts or expired manifests).

Implementations claiming v0.4 conformance **MUST** satisfy the normative requirements of this specification. The standalone conformance suite is the canonical evidence mechanism, and implementations **SHOULD** publish their suite results and claimed level using the guidance at https://universalmanifest.net/conformance/v0.4/.

> **Note:** The v0.4 conformance suite extends the v0.3 suite with tests for newly normative features introduced in this version. Preview sections do not have conformance tests until their designs are finalized.

4.5. Conformance to the Abstract Data Model

The requirements in Sections 4.1–4.3 are stated in terms of the abstract data model and apply identically no matter which production rule (Section 1.7.2) was used to serialize a manifest. An implementation does not become more or less conformant by choosing a different encoding.

A conformant implementation:

1. **MUST** support at least one production rule defined in Section 1.7.2 and, for interoperation, **SHOULD** support the JSON-LD reference encoding (Section 1.7.3).
2. **MUST** consume supported representations into the abstract data model before applying the behavioral requirements of Section 4.1, and **MUST** reject a representation whose encoding or media type it does not support, rather than misinterpreting it.
3. **MUST**, when it both produces and consumes a given encoding, preserve every abstract property defined by this specification across a production/consumption round-trip (Section 1.7.2).
4. **MUST** verify the integrity proof against the bytes of the production it consumed (Section 1.7.5); a signature is bound to one encoding and is not assumed valid across encodings.

An implementation declaring conformance SHOULD state which production rules it supports. Supporting only the JSON-LD reference encoding is sufficient for full conformance; supporting additional encodings such as CBOR-LD (Section 1.7.4) is OPTIONAL.

4.6. Status Endpoint Conformance

A **conformant status endpoint** is the conformance class that responds to `signature.statusRef` resolution (Section 3.4). A status endpoint MUST respond to a resolution request with the response schema and status semantics defined in Section 3.4 (including the `status` values `"active"`, `"revoked"`, and `"suspended"`, and the `cursor`/`nextCheck` fields), and MUST key its response to the `manifestId` being queried. It SHOULD support conditional requests per Section 3.4.1. Resolver conformance (the federated-status conformance class of Section 8) is deferred pending the working-group decision on whether federation moves to a companion specification.

4.7. Optional-Feature Matrix

Several capabilities defined in this specification are optional modules: an implementation that does not implement one still conforms, provided it observes the mandatory baseline behavior for that capability. The keyword in the "Conformance" column is the implementation obligation for the module itself.

Feature	Conformance	Mandatory baseline when not implemented
Encrypted-facet decryption (Section 2.3)	OPTIONAL	Sealed-entry handling is REQUIRED: record the facet as present-but-sealed; never infer its content.
Revocation-aware verification (Section 3.4)	RECOMMENDED	Record `revocationStatus: "unchecked"` and do not grant revocation-gated trust.
Credential binding (Section 6.6)	OPTIONAL (REQUIRED for Tier 1+)	Record `holderBindingStatus: "absent"`; cap relied-upon claims at Tier 0.
CBOR-LD encoding (Section 1.7.4)	OPTIONAL	Support the JSON-LD reference encoding; reject unsupported encodings rather than misinterpreting them.
Transparency anchoring	OPTIONAL	Verify the manifest signature and proofs directly; do not require a transparency log.

5. Extensibility & Profiles

Echoing the extensibility models of generic W3C recommendations, proprietary manifest members can be injected via fully qualified URIs inside the linked `@context`. Because evaluators ignore unrecognized properties while preserving them (Section 3.1.1), domain-specific profiles do not compromise cross-system interoperability.

5.1. Profile Registration Mechanism PREVIEW

This section defines the formal registration process for new cryptographic, trust, and domain profiles that extend the Universal Manifest specification. The mechanism follows IANA-style registry conventions.

5.1.1. Profile Identification

Each registered profile MUST have a canonical identifier following the naming scheme: `um:profile:<category>:<name>`. Categories include:

- `signature` -- Cryptographic signature profiles (e.g., `um:profile:signature:jcs-ed25519`).
- `trust` -- Trust verification profiles (e.g., `um:profile:trust:zkp-bbs-linked-secret`).
- `domain` -- Domain-specific manifest class profiles (e.g., `um:profile:domain:receipt`).
- `binding` -- Credential binding profiles (e.g., `um:profile:binding:sd-jwt-kb`).

Each profile identifier MUST be paired with a human-readable name and a version string. Profile identifiers are globally unique and MUST NOT be reused after deprecation.

The registration process of this section also governs the subsidiary value registries referenced elsewhere in this specification: receipt event classes (Section 3.3.2.2), receipt reason codes (Section 3.3.2.3), agent-delegation scope values (Section 6.5.3.4), liveness `proofType` values (Section 6.6.3), and additional JWE algorithm pairs (Section 2.4.2).

5.1.2. Registration Process

Profile registration follows a five-step process:

1. **Proposal.** A profile author submits a profile specification to the working group via the [UM-RFC] mechanism. The specification MUST include: a scope statement, normative requirements

using RFC 2119 keywords, security considerations, an interoperability assessment, and at least one conformance test fixture.

2. **Review.** The working group reviews the specification for completeness, security, and interoperability with existing profiles. The review MUST verify that the profile does not conflict with existing registered profiles at the same extension point.
3. **Conformance integration.** The profile author provides conformance test fixtures. The working group integrates the fixtures into the conformance suite (Section 4.4).
4. **Registration.** Upon working-group consensus, the profile receives its canonical identifier and is added to the profile registry.
5. **Maintenance.** The profile author maintains the specification and conformance tests. The working group MAY deprecate the profile if maintenance lapses, security issues arise, or a superseding profile is registered.

5.1.3. Conflict Resolution

Two profiles that define incompatible semantics for the same extension point MUST NOT both be registered without resolution. The working group MUST resolve conflicts by: merging the profiles, selecting one, or defining a compatibility boundary that prevents simultaneous use.

5.1.4. Deprecation

The working group MAY deprecate a registered profile by marking it as `"deprecated"` in the registry with a deprecation date and a reference to the superseding profile, if any. Evaluators SHOULD emit a warning when encountering manifests that reference deprecated profiles. Evaluators MUST NOT reject manifests solely because they reference a deprecated profile until the working group declares the profile `"sunset"` at a major-version boundary.

5.1.5. Registry Hosting

The profile registry MUST be published as a standalone document at `https://universalmanifest.net/registry/profiles/`. The registry is updated independently of the specification version. Each registry entry contains the profile identifier, name, version, status (`"active"`, `"deprecated"`, `"sunset"`), specification URI, and registration date.

> **Note:** The working group is evaluating whether low-risk profiles (e.g., new domain entity types) should follow a lighter-weight registration process with designated expert review instead of full working-group consensus. Feedback is requested.

6. Security Considerations

Universal Manifest v0.4 defines a mandatory signature profile, an additive tiered trust model, and resource-limit guidance. This section specifies the security properties these mechanisms provide and the threats they mitigate.

6.1. Signature Limitations

Implementers **MUST** use Signature Profile A (Section 1.6) or a subsequent normative profile for production deployments. The v0.1 permissive signature format **MUST NOT** be relied upon for tamper protection.

6.2. TTL Enforcement

Bounding the `expiresAt` timeline dictates the primary line of defense against presentation replay spoofing. For interactive presentations, the `presentationProof` mechanism (Section 6.6.2) additionally binds a presentation to a specific verifier and challenge, preventing replay within the TTL window.

6.3. Resource Limits

Denial-of-Service vectors originating from disproportionately inflated arrays or recursion logic **SHOULD** be countered with hard limits on payload ingestion. Evaluators **SHOULD** enforce at least the following default limits and **MUST** document any deviation in their conformance claim:

- Maximum manifest size: 1 MB
- Maximum nesting depth: 10 levels
- Maximum array length: 1,000 entries

6.4. Identity Binding and Claim Authenticity

6.4.1. Bag of Claims Limitation

A Universal Manifest may contain claims from multiple issuers and references to multiple DIDs under a single `subject`. The manifest signature proves that the signer produced the manifest. It does **not** prove:

- That the signer controls the `subject` DID (subject-signer binding).
- That the `subject` controls all DIDs mentioned in claims or facets (cross-DID control).
- That an issuer actually issued a claim listed in the manifest (claim authenticity). The `claims[].issuer` field is a string assertion, not a verified provenance chain.

Evaluators **MUST NOT** treat the presence of claims in a signed manifest as proof that those claims are authentic or that multiple DIDs are controlled by the same entity.

6.4.2. Tiered Trust Model

The specification defines four trust tiers for claim verification. Each tier is strictly additive — a claim verified at a higher tier also satisfies all lower-tier requirements. Higher tiers provide stronger guarantees but impose more user ceremony. The specification does not mandate a minimum tier; evaluators choose based on their threat model and acceptable user friction.

Tier 0 — Signature-only. Zero friction. Claims are self-asserted by the manifest signer. No external `claimProof` material is present. Suitable for low-stakes use cases where the evaluator has an out-of-band trust relationship with the signer. Evaluators claiming Tier 0 acceptance **MUST** verify the manifest signature per the declared profile. Tier 0 **MUST NOT** be used as sufficient assurance for Sybil-critical decisions.

Tier 1 — Attested or claimProof-backed. Low friction. Some or all claims carry external `claimProof` material (Verifiable Presentations) or an attested cross-DID binding claim (`identity.crossDidBinding`). Evaluators can verify specific claims against their issuers or evaluate attester trust. Evaluators claiming Tier 1 assurance **MUST** enforce attester trust policy and freshness/expiry checks on the proof material used for Tier 1 acceptance. Evaluators **MUST** validate the `claimProof` proof chain or the attester's cross-DID binding attestation before granting Tier 1 trust. Tier 1 additionally requires a verified `holderBinding` on each claim relied upon (Section 6.6.1): `claimProof` proves issuance, while `holderBinding` proves the presenting subject is the credentialed holder. Suitable for medium-stakes use cases (social identity, reputation, basic access control).

Tier 2 — Cryptographic binding. Medium friction. Cross-DID control is cryptographically proven via zero-knowledge proof of cross-DID control. The subject demonstrates control of multiple DIDs without revealing private key material, using a ZK proof that the same entity controls the keys behind each DID. Evaluators claiming Tier 2 assurance **MUST** verify the ZK proof before granting Tier 2 trust. Suitable for high-stakes Sybil-resistance use cases. See Section 6.4.7 for the proof profile preview.

