Universal Manifest v0.5 — Base Specification

Tier-0 / Tier-1 Core

v0.5.0. Published as tag v0.5.0 of UniversalManifest/universal-manifest, with the same wire files as release candidate 1 (v0.5.0-rc.1). It is a mechanism release, not a completed implementation or conformance qualification. Immutable versions in UniversalManifest/universal-manifest are normative; main is the working version. Earlier versions were reviewed in the OMA3 Technical Working Group, and publication asserts no OMA3 ratification.

8 November 2026

This document: /spec/v0.5/core/

Document set: Universal Manifest v0.5.0 · What changed in v0.5.0

Artifacts: JSON Schema · JSON-LD context

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

Author: Grig Bilham — inventor, architect, and author of the Universal Manifest.

Copyright © 2026 SumSet Tech LLC and the Universal Manifest contributors. This specification is available under the Apache License, Version 2.0. It is free and remains free. Earlier published revisions retain their original licence notices.

---

v0.5.0. Published as tag `v0.5.0` of `UniversalManifest/universal-manifest`, with the same wire files as release candidate 1 (`v0.5.0-rc.1`). Wire version `0.5`; namespace `https://universalmanifest.net/ns/v0.5`; reading routes `/spec/v0.5/`. It is a mechanism release, not a completed implementation or conformance qualification. Feature-specific PREVIEW and conditional requirements remain in force.

This is the **Base** of the Universal Manifest v0.5 document set: the complete Tier-0 / Tier-1 evaluator and holder requirements. The source defines the baseline; implementations and the new-version conformance corpus still require qualification before an interoperability claim. Higher-tier and optional requirements remain specified by the companion profiles. New readers should start with the Primer; worked examples live in the Cookbook.

Universal Manifest v0.5 Specification — Base

This version: https://universalmanifest.net/spec/v0.5/core/ Previous version: https://universalmanifest.net/spec/v0.4/ History: https://universalmanifest.net/spec/ v0.5.0, published as tag `v0.5.0`. The immutable revision of https://github.com/UniversalManifest/universal-manifest identified by a release is authoritative; main is the working source and the website is its generated reading view. Copyright (c) 2026 SumSet Tech LLC and the Universal Manifest contributors. Grig Bilham — inventor, architect, and author of the Universal Manifest. This specification is available under the Apache License, Version 2.0. It is free and remains free. Earlier published revisions retain their original licence notices. Earlier versions were reviewed in the OMA3 Technical Working Group, and publication asserts no OMA3 ratification.

> This document carries all normative content from v0.3 forward and adds Tier-0/Tier-1 baseline content for v0.5. Sections marked **PREVIEW** cover features still under development and may change. For the current stable specification, see Universal Manifest v0.3.

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 conforming implementations are expected to interoperate with; a second compact encoding (CBOR-LD) is defined in EXT-OPT to demonstrate format independence. Formulated as a hybrid of web publication metadata and web application parameters, the Universal Manifest gives developers and holders a standardized envelope to convey linked-data identity references, role permissions, device registrations, and pointers to canonical data sources.

The Universal Manifest is designed for local-first environments (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 a continuous cloud connection, enabling 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

Universal Manifest v0.5.0 carries forward the accepted v0.4 editorial contract and earlier v0.3 baseline, with a distinct wire version, immutable namespace and stricter version-bound Profile A signature acceptance. It is published as tag `v0.5.0`; no complete implementation qualification or institutional ratification is asserted.

This **Base** document carries the Tier-0/Tier-1 conformance surface. Features that a conformance claim *adds* above Tier 1, and capabilities orthogonal to the tier ladder, live in companion extensions (EXT-T1, EXT-T2, EXT-T3, EXT-OPT); the Base names each and points to it. Requirements in PREVIEW sections become final only after maintainer review.

Because Universal Manifest is still in the v0.x line, minor-version bumps may include breaking changes reflected by a new version folder. For implementers, the v0.3 specification remains the stable baseline; run the Standalone Conformance Suite and review Conformance v0.3.

Changes for v0.5 (summary)

Relative to published v0.4, this candidate removes five context definitions, materializes Profile B schema support, and tightens Profile A acceptance under a new wire identity; see migration and release-candidate notes. The inherited additions below describe the earlier v0.3-to-v0.4 editorial work retained in this version, not proof that every feature is implemented or qualified. Historical v0.3 compatibility remains version-routed under EXT-T1; the new signature predicate does not reinterpret old-version signatures.

- **Credential binding (new normative, Tier 1+):** `holderBinding`, `presentationProof`, `livenessAttestation`, the end-to-end claim-proof process, and credential-binding receipt fields — specified in EXT-T1. The Base carries the *requirement* and the field surface.
- **JWE algorithm constraints (new normative):** a MUST-implement algorithm pair for encrypted facets — §2.4.

- **Stage-2 credential-binding verification sub-steps (new normative):** woven into the Verify stage — summarized in §3.1.2, specified in EXT-T1.
- **statusRef resolution schema, cryptographic-requirements summary** — EXT-OPT.
- **Envelope/wire additions (PREVIEW):** manifest classes and the polymorphic envelope (§1.0.1); `actorState`, two-component `devices`, derived-variant sensor consent, receipt-as-class, `trustWeight`/category-trust — all specified in EXT-OPT, named here.
- **Format independence:** abstract data model and production rules, plus ADM conformance — named in §1.7 and §4.5, specified in EXT-OPT.
- **Advanced trust and privacy profiles (PREVIEW):** Tier-2 ZKP (EXT-T2), Tier-3 ceremony (EXT-T3), post-quantum signatures and selective-disclosure dual-track (EXT-T2).
- **Infrastructure (PREVIEW):** bilateral session model, agent-delegation scope registry, profile registration, federation — EXT-OPT.
- **Design-stage candidate profiles (PREVIEW, not yet on the wire):** current policy-state resolution, asset-bound represented-person consent, signed referral-route attribution, and GeoPose primary pose values — drafted in EXT-OPT; their candidate members are absent from the published schema/context and fixtures until finalized.

For the v0.2→v0.3 migration path, see the v0.3 specification.

✧ ✧ ✧

1. Universal Manifest

A **Universal Manifest** is a cross-platform data envelope defined by an abstract data model (§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-LD11 reference encoding; the same manifest MAY be produced in any other conformant encoding, such as CBOR-LD (EXT-OPT), 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 (§1.3.2).
- **holder** — The party that assembles, signs, and presents a manifest. The holder controls selective disclosure (§3.1.3) and is the party bound by the `holderBinding` mechanisms (EXT-T1).
- **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 (§3.1). Also called the *verifier*; it is the relying party applying the result. This specification uses "evaluator" uniformly.
- **issuer** — The party that issued a credential carried as a claim or `claimProof` (§6.4.3).
- **attester** — A party that asserts a cross-DID binding or other attestation an evaluator may trust by policy (§6.4.4).
- **resolver** — A service that resolves manifest status or coordinates federated status (EXT-OPT).
- **status endpoint** — The HTTP endpoint a `signature.statusRef` resolves to (EXT-OPT).
- **session** — A time-bounded interaction; for bilateral exchanges, the object specified in EXT-OPT.
- **receipt** — The structured, machine-readable record an evaluator produces (§3.3).
- **production / consumption** — Serializing the abstract manifest into a wire encoding / parsing it back (§1.7).
- **sealed entry** — An encrypted facet an evaluator cannot decrypt; recorded as present but not read (§3.1.4).
- **trust tier** — One of the four claim-verification tiers (§6.4.2). The *achieved* tier (`effectiveTrustTier`) is distinct from a declared `requiredTrustTier` floor and from the *interaction tier floor* negotiated in a bilateral exchange (§6.4.6).
- **interactive presentation** — A presentation made in response to a verifier-issued `challenge` (EXT-T1).

1.0.1. Manifest Classes and the Polymorphic Envelope PREVIEW

The Universal Manifest is a **polymorphic envelope**: a single fixed envelope shape carries many kinds of payload. The envelope members (§1.2), identity and lifespan members (§1.3), structural state members (§1.4), and integrity proof (§1.5) are common to every manifest. What a particular

manifest *is* — identity capsule, device-capability descriptor, consent record, receipt — 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 v0.3 already enables; v0.5 names it and states the rule explicitly.

A **manifest class** is a named, profile-defined combination of structural members 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, and any additional top-level `@type` values a class profile declares alongside `um:Manifest` (for example `um:Receipt`). Evaluators MUST NOT rely on the 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.

Because classes are discriminated by member presence rather than 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 (§3.1.1). This class/member extensibility is compatible at the abstract-model level; older envelopes retain their declared version and context and require an explicitly supported old-version processing path, while conversion to v0.5 follows the migration rules, including a new signature over the changed signing input. New manifest classes are defined by **profile documents** registered through the profile registration mechanism (EXT-OPT) under the `domain` category, and MUST NOT redefine or remove any common envelope member. An informative snapshot of classes currently surfaced by integration profiles is in EXT-OPT (Manifest Class Registry Snapshot).