Tier 3 — Multi-party ceremony. High friction. Multiple keyholders (potentially different people, different locations) must co-sign. Analogous to multi-sig wallets. Suitable for the highest-stakes organizational and financial contexts. See Section 6.4.8 for the ceremony model preview.

Evaluators **MUST** define their required trust tier based on their threat model. Evaluators **MUST NOT** extend trust from one DID in a manifest to another DID in the same manifest unless binding proof material (Tier 1 or Tier 2) is present for that specific DID pair.

6.4.3. `claims[].claimProof` Field

The `claimProof` field is an **OPTIONAL** property on any claim object. It carries proof material demonstrating claim issuance to the manifest subject. This field enables Tier 1 verification.

The field is named `claimProof` rather than `evidence` to avoid collision with the W3C Verifiable Credentials Data Model v2.0 `evidence` property (VCDM section 9.2), which has overlapping but distinct semantics.

`claimProof` **MAY** be:

- A **string** (URI reference to a VP or attestation endpoint);
- An **object** (an embedded VP, attestation proof, or proof entry); or
- An **array** of proof entry objects, each independently verifiable. This form supports claims backed by multiple independent proofs (e.g., issuer signature plus notary counter-signature, or proofs from two independent attesters).

Existing manifests with a single-value `claimProof` (string or object) remain valid. The array form is additive and non-breaking.

6.4.3.1. Proof Entry Fields

When `claimProof` is an object or an array element, the following **OPTIONAL** fields **MAY** appear on each proof entry:

- `proofType` -- Declares the proof mechanism. RECOMMENDED values: `VerifiablePresentation`, `DataIntegrityProof`, `sd-jwt-kb`, `pop-jws`, `evidence-pointer`. Extensible for integration-lane-specific types. If absent, evaluators **SHOULD** infer the type from the proof structure (e.g., an object with `@type: "VerifiablePresentation"` implies type `VerifiablePresentation`; a URI string implies `evidence-pointer`).
- `proofPurpose` -- Declares the verification relationship this proof satisfies, per W3C DID Core Section 5.3. RECOMMENDED values: `assertionMethod`, `authentication`, `keyAgreement`, `capabilityDelegation`. If absent, evaluators **SHOULD** assume `assertionMethod` (the default for VC issuance).

Proof entries **MAY** contain additional properties (e.g., `@type`, `verifiableCredential`, `proofValue`, `verificationMethod`, `statusRef`). Evaluators **MUST** preserve unrecognized properties.

6.4.3.2. Verification Procedure

When claiming Tier 1 assurance or higher, the evaluator **MUST** perform the verification steps below. At Tier 0, these checks are **OPTIONAL**.

When present as an embedded object, the evaluator **MUST** verify the VP proof chain: (a) VC signature validates to the stated issuer, (b) VP signature validates to the holder, (c) holder DID matches the manifest subject.

When present as a URI string, the evaluator **MAY** fetch the VP for verification when network access is available. Evaluators that cannot resolve the URI **SHOULD** record the claim as `"unverified"` with a reason such as `um:reason:trust:claimproof-unresolved`, rather than `"verified"` (`claimStatuses`, Section 3.3.1).

When `claimProof` is an array, each entry is independently verified. The claim is backed by the conjunction of all valid proofs.

VPs used as `claimProof` **SHOULD** include domain (audience binding) and challenge (nonce) parameters to prevent cross-manifest replay.

Evaluators **MUST** enforce size limits of at most 50 KB per embedded VP and at most 500 KB total VP payload across all claims in one manifest, and **MUST** document any stricter limit in their conformance claim.

6.4.4. `identity.crossDidBinding` Claim

The `identity.crossDidBinding` claim type provides a pragmatic, trust-delegated mechanism for asserting that multiple DIDs are controlled by the same entity. It works within the existing `claims[]` array and requires no schema changes.

Example 15: Cross-DID binding claim

```
{
  "@type": "identity.crossDidBinding",
  "issuer": "did:web:verify.example",
  "boundDids": ["did:key:z6MkAlice", "did:plc:alice-bsky"],
  "attester": "did:web:verify.example",
  "attestationMethod": "AT Protocol handle resolution",
  "attestedAt": "2026-03-15T10:30:00Z",
  "expiresAt": "2026-06-15T10:30:00Z"
}
```

A `crossDidBinding` claim **MUST** contain the following fields:

- `@type` — **MUST** be `"identity.crossDidBinding"`.
- `issuer` — DID or URI identifying the entity that asserts the claim. **MUST** be present (inherited from the base claim schema, Section 1.4.3).
- `boundDids` — Array of DID strings asserted as controlled by the same entity. **MUST** contain at least 2 DIDs. One **MUST** match the manifest `subject`.

An **attester-asserted** binding — one offered for Tier 1 evaluation on the strength of an attester's assertion rather than a cryptographic proof — **MUST** additionally contain the following three fields. When the claim instead carries a cryptographic `bindingProof` (Section 6.4.7) or `ceremonyProof` (Section 6.4.8), they are **OPTIONAL**: the proof object, not an attester assertion, carries the binding.

- `attester` — DID or URI of the entity attesting the binding.
- `attestationMethod` — Human-readable description of the verification method used.
- `attestedAt` — RFC 3339 timestamp of when the attestation was produced.

A `crossDidBinding` claim **MAY** contain the following fields:

- `claimProof` — URI, structured object, or array of proof entries pointing to the attestation proof (see Section 6.4.3).

- `expiresAt` — Attestation expiry. Evaluators **SHOULD** reject expired attestations.
- `bindingProof` — A Tier 2 zero-knowledge proof of cross-DID control (Section 6.4.7, PREVIEW).
- `ceremonyProof` — A Tier 3 multi-party ceremony proof (Section 6.4.8, PREVIEW).

Evaluators **MUST NOT** treat the presence of a binding claim as proof of common control unless they trust the attester (for attester-asserted bindings) or have verified its cryptographic proof (Section 6.4.7, Section 6.4.8). Evaluators **SHOULD** maintain a configurable attester trust list. Multiple binding claims for overlapping DID sets are independent assertions, not cumulative proof.

6.4.5. `requiredTrustTier` Declaration

A manifest **MAY** declare the minimum trust tier required for specific claims, facets, or the manifest as a whole via the `requiredTrustTier` field. This field is an integer (0–3) indicating the minimum verification tier an evaluator **MUST** satisfy before acting on the associated data.

- **Manifest-level:** a top-level `requiredTrustTier` sets the floor for the entire manifest.
- **Claim-level:** a `requiredTrustTier` on an individual claim applies to that claim only.
- **Facet-level:** a `requiredTrustTier` on a facet applies to that facet only.

If a claim carries `requiredTrustTier: 2` but the evaluator can only verify at Tier 1, the evaluator **MUST** treat that claim as unverified. If absent, the default is 0 (no minimum required). The manifest-level value sets the floor; claim/facet-level values can only raise it, not lower it.

If a manifest, claim, or facet specifies a `requiredTrustTier` value for which the evaluator has no implemented verification profile (e.g., Tier 3, where the ceremony profile is under development — see Section 6.4.8), the evaluator **MUST** treat the item as unverifiable and record it in the receipt as `"trustTierUnsupported"` — in `claimStatuses` for claims, in `facetStatuses` for facets, and in `crossDidBindingStatus` for binding claims (Section 3.3.1). The evaluator **MUST NOT** downgrade to a lower tier. The evaluation-sequence outcome for the overall manifest is `"accepted-partial"` if other items can still be processed, or `"rejected"` if the unsupported tier applies at the manifest level.

Example 16: Manifest with requiredTrustTier

```
{
  "requiredTrustTier": 1,
  "claims": [
    {
      "@type": "personhood.worldId.verification",
      "issuer": "did:web:worldcoin.org",
      "requiredTrustTier": 2
    }
  ]
}
```

6.4.6. Bilateral Exchange

In any transaction or interaction, both parties present a Universal Manifest to each other. Trust verification is inherently bilateral:

- Alice presents her UM to a venue; the venue verifies Alice's claims at the trust tier the venue requires.
- The venue presents its UM to Alice; Alice verifies the venue's claims at the trust tier Alice requires.
- A peer-to-peer exchange has both sides presenting and verifying simultaneously.

The *interaction tier floor* for an exchange is the maximum of what either party demands; it is a negotiated floor, distinct from the `effectiveTrustTier` each party verifies and records (Section 3.3.1.1). Asymmetric requirements are valid — each party sets its own `requiredTrustTier` independently. Two devices **MAY** exchange manifests via local transport (NFC, BLE, QR) and each independently verify the other's claims at the declared tier without a server.

When asymmetric verification outcomes occur (for example, one party cannot satisfy the other party's required tier), each party **MUST** evaluate policy independently. For Sybil-critical or otherwise high-risk actions, parties **MUST** fail closed (deny the action) when required tier checks are not satisfied. For lower-risk actions, parties **MAY** degrade to a restricted mode that excludes trust-transitive or high-impact operations.

6.4.7. Tier 2 ZKP Proof Profile PREVIEW

This section defines the zero-knowledge proof profile for Tier 2 trust verification: cryptographic proof that the same entity controls multiple DIDs without revealing private key material or linking secrets.

v0.4 provides two proof profiles for Tier 2 cross-DID binding. Both use the `bindingProof` field on `identity.crossDidBinding` claims.

6.4.7.1. Profile 2A: BBS+ Linked-Secret Proof

The holder possesses a secret `s` that is committed in the credentials of both DIDs. The ZK proof demonstrates equality of committed secrets across credentials without revealing `s`. This follows the AnonCreds link-secret model adapted for Universal Manifest.

Evaluators **MUST** verify the BBS+ proof of equality against the issuer's BLS12-381 G2 public keys for both credentials. The proof inherently demonstrates that the presenter possesses the holder-committed secret in both credentials.

A Profile 2A `bindingProof` **MUST** contain:

- `type` -- **MUST** be `"ZkLinkedSecretProof"`.
- `cryptosuite` -- **MUST** be `"bbs-2023"` per the W3C Data Integrity BBS Cryptosuites.
- `proofPurpose` -- **MUST** be `"authentication"`.
- `proofValue` -- Base64url-encoded BBS+ derived proof bytes.
- `publicInputs` -- An object containing `commitmentA` and `commitmentB`, the BBS+ commitments from each credential.

BBS+ signatures ([BBS+]) are the **RECOMMENDED** cryptographic suite for privacy-preserving Tier 2 operations. BBS+ derived proofs are unlinkable across presentations, and selective disclosure is native to the signature scheme.