Preview: Naming the polymorphic envelope and the discriminator rule is built on the editors' default (add; non-breaking). Input is requested on whether the discriminator should remain member-presence-based or also admit an explicit optional `manifestClass` hint member.

1.1. Examples

A minimal Universal Manifest carries only the anchor, identity, and lifespan members plus a signature:

```
{
  "@context": ["https://universalmanifest.net/ns/v0.5"],
  "@id": "urn:uuid:123e4567-e89b-12d3-a456-426614174000",
  "@type": ["um:Manifest"],
  "manifestVersion": "0.5",
```

```
  "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"
  }
}
```

The `signature` values above are illustrative; see §1.6 for the full signature shape and signing procedure. Worked examples for every concept in this document are collected in the Cookbook.

1.1.1. Universal Manifest object at a glance

This table is a reader aid for the top-level Universal Manifest object. It restates the member surface already defined in the cited sections and extension profiles; those detailed sections remain the source of the requirements.

Surface	JSON-LD reference-encoding member(s)	Status in v0.5	Detailed definition
Semantic anchors	`@context`, `@id`, `@type`	REQUIRED. `@type` MUST include `um:Manifest`.	§1.2; context integrity in §6.10 and Appendix B.
Version, subject, and lifespan	`manifestVersion`, `subject`, `issuedAt`, `expiresAt`	REQUIRED.	§1.3.
Structural payload arrays	`facets`, `claims`, `consents`, `pointers`, `devices`	OPTIONAL top-level members. When absent, evaluators treat them as empty arrays.	§1.4; facet/entity mechanics in §2; trust-tier and agent-delegation details in §6.4 and §6.5; device details in EXT-OPT.
Manifest-level trust floor	`requiredTrustTier`	OPTIONAL; default 0. Claim-level and facet-level	§1.4.2 and §6.4.5.

Surface	JSON-LD reference-encoding member(s)	Status in v0.5	Detailed definition
		floors can only raise this floor.	
Integrity proof	`signature`	REQUIRED for conformant manifests.	§1.5 and §1.6.
Interactive and binding proof surface	`presentationProof`, `livenessAttestation`	OPTIONAL members generally; REQUIRED only when the interaction or claimed tier requires the corresponding proof.	Holder obligations in §4.2; mechanics in EXT-T1.
Session actor state	`actorState`	OPTIONAL, PREVIEW.	§1.4.7 and EXT-OPT §O9.3.
Supplemental signature profiles	`postQuantumSignature`	OPTIONAL, PREVIEW; safely ignored unless the evaluator supports the profile.	Signature-profile handling in §1.6; post-quantum profile in EXT-T2.

For the serialization-independent abstract property table behind these JSON-LD member names, see EXT-OPT §O1.1. Supporting the JSON-LD reference encoding alone remains sufficient for full v0.5 conformance (§4.5).

1.2. Core Identity Members

Three properties — the manifest's **contextReference**, **id**, and **type** — identify and semantically anchor every manifest. They are defined here in the JSON-LD reference encoding, where they are expressed using the JSON-LD keywords `@context`, `@id`, and `@type`. The requirements apply to the corresponding abstract properties in any conformant encoding.

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/v0.5`).

The namespace URI is versioned. Manifests conforming to this specification MUST use `https://universalmanifest.net/ns/v0.5`. Evaluators that support an earlier version MAY

process manifests declaring it; an evaluator processing a v0.3 manifest under this specification applies the v0.5 evaluation sequence with the backwards-compatibility rules of EXT-T1.

The `um:` prefix used throughout (in `@type` values such as `um:Manifest` and 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, identified by the versioned namespace URI (and served there); this version's context document, with its content hash, is reproduced in Appendix B and is normative for context integrity (§6.10). The same normative document is `spec/v0.5/schema.jsonld` in the specification repository.

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 v0.5 conformant manifests, the value MUST be `"0.5"`. Evaluators MUST check that they support the declared version before processing — meaning a defined processing path exists, either native support or, for v0.3, the compatibility path of EXT-T1.

1.3.2. `subject` member

The `subject` member is REQUIRED. It specifies the primary entity (user, app, venue) the manifest describes, and 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 [RFC 3339] `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 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.

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

These arrays group operational contexts representing permissions, deployed hardware targets, and external data reference pointers connected to the `subject`. Each array is a top-level structural member. 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 top-level member `requiredTrustTier` — an integer (0–3) specifying the minimum trust tier an evaluator MUST satisfy before acting on this manifest. OPTIONAL; default 0. See §6.4.5.

1.4.3. `claims` Array Schema

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

A base claim object MUST contain:

- `@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:

- `@id` — An opaque identifier for this claim entry. RECOMMENDED when receipts or warnings reference the claim.
- `subject` — DID or URI of the entity the claim is about. When absent, the manifest-level `subject` is implied.
- `issuedAt` / `expiresAt` — RFC 3339 date-times. Evaluators SHOULD reject expired claims.
- `claimProof` — A Verifiable Presentation, attestation proof object, URI reference, or array of proof entries enabling Tier 1 verification (§6.4.3). When an array, each entry is independently verifiable.
- `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"`. The binding verification procedure and mode-specific fields are specified in EXT-T1.
- `requiredTrustTier` — Integer (0–3) declaring the minimum trust tier for this claim (§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 above Tier 0). The `identity.crossDidBinding` claim type is defined in §6.4.4.

(Worked example: base claim object — see Cookbook.)

1.4.4. `consents` Array Schema

The `consents` array contains zero or more consent entry objects, each recording a permission grant governing how an evaluator may act on specific facets. The Consent stage (§3.1.4) uses these entries to determine whether processing a facet is authorized.

A consent entry MUST contain:

- `@id` — A URI uniquely identifying this consent entry. MUST be present (used for receipt recording).
- `@type` — MUST include `"um:Consent"`.
- `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.
- `scope` — An array of scope strings defining authorized operations (e.g., `"read"`, `"display"`, `"cache"`, `"process"`, `"forward"`). These are deployment-defined; this specification does not enumerate a closed set. Evaluators MUST NOT perform any operation not literally present in `scope` (fail closed), except where an optional profile defines a strictly-narrowing implication rule that the deployment documents in its conformance claim (e.g. the sensor derivation-tier rule, EXT-OPT §O9.1).
- `purpose` — A string declaring the stated purpose for data use (e.g., `"session-personalization"`, `"age-verification"`). Evaluators MUST verify that their intended use falls within the declared purpose. Purpose comparison is exact string equality unless both parties support a shared vocabulary (e.g., 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. A request that omits its purpose MUST NOT match a purpose-bearing consent, for disclosure or mutation. A consent whose required `purpose` is missing or invalid is malformed, not an unrestricted grant.
- `grantedAt` — RFC 3339 date-time when consent was granted.
- `expiresAt` — RFC 3339 date-time when consent expires. Evaluators MUST treat an expired entry as expired and not satisfied (§3.1.4).

A consent entry MAY contain:

- `grantor` — DID or URI of the granting entity. When absent, the manifest `subject` is implied.
- `withdrawnAt` — RFC 3339 date-time of withdrawal. When present, whatever its value, the consent is no longer active regardless of `expiresAt`; evaluators MUST treat the entry as withdrawn and not satisfied (§3.1.4).
- `conditions` — An array of condition strings imposing additional constraints (e.g., `"offline-only"`, `"no-third-party-sharing"`). Deployment-defined. Evaluators MUST NOT process a facet governed by a condition they do not recognize or cannot enforce (fail closed).

When a facet has no matching consent entry, the evaluator MUST record the facet status as `"consent-missing"` and MUST NOT process the facet's data. When the `consents` array is empty or absent, evaluators MUST treat all facets as lacking consent, unless the deployment operates

under a consent model external to the manifest (e.g., a pre-negotiated bilateral agreement); such external models are outside this specification's scope.

The derived-variant **sensor-consent vocabulary** (per-sensor consent keys with purpose binding, for spatial-computing sensors) extends this base schema and is specified in EXT-OPT.

(Worked example: consent entry — see Cookbook.)

1.4.5. `pointers` Array Schema

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

A base pointer object MUST contain `@type` (a string identifying the pointer type; MUST be present) and `target` (a URI referencing the external resource; MUST be present). Pointer types MAY define type-specific fields that replace the base `target` requirement. A base pointer object MAY contain `@id`, `label`, `createdAt`, and `expiresAt` (RFC 3339; evaluators SHOULD ignore expired pointers for processing purposes — with one exception: an expired `um:agentDelegation` pointer is not ignored but rejects the manifest, §6.5.1).