6.4.7.2. Profile 2B: HD Derivation Proof

Both DIDs are derived from a common master seed via hierarchical deterministic key derivation (BIP-32/SLIP-10). The ZK proof demonstrates that a master seed exists such that the two public keys are derivable at declared paths, without revealing the seed.

A Profile 2B `bindingProof` **MUST** contain:

- `type` -- **MUST** be `"ZkHdDerivationProof"`.
- `proofSystem` -- The ZK proof system used (e.g., `"groth16"`). Evaluators **MUST** support Groth16 [Groth16]; **MAY** support PLONK or Bulletproofs.
- `circuit` -- URI identifying the HD derivation circuit (e.g., `"urn:uuid:circuit-hd-derivation-v1"`).
- `proofValue` -- Base64url-encoded proof bytes.
- `publicInputs` -- An object containing `publicKeyA`, `publicKeyB`, `derivationPathA`, and `derivationPathB`.

Groth16 proofs require a per-circuit trusted setup ceremony. The ceremony parameters **MUST** be publicly auditable. Evaluators **MUST** verify proofs against the published verification key.

6.4.7.3. Verification Procedure

When an evaluator encounters an `identity.crossDidBinding` claim carrying a `bindingProof`, the evaluator **MUST** apply the following procedure (the declared `requiredTrustTier` governs what the verified result must satisfy, not whether the proof is verified):

1. Identify the proof type from `bindingProof.type`.
2. For `ZkLinkedSecretProof`: verify the BBS+ derived proof against the issuer's BLS12-381 G2 public keys referenced in the claim's credentials. Confirm that the proof demonstrates equality of the committed holder secret.
3. For `ZkHdDerivationProof`: resolve the circuit URI, retrieve the verification key, construct the public inputs from the declared public keys and derivation paths, and verify the ZK proof.
4. Record the result in the receipt. A verified proof elevates the claim to Tier 2. A proof that *fails* verification **MUST** cause the claim to be recorded as `failed` and treated at Tier 0, regardless of any accompanying attestations; evaluators **MAY** apply stricter local policy (for example, rejecting the manifest). Only an *absent* `bindingProof` falls back to attester-asserted Tier 1 evaluation.

Example 17: Tier 2 cross-DID binding with linked-secret proof

```
{
  "@type": "identity.crossDidBinding",
  "issuer": "did:key:z6MkAlice",
  "boundDids": ["did:key:z6MkAlice", "did:pkh:eip155:1:0xabc"],
  "requiredTrustTier": 2,
  "bindingProof": {
    "type": "ZkLinkedSecretProof",
    "cryptosuite": "bbs-2023",
    "proofPurpose": "authentication",
    "proofValue": "u2V0BhV...",
    "publicInputs": {
      "commitmentA": "z3tC...",
      "commitmentB": "z7dF..."
    }
  }
}
```

Preview — built on default: This preview makes **Groth16** the mandatory-to-implement proof system for Profile 2B (smallest proofs, fastest verification), with PLONK permitted as an optional alternative for deployments that prefer a universal setup. Working-group input is requested on whether the trusted-setup cost of Groth16 justifies making PLONK the baseline instead; flag to revise.

6.4.8. Tier 3 Multi-Party Ceremony PREVIEW

This section defines the multi-party ceremony model for Tier 3 trust verification: co-signing by multiple independent keyholders to provide the highest-assurance identity binding.

6.4.8.1. Ceremony Model

A Tier 3 ceremony is a protocol by which multiple independent attesters jointly verify and co-sign a cross-DID binding. The ceremony produces a single aggregate proof represented in a `ceremonyProof` object on the `identity.crossDidBinding` claim.

A `ceremonyProof` **MUST** contain:

- `type` -- **MUST** be `"ThresholdAttestationProof"`.
- `threshold` -- A string in the format `"M-of-N"` (e.g., `"3-of-5"`) declaring the quorum requirement.
- `attesters` -- An array of DID strings identifying the ceremony participants who contributed to the aggregate proof. The array **MUST** contain at least `M` entries.
- `ceremonyId` -- A globally unique URI identifying this ceremony instance.
- `aggregateProof` -- The aggregate proof value. The encoding depends on the threshold protocol used.

6.4.8.2. Signer-Role Taxonomy

Ceremony participants **MAY** declare roles. The following roles are recognized:

- `"subject"` -- The manifest subject, asserting their own identity.
- `"witness"` -- An independent party who attests to the subject's identity through direct verification.
- `"custodian"` -- A keyholder who holds a key share on behalf of an organization.
- `"auditor"` -- A party who reviews the ceremony transcript and co-signs as a quality check.

Profile documents **MAY** define role constraints (e.g., "at least 1 witness and 1 custodian"). When role constraints are declared, evaluators **MUST** verify that the attester list satisfies them.

6.4.8.3. Threshold Protocol Selection

v0.4 defines the ceremony slot (the `ceremonyProof` schema) and proof format. The specific threshold protocol used to produce the aggregate proof is deferred to profile documents. Implementations **MAY** use any threshold protocol that produces a verifiable aggregate proof.

> **Non-normative guidance:** FROST (Flexible Round-Optimized Schnorr Threshold signatures, [RFC9591]) is a candidate threshold protocol for Ed25519-based ceremonies. FROST produces standard Schnorr/Ed25519 signatures from a threshold of signers, and verifiers see a normal signature. Threshold BBS+ (Doerner et al. 2023) is a candidate for anonymous credential issuance ceremonies. Neither protocol is normatively required; profile documents define the binding.

6.4.8.4. Verification Procedure

When an evaluator encounters an `identity.crossDidBinding` claim with `requiredTrustTier: 3` and a `ceremonyProof`, the evaluator **MUST**:

1. Parse the `threshold` string to extract `M` and `N`.
2. Verify that the `attesters` array contains at least `M` entries.
3. If role constraints are declared in the profile, verify that the attester list satisfies them.
4. Verify the `aggregateProof` against the declared attesters using the threshold protocol specified by the profile.
5. For each attester DID, confirm the attester is on the evaluator's configurable trust list (per Section 6.4.4).
6. Record the result. A verified ceremony elevates the claim to Tier 3. A `ceremonyProof` that *fails* verification **MUST** cause the claim to be recorded as `failed` and treated at Tier 0, regardless of any accompanying attestations; only an *absent* `ceremonyProof` falls back to lower-tier evaluation of whatever binding material is present.

Evaluators that do not implement any Tier 3 threshold protocol **MUST** record the claim as `"trustTierUnsupported"` per Section 6.4.5.

Example 18: Tier 3 ceremony proof

```
{
  "@type": "identity.crossDidBinding",
  "issuer": "did:key:z6MkAlice",
  "boundDids": ["did:key:z6MkAlice", "did:pkh:eip155:1:0xabc"],
  "requiredTrustTier": 3,
  "ceremonyProof": {
    "type": "ThresholdAttestationProof",
    "threshold": "3-of-5",
    "attesters": [
      "did:web:notary-a.example",
      "did:web:notary-b.example",
      "did:web:notary-c.example"
    ],
    "ceremonyId": "urn:uuid:ceremony-2026-05-30-abc",
    "aggregateProof": "z..."
  }
}
```

Note: The working group is evaluating whether the ceremony model should support asynchronous signing (signers sign at different times) or require synchronous co-presence. Asynchronous ceremonies are more practical for geographically distributed keyholders. Feedback is requested.

6.4.9. Claim Proof Process -- End-to-End Verification Path

This section defines the normative end-to-end procedure an evaluator follows to determine whether a claim carried in a manifest is trustworthy at Tier 1 or above. It ties together the outer manifest signature verification (Section 1.6), the `claimProof` field (Section 6.4.3), and the cross-DID binding attestation (Section 6.4.4) into a single verification chain.

6.4.9.1. Verification Chain

For each claim requiring Tier 1 or higher trust, the evaluator **MUST** execute the following seven-step procedure:

1. **Verify outer manifest signature** per Section 1.6. If the signature is invalid, the entire manifest is rejected.
2. **Confirm manifest signing key authorization.** Resolve the manifest `subject` DID document. Confirm that the signing key (referenced via `signature.keyRef`) appears in the subject's DID document under the `authentication` or `assertionMethod` verification relationship, per [DID-CORE] Section 5.3. If the manifest signing key is not authorized for the subject, the evaluator **MUST** record this as a verification failure.
3. **Resolve the claim issuer DID.** For each claim carrying `claimProof`, resolve the `claims[].issuer` DID document per W3C DID Resolution.
4. **Verify the claim proof chain.** If `claimProof` is present:
   a. Identify the proof type. If `proofType` is absent, infer from the proof structure (e.g., an object with `@type: "VerifiablePresentation"` implies type `VerifiablePresentation`).
   b. Verify the proof material per the declared proof type, following the VP proof chain verification in Section 6.4.3.
   c. If `verificationMethod` is present, resolve the URI to the issuer's DID document. Confirm the key appears under the `assertionMethod` verification relationship. This is the key-authorization step: it proves the signing key was authorized by the stated issuer DID to issue credentials. (See [DID-CORE] Section 5.3 and [VC-DATA-MODEL] Section 6.)
   d. If `statusRef` is present on the proof entry, check revocation status per Section 3.4.
   e. If `claimProof` is an array, each entry is independently verified. The claim is backed by the conjunction of all valid proofs.
5. **Verify attester key authorization** (for attester-asserted `identity.crossDidBinding` claims; skipped when the binding carries only a cryptographic `bindingProof` or `ceremonyProof`, whose verification is defined in Section 6.4.7 and Section 6.4.8). Resolve the `attester` DID document. Confirm the attester's signing key appears under the `assertionMethod` verification relationship. Confirm the attester is on the evaluator's configurable trust list per Section 6.4.4.
6. **Confirm key revocation status.** For all signing keys encountered in steps 1-5 (manifest signer, claim issuer, attester), confirm that no key relied upon has been revoked, to the extent revocation information is available. Key revocation is determined per key source: for DID-based keys, the verification-method state in the key's DID document; for keys carried by a credential, the proof entry's own `statusRef` (Section 6.4.3). This is distinct from the manifest-instance status resolved via `signature.statusRef` (Section 3.4). Inner key revocation (claim-issuer and attester keys) **SHOULD** be checked when revocation information is available.

7. **Compose per-claim outcomes.** Aggregate the verification outcomes into the manifest's trust decision per Section 6.4.5 (`requiredTrustTier` floors) and emit the receipt per Section 3.3.