The `um:agentDelegation` pointer type, which declares delegated session authority, is defined in §6.5 and replaces the base `target` requirement with the fields of §6.5.1. Evaluators encountering an unrecognized pointer `@type` MUST record the pointer as present in the receipt but MUST NOT act on it.

1.4.6. `devices` Array Schema PREVIEW

The `devices` array registers hardware endpoints (XR headsets, NFC readers, smart displays, wearable sensors) associated with the `subject`. v0.3 reserved this member without defining its entries. **v0.5 defines device entries** as a two-component split — a long-lived, manufacturer-signed `deviceAttestation` component and a session-scoped, user-signed `deviceCapability` component — specified in full in EXT-OPT.

In the Base, the `devices` array is a **named reserved structural member**: it is part of the signed payload, so evaluators MUST include it when recomputing the signing input (§1.6.3) and MUST NOT discard it during Arrive-stage unknown-field handling. Evaluators that do not implement the device components MUST still preserve the array and record device entries as present-but-unprocessed.

1.4.7. `actorState` member PREVIEW

The `actorState` member is an OPTIONAL top-level object declaring who is operating the session (the human principal, a delegated agent, or a hybrid), bridging the `um:agentDelegation` pointer (§6.5) to session-state semantics. It is named here as a reserved optional member and specified in full in EXT-OPT. When present, `actorState.principal` MUST match the manifest `subject`; `actorState` is additive and non-breaking, and evaluators that do not implement it record it as present-but-unprocessed.

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 v0.5 conformant manifest MUST include a `signature` member conforming to a signature profile this version recognizes (§1.6): **Signature Profile A** (JCS + Ed25519), the default and the mandatory-to-implement baseline for every evaluator, or **Signature Profile B** (JCS + ES256), the additive hardware-custody suite specified in EXT-OPT. The recognized set is governed and open, and it grows by registration rather than by re-architecture (§1.6). Evaluators MUST reject manifests whose signature profile they do not support.

Unsigned manifests MAY exist as development artifacts but are not v0.5 conformant and MUST be rejected by conformant evaluators.

1.6. Signature Profile A: JCS + Ed25519

Signature Profile A is the baseline normative signature profile. It constrains the `signature` member to a deterministic, portable format suitable for local-first verification on constrained devices. Introduced in v0.2, it remains the required baseline in v0.3, v0.4 and this v0.5 candidate; the new version separately defines the acceptance boundary below.

Signature profiles are additive, and the recognized set is governed and open. A profile's identity is the explicit pair `signature.algorithm` + `signature.canonicalization` (§1.6.5). The set of pairs this version recognizes is a *governed registry*, not a closed list of the algorithms the specification was willing to imagine: the `algorithm` enumeration in the published `schema.json` is that registry's current materialization on the wire, and any number of further suites MAY be recognized by exactly the governance step that recognized the ones here — a proposal carrying

normative text, maintainer review, conformance fixtures, and registration under the `signature` category of the profile registration mechanism (EXT-OPT §O6). Recognizing a suite is the governance act; the enumeration edit is its bookkeeping. Nothing in this design caps the number of signature suites Universal Manifest can carry.

v0.5 recognizes two. **Signature Profile A** (JCS + Ed25519), defined in this section, is the default and the mandatory-to-implement baseline: every evaluator MUST implement it, and an evaluator that implements only Profile A is fully conformant. **Signature Profile B** (JCS + ES256, ECDSA over NIST P-256), specified in EXT-OPT §O15, is additive and OPTIONAL for evaluators; it exists so that a signing key held inside device security hardware — which, on the platforms that ship such hardware, means a P-256 key — can sign a manifest directly. Their registry identifiers (EXT-OPT §O6) are `um:profile:signature:jcs-ed25519` and `um:profile:signature:jcs-es256`.

Additive means additive: a later profile never replaces Profile A, silently or otherwise, and the arrival of a further suite does not itself change Profile A. This v0.5 candidate separately selects the version-bound acceptance rule in §1.6.1.1; its signature shape and signing-input procedure remain unchanged. Future versions MAY recognize more suites (post-quantum algorithms — see EXT-T2 — or W3C Data Integrity proofs) by that same step. Evaluators verify the profiles they support; supplementary proof material from an unknown profile carried alongside a supported signature (for example, a `postQuantumSignature` during dual-signature migration) is safely ignored. A manifest whose only signature uses a profile the evaluator does not support MUST be rejected (§1.6.5) and MUST NOT be reinterpreted under a profile the evaluator does support.

1.6.1. Canonicalization and Algorithm

Signature Profile A uses **JSON Canonicalization Scheme** (JCS, RFC 8785) for canonicalization and **Ed25519** [RFC 8032] for signing. Signature values are encoded as **base64url without padding** [RFC 4648] (the JOSE convention; verifiers MUST NOT require, and SHOULD reject, padded values). This combination provides deterministic signing input without JSON-LD expansion or RDF canonicalization. The profile is defined against the JSON-LD reference encoding and signs the canonical bytes of that production; other production rules either reuse a profile defined against the abstract model or specify their own byte-level signing rule (§1.7).

1.6.1.1. Ed25519 acceptance boundary

This rule is normative for the wire `0.5` candidate. Profile A requires canonical **non-identity prime-order** public-key and signature points, canonical scalar encoding and the full-challenge uncofactored equation. Rejecting small-order points alone is insufficient: mixed-order A and R are

also rejected. This does not establish that a valid supplied key is an authorized Manifest author. Old versions retain their own verification rules.

1. The resolved Ed25519 public key A MUST be exactly 32 octets. Decode the signature to exactly 64 octets and split it into the received 32-octet R encoding and 32-octet little-endian S encoding. Reject any other length. This does not change the existing SPKI carriage or key-resolution contract.
2. A and R MUST decode according to RFC 8032 §5.1.3: clear the x-sign bit to recover y and require y < p, require a valid curve point, and reject x = 0 with the sign bit set. Decoding followed by canonical re-encoding MUST reproduce the received octets exactly. Here p = 2^255 − 19. Noncanonical encodings MUST NOT be normalized and then accepted.
3. Interpret S as a little-endian integer and require 0 ≤ S < L, where L = 2^252 + 27742317777372353535851937790883648493. Do not reduce an out-of-range S and accept it.
4. Both A and R MUST belong to the subgroup of order L and MUST differ from its identity O: require [L]A = O and [L]R = O, and reject A = O or R = O. In particular, small-order points and mixed-order points MUST be rejected. A test of [8]P = O distinguishes small-order points; it does not establish that P has no torsion component.
5. Let M be the exact signing-input octets produced by §1.6.3. For this plain Ed25519 profile, compute SHA-512(Renc ‖ Aenc ‖ M) over the exact received canonical R and A encodings and M; interpret the full 64-octet digest as a little-endian integer k. Do not hash re-encoded, substituted, or parsed-object representations. This follows RFC 8032 §5.1.7; the Ed25519ctx/Ed25519ph domain/prehash variants are not selected here.
6. Accept the signature only if [S]B = R + [k]A, using the Ed25519 base point B. This is the uncofactored equation permitted by RFC 8032 §5.1.7. After the subgroup checks in step 4, replacing k with k mod L in multiplication by A is equivalent; without that check, equivalence MUST NOT be assumed. Do not substitute a cofactored equation or a library function name as the acceptance definition. RFC 8032 §8.8 explains why the exact equation must be consistent across implementations.

The TypeScript v0.5 implementation explicitly enforces these predicates, including did-auth proof signatures. Its finite adversarial and signed-fixture qualification, plus detached same-input predicate agreement with Rust, are described in CONFORMANCE. These bounded results do not establish complete evaluator conformance or full runtime parity; a library function name is not a substitute for these predicates. Profile B raw `R || S` encoding and its high-S acceptance are unchanged.

1.6.2. Signature Shape

The `signature` object MUST contain, 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 (unpadded, §1.6.1) 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 when the signature was produced.
- `signature.statusRef` — URI to status material for this manifest instance (the manifest identified by its `@id`; see the statusRef resolution schema in EXT-OPT). It conveys the status of the manifest, not of any key.
- `signature.revocationCursor` — status cursor/version token for cache-aware revocation checks; ordering exists only when the selected status profile defines it for the relevant authority and target (EXT-OPT O2.1).

Additional fields MAY exist for future profiles, but evaluators SHOULD rely on `algorithm` + `canonicalization` to decide whether they can verify a given signature. The entire `signature` property is **not included** in the signing input (to avoid circularity). Consequently, its `statusRef` and `revocationCursor` members are **not authenticated by the envelope signature**. Their presence cannot establish the holder's authorization of an endpoint or a trusted cursor floor; the authority, target, and any relied-upon ordering/freshness binding must be established independently under EXT-OPT O2.1.