6.4.9.2. Key Purpose Requirements

For each step in the verification chain that names a verification relationship, the evaluator **MUST** confirm the signing key appears under the matching W3C DID Core verification relationship:

- **Manifest signature:** `authentication` or `assertionMethod` on the manifest subject's DID document.
- **VC issuance (claimProof embedded VP):** `assertionMethod` on the VC issuer's DID document.
- **Cross-DID binding attestation:** `assertionMethod` on the attester's DID document.
- **Agent delegation signing:** `capabilityDelegation` on the delegator's DID document (when applicable; see Section 6.5).

6.4.9.3. Trust Policy vs. Cryptographic Verification

The verification chain separates two concerns that **MUST NOT** be conflated:

- **Cryptographic verification** (steps 1-2, 4-6): mathematical proof that signatures are valid and keys are authorized. This is deterministic.
- **Trust policy** (step 5 trust list, step 7 composition): which DIDs and attesters does the evaluator trust? This is local configuration. The specification **MUST NOT** mandate a centralized trust registry. Each evaluator defines its own trust policy.

> **Non-normative note:** OMATrust's Key Binding attestation type operationalizes the key-authorization pattern described in this section for the EVM ecosystem. UM cites W3C DID Core and VCDM 2.0 as the normative sources for the verification-relationship model.

> **Note:** Section 6.4.9 closes the end-to-end claim-proof verification gap: prior versions defined the individual proof fields but not the single normative chain that ties outer-signature, claim-issuer, and attester key-authorization checks together. (See Acknowledgements.)

6.5. Agent Delegation

The `um:agentDelegation` pointer type declares when a manifest subject has delegated session authority to an AI agent, bot, or proxy. This enables platforms to distinguish human-controlled sessions from delegated ones.

The `um:agentDelegation` pointer is placed in the manifest's `pointers` array.

6.5.1. Structure

Required fields:

- `@type` — **MUST** be `"um:agentDelegation"`.
- `delegateType` — One of `"ai-agent"`, `"bot"`, `"proxy"`, or `"human-delegate"`.
- `delegatedBy` — DID of the delegating subject. **MUST** match the manifest `subject`.
- `delegatedAt` — RFC 3339 date-time when delegation was granted.
- `expiresAt` — RFC 3339 date-time when delegation expires. Evaluators **MUST** reject expired delegations.

Optional fields:

- `delegateId` — DID or identifier of the delegate entity. **REQUIRED** whenever the delegation is intended to be exercised.
- `scope` — Array of capability strings the delegate may exercise.
- `livenessEndpoint` — URI for real-time liveness/delegation status queries.

Delegation defaults fail closed. When `scope` is absent or empty, evaluators **MUST** treat the delegation as granting **no** capabilities. When `delegateId` is absent, evaluators **MUST NOT** attribute the delegation to any specific agent and **MUST NOT** grant delegated authority on its basis. A delegation pointer referenced by `actorState.delegationRef` **MUST** carry an `@id` so that the reference resolves.

Example 19: Agent delegation pointer

```
{
  "@type": "um:agentDelegation",
  "delegateType": "ai-agent",
  "delegateId": "did:key:z6MkAgentBot",
  "delegatedBy": "did:key:z6MkAlice",
  "delegatedAt": "2026-04-01T10:00:00Z",
  "expiresAt": "2026-04-01T11:00:00Z",
  "scope": ["spatial.session", "social.messaging"]
}
```

6.5.2. Platform Guidance

Evaluators SHOULD display delegation status to other users when present. Evaluators MAY require human-only sessions for high-stakes actions (financial, governance). Evaluators MAY query the `livenessEndpoint` for real-time status when available. Evaluators MUST treat the delegation pointer as static for the manifest's TTL if `livenessEndpoint` is absent.

6.5.3. Agent Delegation Scope Registry PREVIEW

This section defines a registry of recognized scope values for the `scope` field in `um:agentDelegation` pointers (Section 6.5.1), along with a namespace convention and registration procedure.

6.5.3.1. Namespace Convention

Scope values MUST follow a dot-separated naming convention: `<domain>.<capability>`. The domain identifies the functional area; the capability identifies the specific authority granted. Examples: `spatial.session`, `social.messaging`, `commerce.transaction`.

Custom scope values defined by profile documents SHOULD use a reverse-DNS prefix to prevent collisions (e.g., `com.example.custom-capability`).

6.5.3.2. Core Scope Values

The following scope values are defined by this specification. All evaluators recognizing `um:agentDelegation` pointers **MUST** understand these values:

- `spatial.session` -- Authority to participate in spatial computing sessions on behalf of the subject.
- `spatial.navigation` -- Authority to navigate spatial environments on behalf of the subject.
- `social.messaging` -- Authority to send and receive messages on behalf of the subject.
- `social.presence` -- Authority to represent the subject's presence status.
- `commerce.transaction` -- Authority to initiate transactions on behalf of the subject, within any spending limits declared in the manifest.
- `identity.attestation` -- Authority to present the subject's identity claims to verifiers. Does **not** grant authority to create or modify claims.

6.5.3.3. Evaluator Behavior for Unrecognized Scopes

When an evaluator encounters a delegation pointer with scope values it does not recognize, the evaluator **MUST**:

1. Record the unrecognized scope values in the receipt.
2. Restrict the delegate to only the recognized scopes. The delegate **MUST NOT** exercise capabilities corresponding to unrecognized scopes.
3. If the delegation pointer contains **only** unrecognized scopes, the evaluator **SHOULD** treat the delegation as having no effective scope and record this in the receipt.

6.5.3.4. Registration Procedure

New scope values are registered through the profile registration mechanism (Section 5.1). A scope registration **MUST** include: the scope string, a human-readable description, the domain it governs, and any constraints on its use (e.g., "requires `requiredTrustTier` >= 1").

> **Non-normative note:** Mastercard's Verifiable Intent pattern uses a three-layer SD-JWT structure (issuer-bound credential, user-signed intent, agent-signed fulfillment) for bounded-scope agent delegation. This pattern is compatible with the scope registry: each layer maps to a scope constraint in the delegation pointer.

> **Note:** The working group is evaluating whether scope values should use URI-based naming for stronger uniqueness guarantees. Feedback is requested.

6.6. Credential Binding Security Considerations

v0.4 adds cryptographic binding layers to the Universal Manifest. This section defines the security properties these mechanisms provide and the threats they mitigate.

6.6.1. Holder Binding

Manifests without `holderBinding` on their claims **MUST NOT** be treated as providing identity assurance above Tier 0, regardless of the trust tier declared by the holder.

The `holderBinding` object supports three modes:

- `"sd-jwt-kb"` -- SD-JWT Key Binding per [RFC9901]. The credential contains a `cnf` (confirmation) claim ([RFC7800]) with a JWK Thumbprint ([RFC7638]) binding the credential to a specific holder key. The `cnfThumbprint` field is **REQUIRED** when this mode is used. This is the mandatory-to-implement baseline.
- `"bbs-holder-commitment"` -- BBS+ holder binding per the W3C Data Integrity BBS Cryptosuites. The issuer commits to a holder-private key during credential issuance. The holder derives a zero-knowledge proof demonstrating possession of the bound key without revealing it. Presentations are unlinkable. The `pseudonymScope` field is **OPTIONAL**; when present, it enables scope-exclusive pseudonyms per verifier.
- `"reciprocal-control"` -- Direct DID binding. Both the manifest subject and the bound DID sign the same challenge (the manifest `@id`). The `boundDid`, `subjectProof`, and `boundDidProof` fields are **REQUIRED**.

Every claim intended to be relied upon at Tier 1 or above **MUST** carry `holderBinding`. The `sd-jwt-kb` mode is mandatory-to-implement. The `bbs-holder-commitment` mode is **RECOMMENDED** for privacy-sensitive deployments.

For each mode, the evaluator performs the following verification procedure during Stage-2 sub-step 2a (Section 3.1.2):

- **`sd-jwt-kb`.**
    1. Locate the SD-JWT and its Key Binding JWT (KB-JWT) in the claim's `claimProof`.

2. Confirm the credential's `cnf` JWK thumbprint ([RFC7638]) equals the declared `cnfThumbprint`.
3. Validate the KB-JWT signature with the confirmed holder key.
4. Confirm the KB-JWT `aud` equals the presentation `audience` and its `nonce` equals the presentation `challenge` (Section 6.6.2), and that its `sd_hash` covers the disclosed credential.
5. On any failure, record `holderBindingStatus: "failed"` and cap the claim at Tier 0.

- **`bbs-holder-commitment`.** The binding carries the **REQUIRED** fields `commitment` (the issuer's commitment to the holder secret) and `proofValue` (the holder's zero-knowledge proof of possession), and the **OPTIONAL** `pseudonymScope` ([VC-DI-BBS]).
  1. Verify the BBS+ derived `proofValue` demonstrates possession of the secret committed in `commitment`, against the issuer's BLS12-381 public key.
  2. If `pseudonymScope` is present, confirm the pseudonym is bound to that scope (and not reused across verifiers, Section 6.6.4).
  3. On failure, record `holderBindingStatus: "failed"` and cap the claim at Tier 0.
- **`reciprocal-control`.** Confirm `subjectProof` and `boundDidProof` are each valid signatures over the manifest `@id` by the manifest `subject` key and the `boundDid` key respectively; on failure record `holderBindingStatus: "failed"`. (See the limitation in Section 6.6.6.)

Verifying an `sd-jwt-kb` binding presupposes an interactive presentation: step 4 compares the KB-JWT `aud` and `nonce` against the `audience` and `challenge` of the presentation (Section 6.6.2). In non-interactive contexts (offline or cached presentation), holder binding at Tier 1 or above is carried by the `bbs-holder-commitment` or `reciprocal-control` modes, which verify without a verifier-issued challenge.

6.6.2. Presentation Proof

The `presentationProof` field at the manifest root provides proof-of-possession at verification time, binding the manifest to a specific verifier and moment. It prevents manifest replay.

When presenting a manifest in response to a verifier-issued challenge (an *interactive presentation*, see Terminology), the presenter **MUST** include a `presentationProof` containing:

- `proofType` -- One of `"sd-jwt-kb"`, `"bbs-derived"`, or `"did-auth"`.
- `challenge` -- The verifier-issued nonce. **MUST** be unique per presentation request.
- `audience` -- The verifier's identifier (DID or origin). **MUST** match the requesting verifier.

- `created` -- RFC 3339 timestamp of proof generation.
- `proofValue` -- Base64url-encoded proof (KB-JWT, BBS+ derived proof, or DID Auth signature).