The `signature.created` and `signature.keyRef` members are likewise **not authenticated by the envelope signature**, and a change to either does not invalidate it. An evaluator MUST NOT use `signature.created` as evidence of signing time, freshness, or ordering; a relying party that needs a signed time uses a signed member such as `issuedAt`, or a `presentationProof` whose binding covers its own `created`. `signature.keyRef` only names where key material may be found: it establishes nothing about the signer until the checks of §1.6.4 and any separately required authority method succeed. These statements concern the signature only. Both members keep their syntax rules: `signature.created` MUST be an RFC 3339 `date-time` as the schema defines it,

and a value that does not conform fails validation of the document, whether or not the signature verifies.

(Worked example: Signature Profile A object — see Cookbook.)

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).
2. Remove the `signature` property entirely. Also remove `presentationProof` if present (a verification-time proof computed over the signing-input hash; see EXT-T1) and `postQuantumSignature` if present (see EXT-T2).
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.

JCS presumes I-JSON-compatible input: the JSON-LD production consumed for signing MUST be an I-JSON document [RFC 7493], and a document with duplicate member names MUST be rejected at the Arrive stage (§3.1.1) — parser behavior on duplicates is otherwise implementation-defined, which would make the signing input ambiguous between two conformant verifiers. The whole manifest as received MUST also be an I-JSON document, including members outside the signing input such as `signature`. An evaluator MUST reject a manifest that is not I-JSON at the Arrive stage as a structural failure, before the resource limits (§6.3) and the signature check. Examples are a string with a lone surrogate and a number outside the IEEE 754 double range.

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, in order: (1) confirm the document is a v0.x Universal Manifest (required fields present, `@type` includes `um:Manifest`); (2) enforce TTL (§3.1.2); (3) confirm profile support via the `algorithm`+`canonicalization` pair (§1.6.5); (4) **resolve the verification key** (see below); (5) recompute the signing input (§1.6.3); (6) verify the Ed25519 signature over the canonical bytes. If verification fails, the manifest MUST be rejected for use (MAY be retained for debugging).

For a wire `0.5` manifest, checklist step 6 MUST apply all six requirements of §1.6.1.1 to the exact signing input. A separately supported old-version path retains that version's acceptance rule.

Key resolution (step 4). A resolving evaluator MUST attempt keyRef resolution through the authorized, bounded resolver configured for that identifier. A deployment that deliberately operates without resolution MAY verify a supported signature with embedded key material, but MUST record `keyRefResolution: "not-attempted"` and declare that capability boundary. A failed or unavailable attempted resolution is `unresolved`; a successful authorized resolution is `resolved`. No network access is implied merely by receiving an arbitrary keyRef URI. The ladder is:

- Resolution succeeds **and** `publicKeySpkiB64` present → the resolved key MUST be byte-identical to the decoded `publicKeySpkiB64`; on mismatch, reject with `rejected` / `signatureCheck: "invalid"`. Because DID resolution yields verification keys in method-specific representations (JWK, multibase/multicodec), the evaluator first converts the resolved key to its SPKI DER encoding — equivalently, compares the raw 32-byte Ed25519 public-key values — before the byte comparison.
- Resolution succeeds **and** `publicKeySpkiB64` absent → use the resolved key.
- Resolution deliberately not attempted **and** `publicKeySpkiB64` present → the evaluator MAY verify embedded-key mathematics but MUST record `keyRefResolution: "not-attempted"`.
- Resolution attempted but failed/unavailable **and** `publicKeySpkiB64` present → the evaluator MAY verify embedded-key mathematics but MUST record `keyRefResolution: "unresolved"`.
- In either case, the evaluator MUST NOT attribute the keyRef's named identity or grant trust above Tier 0 on the strength of that label. Separately required sender, issuer, subject or delegate authority still requires its selected method. Where embedded and resolved keys are both available, the matching-key check remains mandatory. Unresolved bindings SHOULD be revalidated when the configured resolution capability becomes available.
- Resolution unavailable **and** `publicKeySpkiB64` absent → the manifest cannot be verified and MUST be rejected for use (MAY be retained for retry).

Security Note: Without the byte-identity check, a malicious holder could bundle a `keyRef` pointing to a high-reputation DID while supplying their own key material in `publicKeySpkiB64` (key substitution). The offline path caps identity assurance rather than blocking verification: integrity is still confirmed against the embedded key, but the `keyRef` identity MUST NOT be attributed to that key until resolution confirms the binding.

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"`. Evaluators MUST NOT reinterpret unknown pairs as the baseline profile.