`presentationProof` is excluded from the manifest signing input (removed alongside `signature` in step 2 of Section 1.6.3). Its `proofValue` **MUST** be computed over the concatenation of: the manifest's signing-input hash (SHA-256 of the Section 1.6.3 canonical bytes, as its 32 raw bytes), then the UTF-8 bytes of `challenge`, `audience`, and `created`, in that order, per the mechanics of the declared `proofType` (KB-JWT per [RFC9901]; BBS+ derived proof per [VC-DI-BBS]; DID Auth signature per [DID-CORE]). Binding the proof to the signing-input hash ties the presentation to the exact signed manifest without requiring the holder to re-sign the manifest per presentation.

A manifest presented without a `presentationProof` is valid only for offline or cached identity display, not for interactive verification. When the evaluator itself issued a challenge for this presentation and `presentationProof` is absent, the evaluator **MUST** treat the manifest as replay-suspect, **MUST NOT** accept it for interactive verification, and records `presentationProofStatus: "missing-required"` (Section 3.3.1.1).

6.6.3. Liveness Attestation

A `livenessAttestation` proves human presence at `attestedAt`. It does **not** prove human presence at verification time. Evaluators **MUST NOT** treat a stale liveness attestation as equivalent to current human presence.

Four freshness classes are defined. An attestation past its `validUntil` is `"unknown"` regardless of `attestedAt`; otherwise the classes are evaluated in the order listed and the first match applies:

- `"live"` -- Attested within 60 seconds. Suitable for high-stakes transactions.
- `"recent"` -- Attested within 4 hours. Suitable for spatial platform entry.
- `"stale"` -- Attested more than 4 hours ago but within the attestation's `validUntil`. Evaluators **SHOULD** require re-attestation for sensitive operations.
- `"unknown"` -- No liveness attestation present (default for offline mode), or an attestation whose `validUntil` has passed. An attestation past its `validUntil` **MUST** be treated as `"unknown"`.

The `livenessAttestation` is a root-level member of the manifest (see the abstract data model, Section 1.7.1, which carries it among the optional credential-binding members). When present it **MUST** contain:

- `proofType` — A string identifying the liveness method (for example `"webauthn-uv"`); extensible via the profile registry (Section 5.1). **REQUIRED**.
- `attestedAt` — RFC 3339 date-time at which human presence was attested. **REQUIRED**.
- `validUntil` — RFC 3339 date-time after which the attestation is treated as `"unknown"`. **REQUIRED**.
- `method` — A string naming the authentication method used (copied into the receipt `livenessStatus`). **OPTIONAL**.
- `userVerified` — A boolean indicating whether user verification (e.g. biometric or PIN) was performed (copied into the receipt). **OPTIONAL**.
- `proofValue` — Base64url-encoded proof. **REQUIRED**. The `proofValue` **MUST** be computed over the UTF-8 bytes of the manifest `subject` concatenated with the UTF-8 bytes of `attestedAt`, signed by the attesting authenticator or platform key.
- `attester` — The DID of the attesting authority. **OPTIONAL**.

WebAuthn user-verification (a WebAuthn Level 3 [WEBAUTHN] assertion with the user-verified flag set, `proofType: "webauthn-uv"`) is the **RECOMMENDED** liveness method. Evaluators verifying a `livenessAttestation` **MUST** validate `proofValue` against the named `proofType`'s mechanics before treating the attestation's freshness class as evidence of human presence.

6.6.4. BBS+ Pseudonym Scope

If `pseudonymScope` is reused across verifiers, cross-verifier correlation becomes possible. Issuers **SHOULD** generate unique scope URIs per verifier relationship. Evaluators **MUST NOT** use a shared pseudonym scope for multiple independent verifier relationships.

6.6.5. ZK Proof System Trusted Setup

Tier 2 profiles using Groth16 require a trusted setup ceremony. The ceremony parameters **MUST** be publicly auditable. Verifiers **MUST** verify proofs against the published verification key, not against a key provided by the prover.

6.6.6. Reciprocal Control Limitations

Reciprocal control (two DIDs signing the same challenge) proves the same entity or cooperating entities control both keys. It does **not** prove they are the same natural person. For `sameSubject`

assurance, combine reciprocal control with issuer-attested binding (Tier 1) or linked-secret proof (Tier 2).

6.6.7. Backwards Compatibility

v0.4 evaluators encountering a v0.3 manifest (no `holderBinding`, no `presentationProof`, no `livenessAttestation`) **MUST**:

1. Parse and process the manifest through the six-stage evaluation sequence.
2. Record `holderBindingStatus: "absent"` in the receipt.
3. Record `presentationProofStatus: "absent"` in the receipt.
4. Record `livenessStatus.freshnessClass: "unknown"` in the receipt.
5. Cap the effective trust tier at Tier 0 regardless of `requiredTrustTier` declarations.

A v0.3 manifest that declares no manifest-level `requiredTrustTier` (or declares Tier 0) is accepted but flagged as "unbound" at Tier 0; evaluators decide their own policy for unbound manifests. Where a v0.3 manifest declares a manifest-level `requiredTrustTier` that cannot be satisfied because binding material is absent, Section 6.4.5 governs the outcome: the manifest is processed with outcome `rejected` for trust-gated use, or `accepted-with-warnings` for display-only use, per local policy. Capping the effective trust tier at Tier 0 (step 5) does not by itself reject the manifest; the declared floor does.

v0.4 evaluators **SHOULD** emit a warning in the receipt when processing manifests without holder binding:

Example 20: Unbound-claims warning

```
{
  "warnings": [
    {
      "code": "um:reason:trust:unbound-claims",
      "message": "One or more claims lack holder binding. Claims are accepte
      "affectedClaims": ["urn:claim:poh", "urn:claim:avatar-control"]
    }
  ]
}
```

6.7. Post-Quantum Signatures PREVIEW

This section defines a post-quantum signature profile for the Universal Manifest, providing quantum-resistant signing as a supplement to the baseline Ed25519 profile (Signature Profile A, Section 1.6).

6.7.1. Candidate Algorithms

The following NIST-standardized post-quantum signature algorithms are candidates for the post-quantum profile:

- **ML-DSA (CRYSTALS-Dilithium)** -- NIST FIPS 204. Lattice-based, moderate signature size (~2.4 KB for ML-DSA-65), fast verification. The leading candidate for general-purpose post-quantum signatures.
- **SLH-DSA (SPHINCS+)** -- NIST FIPS 205. Hash-based, larger signatures (~8-30 KB), no lattice assumptions (conservative choice). Suitable as a fallback if lattice-based schemes are broken.
- **FN-DSA (FALCON)** -- NIST (pending standardization). Lattice-based, compact signatures (~666 bytes for FALCON-512), more complex implementation. Suitable for constrained devices where signature size is critical.

6.7.2. Profile Naming

The post-quantum profile follows the signature-profile naming convention of Section 1.6. The recommended baseline profile is **Signature Profile B: JCS + ML-DSA-65**.

6.7.3. Dual-Signature Migration Path

The transition from Ed25519 to post-quantum signatures uses a dual-signature period. During this period, manifests carry both an Ed25519 signature (in the existing `signature` field) and a post-quantum signature (in a new `postQuantumSignature` field).

- Evaluators that support the post-quantum profile MUST verify both signatures.
- Evaluators that do not yet support the post-quantum profile MUST verify only the Ed25519 signature and MUST ignore the `postQuantumSignature` field.

- The dual-signature period begins when the post-quantum profile is published and ends when the working group declares Ed25519-only manifests non-conformant at a major-version boundary.

The `postQuantumSignature` field follows the same schema as `signature` (Section 1.6) with the `algorithm` field set to the post-quantum algorithm identifier (e.g., `"ML-DSA-65"`). Both proofs are computed over the same signing input: the canonical bytes of Section 1.6.3 with the `signature`, `postQuantumSignature`, and (if present) `presentationProof` properties removed, so that neither signature's bytes feed the other's input.

Example 21: Dual-signature manifest

```
{
  "signature": {
    "algorithm": "Ed25519",
    "canonicalization": "JCS-RFC8785",
    "keyRef": "did:key:z6MkAlice#keys-1",
    "value": "base64url-ed25519-signature"
  },
  "postQuantumSignature": {
    "algorithm": "ML-DSA-65",
    "canonicalization": "JCS-RFC8785",
    "keyRef": "did:key:z6MkAlice#keys-pq-1",
    "value": "base64url-ml-dsa-signature"
  }
}
```

Note: The working group is evaluating whether the dual-signature mechanism should use a separate top-level field or an array of signature objects. The separate field is simpler for backward compatibility; an array is more extensible. Feedback is requested.

Note: ML-DSA-65 signatures (~2.4 KB) are significantly larger than Ed25519 (64 bytes). The working group is assessing whether this size increase matters for constrained-device deployments.

6.8. Category Trust and `trustWeight` PREVIEW

A recurring failure mode in service directories is treating a self-declared service category (for example, "emergency-services" or "verified-merchant") as if it were attested. v0.4 separates the *claim* of a category from its *attestation* by splitting the single self-declared category field into four distinct category-trust claim objects and by replacing the v0.1/v0.3 `interpretedAs: "hint-only"` enum with a typed `trustWeight` field that an evaluator can act on uniformly.

6.8.1. The `trustWeight` Field

Any claim or category-trust object **MAY** carry a `trustWeight` field replacing the older `interpretedAs: "hint-only"` convention. `trustWeight` is one of:

- `"hint"` — Self-asserted, unattested. Evaluators **MUST NOT** grant trust-transitive or high-impact authority on the basis of a `hint`-weighted value alone. This is the direct, typed successor to `interpretedAs: "hint-only"`.
- `"attested"` — Backed by a Verifiable Credential or attestation from a named issuer; evaluators verify the attestation per the claim-proof process (Section 6.4.9) before relying on it.
- `"authoritative"` — Issued by an identity service provider or registry the evaluator treats as authoritative for the category by local policy.

When `trustWeight` is absent, evaluators **MUST** default to `"hint"`. Replacing a boolean/enum hint marker with this typed field lets the same evaluator policy apply across categories, agent-delegation scopes, and device-capability claims, and it does so before the wire is fielded so no later breaking change is required.

Evaluators encountering the deprecated `interpretedAs` member (from v0.1/v0.3) **MUST** ignore it; in its absence `trustWeight` defaults to `"hint"`, which preserves the prior `interpretedAs: "hint-only"` semantics. The `interpretedAs` member is removed from the v0.4 vocabulary.

6.8.2. Category Trust Split

The single self-declared category is replaced by four distinct *category-trust objects*, each a typed claim with its own trust treatment. (These are claim-level objects carried in the `claims` array, distinguished by their `@type`; they are not top-level `um:Facet` objects.) Each carries the common claim members of Section 1.4.3 plus the `trustWeight` field of Section 6.8.1:

- `operatorIdentity` — The identity of the operating entity, signed by an identity service provider. **RECOMMENDED** for any non-trivial service.
- `serviceCategoryClaim` — The self-declared service category. Always `trustWeight: "hint"` unless paired with a `categoryAttestation`.
- `categoryAttestation` — A Verifiable Credential attesting the category from a recognized authority. **REQUIRED** for high-impact categories (defined at the profile level, e.g., emergency, medical, financial, child-directed).
- `urgencyClaim` — Any urgency/priority assertion, carried as a separate signed claim so that urgency cannot be smuggled in via the category field.

Normative rules: an evaluator **MUST NOT** act on a `serviceCategoryClaim` for a high-impact category without a verified `categoryAttestation`; an evaluator **MUST** verify `categoryAttestation` credentials for revocation per Section 3.4; and an evaluator **MUST** treat `operatorIdentity` and `serviceCategoryClaim` as independent (operator identity does not attest category, and vice versa). The high-impact category list is profile-level, not fixed by this specification, consistent with eIDAS 2.0 [eIDAS2] assurance-level practice.

> **Preview:** The `trustWeight` field (replacing `interpretedAs: "hint-only"`) and the four-way category-trust claim split are built into this preview on the editors' recommended default (adopt `trustWeight` in v0.4). `trustWeight` is a small but forward-compatibility-critical wire-shape change; doing it before the wire is fielded avoids a v0.5 break. Working-group input is requested on the high-impact category list and on whether `trustWeight` should be a closed enum or an extensible vocabulary.

6.9. Cryptographic Requirements Summary PREVIEW

This section consolidates the cryptographic requirements scattered across the signature profile (Section 1.6), encrypted facets (Section 2.4), the tiered trust model (Section 6.4.2), the ZKP profiles (Section 6.4.7), the ceremony model (Section 6.4.8), credential binding (Section 6.6), and post-quantum signatures (Section 6.7) into a single reference. Requirement levels are stated as mandatory-to-implement, recommended, optional, or future. This is a summary; the normative requirement for each algorithm lives in its referenced section.

Requirement level	Algorithms / cryptosuites	Reference
Mandatory-to-implement	Ed25519 over JCS-RFC8785 (Signature Profile A); SD-JWT Key Binding (KB-JWT); JWK Thumbprint; ECDH-ES+A256KW / A256GCM for encrypted facets	1.6, 6.6.1, 2.4

Requirement level	Algorithms / cryptosuites	Reference
Recommended	BBS+ signatures and BBS+ derived proofs (BLS12-381); WebAuthn Level 3 device/liveness attestation	6.4.7, 6.6.3
Optional (Tier 2 profiles) †	Groth16 (HD-derivation proofs); BBS+ linked-secret equality proofs	6.4.7
Future	FROST threshold Schnorr/Ed25519 [RFC9591]; threshold BBS+; ML-DSA / SLH-DSA / FN-DSA post-quantum signatures	6.4.8, 6.7

† Items marked "Optional (Tier 2 profiles)" are optional to implement at the specification level, but become mandatory *within* the named optional profile: Groth16 is **REQUIRED** for an implementation that supports Profile 2B (Section 6.4.7.2). The mandatory-to-implement row is likewise scoped to the module that uses each algorithm: Ed25519 over JCS-RFC8785 is unconditional for every implementation; the JWE pair is mandatory for evaluators claiming encrypted-facet support (Section 2.4.1); SD-JWT Key Binding and JWK Thumbprint are mandatory for implementations claiming credential-binding support (Section 6.6.1, Section 4.7).

> **Preview:** This consolidated cryptographic-requirements table is built into this preview on the editors' recommended default (add as a reference summary). Working-group input is requested on whether any "recommended" item (notably BBS+ for privacy-preserving Tier 2) should be elevated to mandatory-to-implement for v0.4.

6.10. Context Integrity

Term meanings — and, for compact encodings, CBOR-LD tokenization (Section 1.7.4) — depend on the content of the versioned namespace context (`https://universalmanifest.net/ns/v0.4`). The manifest signature covers the context *reference* (the URI string), not the context *document content*: if the served context changes, or an attacker controls its resolution, term semantics and CBOR-LD round-trips can change silently under a still-valid signature.

To prevent this, the versioned namespace URI identifies an **immutable** context: once published, the context document for a given version **MUST NOT** change. Evaluators **SHOULD** ship or pin the context for each supported version rather than fetching it at evaluation time, and **MUST NOT** fetch contexts from untrusted resolvers at verification time. CBOR-LD deployments **MUST** use the pinned context (Section 1.7.4), since tokenization correctness depends on byte-identical context versions on both sides.

7. Privacy Considerations

- **Opaque Identifiers**: Generating the `@id` via random-distribution UUIDv4 shields session metadata from enumerability.
- **Minimal Disclosure**: Universal Manifest acts as the fundamental passport. Holders are strongly advised against over-bundling capabilities. Only embed contextually relevant facets.
- **Subject Privacy**: Static DID correlations across disparate endpoints create persistent observation footprints. The use of pairwise / pseudonymous DIDs can significantly improve ecosystem privacy.

7.1. Selective Disclosure: Dual-Track Model PREVIEW

Universal Manifest supports selective disclosure at the manifest level through holder-controlled projection (Section 3.1.3), encrypted facets (Section 2.3), and credential-level disclosure inside claims. For credential-level selective disclosure, v0.4 defines two interoperable tracks so that deployments can choose between web-native simplicity and unlinkable privacy:

- **Track A — SD-JWT (baseline, default).** Selective disclosure of JWT/JSON credential fields using SD-JWT [RFC9901], with key binding via KB-JWT (Section 6.6.1). This is the mandatory-to-implement baseline and the default for web and JWT environments, where library support is broad and integration is simplest.
- **Track B — BBS+ derived proofs (privacy-critical).** Unlinkable selective disclosure using BBS+ derived proofs [VC-DI-BBS], where each presentation is cryptographically unlinkable to other presentations of the same credential. Track B is RECOMMENDED for privacy-critical deployments and is required to claim Tier 2 privacy-preserving binding (Section 6.4.7).

Manifest-level selective presentation composes with both tracks via facet projection, claim filtering, and pointer elision: the holder controls which facets, claims, and pointers appear in a given manifest instance, and Track A or Track B controls disclosure *within* a presented credential. Spatial-computing deployments, where the same subject may be observed across many venues, SHOULD use Track B to prevent cross-platform correlation. Deployments default to Track A where unlinkability is not required.

> **Preview:** The dual-track selective-disclosure model is built into this preview on the editors' recommended default (SD-JWT baseline; BBS+ for privacy-critical and Tier 2). Working-group input is requested on whether Track B should be mandatory for any class of spatial-computing deployment.

Privacy-Binding Tension. Universal Manifest supports both privacy-preserving pairwise DIDs and cross-DID binding mechanisms. These goals are in fundamental tension: stronger binding enables correlation, while stronger privacy prevents binding verification. The specification does not fully address this tension at the protocol level. Instead, it provides mechanisms for both and requires evaluators to define their trust tier based on their specific threat model. Evaluators **MUST NOT** require cross-DID binding unless their use case demands Sybil resistance or trust transitivity. Subjects **SHOULD** use the minimum binding tier that satisfies their evaluators' requirements.

Sealed Entries and Selective Disclosure. Encrypted facets (see Section 2.3) provide data minimization at the envelope level. Sensitive data is encrypted inline; evaluators without the appropriate decryption key observe the facet as a sealed entry — present but unreadable. Combined with holder-controlled selective disclosure (Section 3.1.3), holders control both which facets are included in a given manifest instance and which parties can decrypt specific facets. This two-layer model (selective disclosure controls visibility; encryption controls readability) enables fine-grained data minimization without requiring separate credential presentations for each evaluator. Holders **SHOULD** encrypt facets containing personal data or sensitive attributes. Evaluators **MUST NOT** infer information about the content of sealed entries from their presence, size, or metadata.

Sealed-Entry Forward Secrecy. Recipient revocation (Section 2.3.4) governs *future* issuances only. Re-encrypting on revocation cannot un-share copies already distributed: a revoked recipient retains the ability to decrypt every encrypted facet it already received. Holders **SHOULD NOT** rely on recipient revocation to withdraw access to data already conveyed.

Status-Resolution Correlation. Resolving `signature.statusRef` (Section 3.4) reveals to the status provider which evaluator is checking which manifest, and when — the classic credential phone-home correlation risk. Evaluators **SHOULD** mitigate this by respecting the `nextCheck` caching interval, by preferring herd-privacy status mechanisms such as the W3C Bitstring Status List (Section 3.4 note), by spreading queries across federated resolvers (Section 8), and by tolerating offline operation per Section 3.4.4 rather than blocking on a live status fetch.

Receipt Minimization. Receipts (Section 3.3) are durable records of where a subject presented a manifest, carrying `manifestId`, facet identifiers, timestamps, and optionally `evaluatorId`. Receipts **SHOULD** carry the minimum facet detail needed for accountability, and retention **SHOULD** be bounded. Receipt chains **SHOULD** be signed with session-scoped keys (Section 3.3.2.1) to avoid creating a long-lived correlator across unrelated interactions.

Device Identifiers. The `deviceAttestation` component (Section 1.4.6.1) carries long-lived, potentially correlatable hardware identifiers such as `modelHash` and manufacturer attestations. Holders **SHOULD** omit `deviceAttestation` from session-only or pseudonymous manifests and carry

only the session-scoped `deviceCapability` component, which uses a per-session ephemeral signing key to avoid cross-session linkage.

Data Protection. The privacy considerations in this specification identify relevant data protection provisions but do not constitute a Data Protection Impact Assessment. Deployers operating under GDPR or equivalent frameworks **MUST** conduct their own assessment for cross-DID binding operations, cross-border attester data flows, and mandatory binding requirements.

8. Federation PREVIEW

This section defines the federation model for Universal Manifest infrastructure: how multiple resolver operators coordinate to provide status checks, cache invalidation, and availability across a distributed network.

8.1. Resolver Coordination

A federated resolver network consists of two or more resolver operators that synchronize manifest status information. Resolvers **MUST** identify themselves using DIDs and **MUST** establish mutual trust through bilateral attestation (each resolver attests to the other's identity via an `identity.crossDidBinding` claim or equivalent mechanism).

Resolver discovery is out of scope for this specification. Deployments **MAY** use a well-known URI (`/.well-known/um-resolvers`), a DID service endpoint, or an out-of-band configuration.

8.2. Status Check Distribution

When an evaluator resolves a `statusRef` URI, the evaluator **MAY** query multiple federated resolvers. The following rules apply:

- If any resolver reports `"revoked"`, the evaluator **MUST** treat the manifest as revoked. Revocation is authoritative and overrides other responses.
- If resolvers disagree (one reports `"active"`, another reports `"suspended"`), the evaluator **MUST** act on the most restrictive status received (this section governs the *decision*) and **SHOULD** record a warning with reason `um:reason:status:federation-inconsistent`. The recorded `revocationStatus` reflects the most restrictive answer received, not `"unchecked"` (Section 8.4 governs the *recording*).

- If all queried resolvers are unreachable, the evaluator follows the error handling in Section 3.4.4.

8.3. Cache Invalidation

When a manifest's status changes (e.g., from `"active"` to `"revoked"`), the issuing resolver **MUST** propagate the change to all federated resolvers. The federation **SHOULD** achieve propagation within 60 seconds for revocation events. During the propagation window, different evaluators querying different resolvers may receive different answers.

Evaluators **SHOULD** treat cached status as provisional and re-check per the `nextCheck` interval returned by the status endpoint.

8.4. Availability and Failover

The federation model uses eventual consistency for status queries. This means:

- Evaluators **MAY** query any federated resolver, not only the primary `statusRef` endpoint.
- If the primary resolver is unavailable, the evaluator **SHOULD** fall back to alternative resolvers discovered through the federation.
- In split-brain scenarios (network partition between resolvers), the recorded `revocationStatus` reflects the most restrictive answer actually received from any reachable resolver (per Section 8.2), accompanied by a warning with reason `um:reason:status:federation-inconsistent`. The `"unchecked"` value is reserved for the case where *no* resolver returned an answer.

8.5. Manifest Forwarding

When a manifest is forwarded across federation boundaries, the forwarding agent **MUST** preserve all fields (per Section 3.1.1, unknown fields survive the evaluation sequence). Forwarding metadata **MUST NOT** be added to the manifest payload, because the payload's bytes are covered by the signature (Section 1.6.3). A forwarding agent **MAY** convey a `forwardedVia` array (resolver DIDs in path order) in transport-level metadata or in a wrapper object that carries the manifest unmodified. Such metadata is informational and **MUST NOT** affect evaluation outcomes.

Forwarded manifests **MUST** be re-verified by the receiving evaluator. The forwarding agent's handling of the manifest does not substitute for the receiver's own evaluation.

Note: The working group is evaluating whether the federation model should be published as a separate companion specification. Federation introduces protocol-layer complexity that may evolve independently of the manifest format. Feedback is requested.

Acknowledgements

The editors thank the contributors to the OMA3 Trustless Web Group and the Metaverse Standards Forum Digital Asset Management Working Group for review and discussion that shaped this specification. The end-to-end claim-proof verification chain (Section 6.4.9) was prompted by a gap surfaced by Alfred Tom in OMA3 TWG discussion in May 2026.

9. References

9.1. Normative References

[RFC2119]
S. Bradner. *Key words for use in RFCs to Indicate Requirement Levels*. IETF BCP 14, RFC 2119, March 1997.

[RFC8174]
B. Leiba. *Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words*. IETF BCP 14, RFC 8174, May 2017.

[RFC8785]
A. Rundgren, B. Jordan, S. Erdtman. *JSON Canonicalization Scheme (JCS)*. IETF RFC 8785, June 2020.

[RFC8032]
S. Josefsson, I. Liusvaara. *Edwards-Curve Digital Signature Algorithm (EdDSA)*. IETF RFC 8032, January 2017.

[RFC7516]
M. Jones, J. Hildebrand. *JSON Web Encryption (JWE)*. IETF RFC 7516, May 2015.

[RFC7517]
M. Jones. *JSON Web Key (JWK)*. IETF RFC 7517, May 2015.

[RFC4648]
S. Josefsson. *The Base16, Base32, and Base64 Data Encodings*. IETF RFC 4648, October 2006.

[RFC3339]

G. Klyne, C. Newman. *Date and Time on the Internet: Timestamps*. IETF RFC 3339, July 2002.

[JSON-LD]

M. Sporny, D. Longley, G. Kellogg, M. Lanthaler, N. Lindstrom. *JSON-LD 1.1*. W3C Recommendation, July 2020.

[DID-CORE]

M. Sporny, D. Longley, M. Sabadello, D. Reed, O. Steele, C. Allen. *Decentralized Identifiers (DIDs) v1.0*. W3C Recommendation, July 2022.

[INFRA]

A. van Kesteren, D. Denicola. *Infra Standard*. WHATWG / W3C. Defines the format-agnostic primitive data types (maps, lists, sets, strings, booleans, etc.) used by the abstract data model.

[RFC8949]

C. Bormann, P. Hoffman. *Concise Binary Object Representation (CBOR)*. IETF STD 94, RFC 8949, December 2020.

[CBOR-LD]

M. Sporny, D. Longley. *CBOR-LD 1.0: A CBOR-based Serialization for Linked Data*. W3C JSON-LD Working Group, Editor's Draft.

[RFC7518]

M. Jones. *JSON Web Algorithms (JWA)*. IETF RFC 7518, May 2015.

[RFC9901]

D. Fett, K. Yasuda, B. Campbell. *Selective Disclosure for JWTs (SD-JWT)*. IETF RFC 9901, November 2025.

[RFC7638]

M. Jones, N. Sakimura. *JSON Web Key (JWK) Thumbprint*. IETF RFC 7638, September 2015.

[RFC7800]

M. Jones, J. Bradley, H. Tschofenig. *Proof-of-Possession Key Semantics for JSON Web Tokens (JWTs)*. IETF RFC 7800, April 2016.

[RFC6838]

N. Freed, J. Klensin, T. Hansen. *Media Type Specifications and Registration Procedures*. IETF BCP 13, RFC 6838, January 2013.

[RFC8141]

P. Saint-Andre, J. Klensin. *Uniform Resource Names (URNs)*. IETF RFC 8141, April 2017.

9.2. Informative References

The following standards, governance documents, and prior art inform the identity binding, tiered trust model, and signature profile design in this specification.

[NIST-800-63-3]
NIST Special Publication 800-63-3: Digital Identity Guidelines. June 2017, updated 2024 supplement. Defines Identity Assurance Levels (IAL 1–3) and Authenticator Assurance Levels (AAL 1–3).

[eIDAS2]
Regulation (EU) 2024/1183: European Digital Identity Framework (eIDAS 2.0). Defines assurance levels (Low, Substantial, High) for electronic identification.

[OID4VP]
OpenID for Verifiable Presentations (OID4VP). Draft specification, OpenID Foundation (2024). Defines VP presentation with audience binding and nonce parameters.

[PE2]
Presentation Exchange 2.0. Decentralized Identity Foundation, v2.0.0 (2023). Defines how relying parties express credential/claim requirements.

[DID-CONFIG]
Well Known DID Configuration. Decentralized Identity Foundation, v0.2 (2023). Prior art for cross-DID linking via Domain Linkage Credentials.

[RFC9449]
RFC 9449: OAuth 2.0 Demonstrating Proof of Possession (DPoP). IETF, September 2023. Proof-of-possession binding for OAuth2 tokens.

[VC-STATUS]
Verifiable Credential Status List v2021. W3C CCG (2022); superseded by Bitstring Status List v1.0 (W3C, 2024). Privacy-preserving credential revocation.

[VC-DATA-MODEL]
Verifiable Credentials Data Model v2.0. W3C Recommendation, March 2025. Core data model for VCs and VPs.

[ISO-29115]
ISO/IEC 29115:2013. Entity authentication assurance framework. Four assurance levels (LoA 1–4).

[UM-BREAKING-CHANGE]
Universal Manifest Breaking-Change Policy. Project governance document.

[UM-DEPRECATION]
Universal Manifest Deprecation Policy. Project governance document.

[UM-INCIDENT]
Universal Manifest Incident Response and Rollback. Project governance document.

[UM-RELEASE]
Universal Manifest Release Cadence. Project governance document.

[UM-RFC]
Universal Manifest Spec Improvement Queue (RFC mechanism). Project governance document.

[BBS+]
The BBS Signature Scheme. IRTF Internet-Draft, draft-irtf-cfrg-bbs-signatures-10, January 2026. BBS+ multi-message signatures with selective disclosure.

[VC-DI-BBS]
Data Integrity BBS Cryptosuites v1.0. W3C Candidate Recommendation, 2025. BBS+ integration for W3C Verifiable Credentials.

[RFC9591]
C. Komlo, I. Goldberg, D. Stebila. *FROST: Flexible Round-Optimized Schnorr Threshold Signatures*. IETF RFC 9591, 2025. Threshold Schnorr/Ed25519 signatures.

[FIPS204]
FIPS 204: Module-Lattice-Based Digital Signature Standard (ML-DSA). NIST, August 2024. Post-quantum digital signature standard (CRYSTALS-Dilithium).

[FIPS205]
FIPS 205: Stateless Hash-Based Digital Signature Standard (SLH-DSA). NIST, August 2024. Post-quantum hash-based signatures (SPHINCS+).

[WEBAUTHN]
Web Authentication: An API for accessing Public Key Credentials Level 3. W3C Recommendation, 2025. Authentication ceremony and device attestation.

[RFC9162]
B. Laurie, E. Messeri, R. Stradling. *Certificate Transparency Version 2.0*. IETF RFC 9162, December 2021. Merkle-tree transparency logs.

[ISO-27560]
ISO/IEC TS 27560:2023. Privacy technologies — Consent record information structure. Informs the structured consent and receipt models.

[DPV]
Data Privacy Vocabulary (DPV). W3C Data Privacy Vocabularies and Controls Community Group. Recommended vocabulary for the per-consent `purpose` field.

[OPENXR]
Khronos OpenXR Specification and Extension Registry. The Khronos Group. Source for sensor-class naming and OpenXR extension identifiers in the device-capability facet.

[RFC9635]
RFC 9635: Grant Negotiation and Authorization Protocol (GNAP). IETF, 2024. Informs bounded-scope agent authorization patterns.

[TPM2]
Trusted Platform Module Library Specification, Family 2.0. Trusted Computing Group. Hardware root-of-trust quotes referenced by device attestation.

[MULTIFORMATS]
Multiformats: multibase and multihash. Protocol Labs / Multiformats community. Self-describing base and hash encodings used by `prevHash` (Section 3.3.2.1).

[BIP-32]
BIP-32: Hierarchical Deterministic Wallets; SLIP-0010: Universal Private Key Derivation. Hierarchical-deterministic key derivation referenced by the Tier 2 HD-derivation profile (Section 6.4.7.2).

[Groth16]
J. Groth. On the Size of Pairing-Based Non-Interactive Arguments. EUROCRYPT 2016. The zk-SNARK proving system used in Tier 2 Profile 2B.

[AnonCreds]
AnonCreds Specification. Hyperledger. Link-secret holder-binding model informing the BBS+ linked-secret profile (Section 6.4.7.1).

[ThresholdBBS]
J. Doerner, Y. Kondi, et al. Threshold BBS+ signatures (2023). Non-normative reference for multi-party ceremony issuance (Section 6.4.8.3).

[OMATrust]
OMA3 Trustless Web Group. OMATrust attestation framework. Source of the end-to-end claim-proof verification model (Section 6.4.9).

[VerifiableIntent]
Mastercard. Agentic / Verifiable Intent pattern. Bounded-scope agent-delegation pattern referenced in Section 6.5.3.

10. IANA Considerations

This section registers the media types used by this specification and records the change control for its identifier registries.

10.1. Media Type Registrations

This specification defines two media types. Registration with IANA per [RFC6838] is requested at specification lock.

- `application/um+ld+json` — the JSON-LD reference encoding (Section 1.7.3). The `+ld+json` structured-syntax suffixes indicate a JSON-LD document; the `um` prefix identifies the Universal Manifest profile. Encoding: UTF-8. Fragment identifiers: per JSON-LD.
- `application/um+cbor-ld` — the CBOR-LD compact encoding (Section 1.7.4, PREVIEW). Encoding: binary (CBOR).

> **Note:** The multi-suffix form `+ld+json` is used for alignment with JSON-LD-based media types; the exact suffix handling will be confirmed against the structured-syntax-suffix registry rules at registration time. Whether `application/um+cbor-ld` registers as a suffix-bearing type or a standalone type depends on the CBOR-LD profile outcome and is an open working-group item.

10.2. Registry Change Control

The `um:profile:` registry (Section 5.1), the `um:reason:` receipt-reason registry (Section 3.3.2.3), and the subsidiary value registries (receipt event classes, agent-delegation scope values, liveness `proofType` values, JWE algorithm pairs; Section 5.1.1) are maintained under the change control of the Universal Manifest specification maintainers, following the IANA-style registration procedure defined in Section 5.1.

> **Working-group item:** Example identifiers in this document use `urn:uuid:` and `https:` forms. Whether to additionally register a `um` URN namespace identifier per [RFC8141] (enabling `urn:um:` identifiers) is left to the working group; this version does not rely on an unregistered URN namespace.

Appendix A. Manifest Class Registry Snapshot (Informative)

PREVIEW

This appendix is informative. It snapshots the manifest classes (Section 1.0.1) currently surfaced by integration profiles, to illustrate the polymorphic-envelope model. It does **not** bind conformance; the authoritative, versioned registry is maintained as a standalone document under the profile

registration mechanism (Section 5.1). Discriminating members are the members whose presence identifies the class.

Manifest class	Discriminating members	Purpose
Identity capsule	`claims[]` with identity claim types; `holderBinding`	Portable identity and credential presentation.
Consent record	`consents[]` with `um:Consent` entries	Records permission grants governing facet use.
Device-capability descriptor	`devices[]` with `deviceCapability`	Advertises session-scoped device capabilities.
Device-attestation record	`devices[]` with `deviceAttestation`	Conveys manufacturer-signed hardware provenance.
Receipt	`@type` includes `um:Receipt`; `seq`/`prevHash`	Signed, chainable evaluation/audit record.
Cross-DID binding	`claims[]` with `identity.crossDidBinding`	Asserts common control of multiple DIDs.
Agent-delegation capsule	`pointers[]` with `um:agentDelegation`; `actorState`	Declares delegated session authority and the operating actor.
Encrypted-facet carrier	`facets[]` with `encryptionProfile`	Carries sealed, recipient-scoped payloads.
Category/service descriptor	`claims[]` with `operatorIdentity` / `categoryAttestation` types	Describes a service with attested category trust.
Bilateral-session record	`@type` `um:BilateralSession`; `exchangeId`	Correlates a two-party manifest exchange.

Note: The snapshot above is illustrative and non-exhaustive; integration profiles (e.g., the RP1 integration profile) surface additional classes. A manifest may belong to more than one class at once; the normative rule is stated in Section 1.0.1.1. The bilateral-session record row anticipates a manifest that *records* a session: the session object of Section 3.5.1 is a protocol-layer object, not itself a manifest, so a session-record manifest would carry the common envelope members with `@type` including both `um:Manifest` and `um:BilateralSession`; its class profile is a working-group deliverable.

Appendix B. Versioned Context Document (Normative Reference)

The term definitions for this version are fixed by the v0.4 context document served at `https://universalmanifest.net/ns/v0.4`, in which the `um:` prefix expands to `https://universalmanifest.net/ns/um#` (Section 1.2.1). For context integrity (Section 6.10), that document is **immutable** for this version: implementations SHOULD pin it by content hash rather than fetch it at evaluation time, and CBOR-LD deployments MUST tokenize against the pinned copy.

The canonical context document and its SHA-256 content hash are published alongside this specification at the versioned namespace URI and in the specification's repository. Implementations pinning the context MUST verify the retrieved document against the published hash for the version they support before relying on its term definitions.

Appendix C. Complete Worked Example (Informative)

This appendix is informative. It shows one complete Universal Manifest carrying a facet, a claim, a consent, and a pointer, signed under Signature Profile A (Section 1.6), followed by the receipt a conformant evaluator produces when it evaluates that manifest. Cryptographic values are illustrative placeholders. The pair doubles as a conformance fixture.

Example 22: Complete signed manifest


```
{
  "@context": ["https://universalmanifest.net/ns/v0.4"],
  "@id": "urn:uuid:6f5b2c40-9d1e-4a8e-b2c1-0a1b2c3d4e5f",
  "@type": ["um:Manifest"],
  "manifestVersion": "0.4",
  "subject": "did:key:z6MkAlice",
  "issuedAt": "2026-06-09T12:00:00Z",
  "expiresAt": "2026-06-09T20:00:00Z",
  "facets": [
    {
      "@id": "urn:uuid:facet-public-profile",
      "@type": "um:Facet",
      "name": "Public profile",
      "entity": {
        "@id": "urn:uuid:entity-alice-profile",
        "@type": ["um:Entity", "ProfileCard"],
        "displayName": "Alice"
      }
    }
  ],
  "claims": [
    {
      "@id": "urn:uuid:claim-over-18",
      "@type": "ageOver",
      "issuer": "did:web:issuer.example",
      "value": 18,
      "requiredTrustTier": 1,
      "claimProof": { "@type": "VerifiablePresentation", "proofValue": "eyJ.
      "holderBinding": { "mode": "sd-jwt-kb", "cnfThumbprint": "NzbLsXh8..."
    }
  ],
  "consents": [
    {
      "@id": "urn:uuid:consent-public-profile-001",
      "@type": "um:Consent",
      "facetRef": "urn:uuid:facet-public-profile",
      "scope": ["read", "display"],
      "purpose": "session-personalization",
      "grantedAt": "2026-06-09T12:00:00Z",
      "expiresAt": "2026-06-09T20:00:00Z"
```

```
    }
  ],
  "pointers": [
    {
      "@id": "urn:uuid:pointer-avatar-1",
      "@type": "mediaRef",
      "target": "https://cdn.example/alice/avatar.glb",
      "createdAt": "2026-06-09T12:00:00Z"
    }
  ],
  "presentationProof": {
    "proofType": "sd-jwt-kb",
    "challenge": "nonce-venue-8f2a",
    "audience": "did:key:z6MkVenueEdge",
    "created": "2026-06-09T12:00:00Z",
    "proofValue": "base64url-kb-jwt-proof"
  },
  "signature": {
    "algorithm": "Ed25519",
    "canonicalization": "JCS-RFC8785",
    "keyRef": "did:key:z6MkAlice#keys-1",
    "publicKeySpkiB64": "MCowBQYDK2VwAyEA...",
    "created": "2026-06-09T12:00:00Z",
    "value": "base64url-encoded-signature-bytes"
  }
}
```

Example 23: Receipt produced by a conformant evaluation of Example 22


```
{
  "@context": ["https://universalmanifest.net/ns/v0.4"],
  "@id": "urn:uuid:receipt-7a2e",
  "@type": ["um:Manifest", "um:Receipt"],
  "manifestVersion": "0.4",
  "subject": "did:key:z6MkVenueEdge",
  "issuedAt": "2026-06-09T12:00:01Z",
  "expiresAt": "2026-06-10T12:00:01Z",
  "manifestId": "urn:uuid:6f5b2c40-9d1e-4a8e-b2c1-0a1b2c3d4e5f",
  "outcome": "accepted",
  "signatureCheck": "valid",
  "freshnessCheck": "fresh",
  "keyRefResolution": "resolved",
  "revocationStatus": "unchecked",
  "effectiveTrustTier": 1,
  "holderBindingStatus": "verified",
  "presentationProofStatus": "verified",
  "facetStatuses": [
    { "facetId": "urn:uuid:facet-public-profile", "status": "processed", "na
  ],
  "consentStatuses": [
    { "facetId": "urn:uuid:facet-public-profile", "consentRef": "urn:uuid:co
  ],
  "claimStatuses": [
    { "claimRef": "urn:uuid:claim-over-18", "status": "verified", "tier": 1
  ],
  "processedAt": "2026-06-09T12:00:01Z",
  "signature": {
    "algorithm": "Ed25519",
    "canonicalization": "JCS-RFC8785",
    "keyRef": "did:key:z6MkVenueEdge#keys-1",
    "created": "2026-06-09T12:00:01Z",
    "value": "base64url-encoded-signature-bytes"
  }
}
```


The receipt records a Tier 1 outcome: the claim carried a verified `holderBinding` (Section 6.6.1), the manifest signature verified against a resolved `keyRef`, the single facet was processed under a

valid consent whose `scope` and `purpose` covered the evaluator's use, and the verifier-issued challenge was answered by a valid `presentationProof` (so `presentationProofStatus` is `"verified"`). The manifest carries no `signature.statusRef`, so revocation status is recorded as `"unchecked"`.