1.6.6. Revocation-Aware Verification Extension

For evaluators claiming revocation-aware verification: if `signature.statusRef` is present, follow EXT-OPT O2, including independent trusted status-authority/target binding before relying on a response. Use `signature.revocationCursor` as a minimum only when its authority/target binding and the selected profile's ordering rule are established; an undefined or untrusted minimum is unsatisfied, not ignored. If revocation status cannot be established and policy requires active status, the manifest MUST be rejected for use. Otherwise an unavailable or unvalidated endpoint yields `unchecked`, never `active`, without by itself requiring envelope rejection. In either case a governed action requiring current permission MUST be denied until an authorized current-state path establishes it. 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

The manifest is defined by a **serialization-independent abstract data model** plus **production rules** that write that model into a concrete wire format, following the W3C DID Core pattern. The abstract data model is a set of properties and semantics; a *production* serializes it, a *consumption* parses it back. **JSON-LD is the reference encoding** used throughout this document and is the encoding implementations are expected to support for interoperation; a second compact encoding, **CBOR-LD**, is defined to demonstrate format independence.

This Base treats format independence at naming altitude: the manifest has an abstract model, JSON-LD is its reference encoding, and a signature is computed over the bytes of one production (so re-encoding requires re-signing). The full abstract-property table, the JSON-LD and CBOR-LD production rules, the signing-and-production rule, and context-integrity production details are specified in EXT-OPT (Abstract Data Model & Format Independence). Supporting the JSON-LD reference encoding alone is sufficient for full conformance (§4.5).

✧ ✧ ✧

2. Entities and Facets

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

2.1. um:Facet Module

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

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

A facet object SHOULD contain `entity` — a `um:Entity` object (§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 `name` (a human-readable display label), `ref` (a URI routing to the facet's authoritative source), and `requiredTrustTier` (integer 0–3 overriding the manifest-level trust tier for this facet; can only raise the floor; see §6.4.5). A facet MAY additionally carry the OPTIONAL `requiredLiveness` floor (a compound liveness-freshness floor — §6.6, EXT-T1 §T1.3.1) and the OPTIONAL `requiredAssuranceClass` floor (a closed authenticator-assurance enum `software` < `hardware-uv` < `hardware-bound`, default `software`, can only raise — §6.6, EXT-T1 §T1.3.3) PREVIEW.

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 a bare string when that is the sole type, or an array when multiple type values apply. Where a definition says a `@type` "MUST be" a single value, read it as "MUST include" under this convention.

Internationalization: Human-readable strings (`name`, `label`, `displayName`, `reason`, `rotationReason`) SHOULD be language-tagged using JSON-LD language maps where multilingual display is expected. Evaluators MUST NOT use display fields for matching.

(Worked example: plaintext facet — see Cookbook.)

2.2. um:Entity Base

The `um:Entity` is the base classification for all embedded configurations, representations, or localized states. An entity object MUST contain `@id` (a URI uniquely identifying the entity) and `@type` (an array of type strings including at least one type value). All other fields are profile-extensible; evaluators MUST ignore unknown entity fields for processing purposes but MUST NOT strip them before signature verification.

(Worked example: plaintext entity — see Cookbook.)

2.3. Encrypted Facets (JWE Inline Profile)

A facet MAY carry an encrypted entity payload using the JWE inline encryption profile, enabling the holder to include sensitive data 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

An encrypted facet declares its encrypted payload in place of (or alongside) a plaintext `entity`. The facet retains its plaintext `@id` and `@type` (`um:Facet`) so consent matching and receipt recording operate normally; the sensitive content is confined to the encrypted payload. These labels are deliberately not hidden. A party that sees a sealed facet learns its `@id`, its `@type` and any other plaintext member, such as `name`, and can match them across presentations. v0.5 gives sealed-facet labels no unlinkability.

An encrypted facet MUST carry the facet member `encryptionProfile` with the value `"jwe-inline-v1"` (this profile's identifier). That member is the declaration that the facet's `entity` value is an encrypted payload rather than a plaintext `um:Entity`: the facet's `entity` value MUST be the JWE object of §2.3.2. An evaluator recognizes an encrypted facet by the presence of `encryptionProfile: "jwe-inline-v1"` and reads its payload as a JWE accordingly.

2.3.2. JWE Structure

The encrypted payload MUST be a JWE in General JSON Serialization [RFC 7516], carrying the standard JWE members (`protected`, `recipients` with per-recipient `encrypted_key`, `iv`,

`ciphertext`, `tag`). The plaintext of the JWE is the JSON serialization of the facet's `um:Entity` payload. Per-recipient encryption allows a single encrypted facet to be readable by multiple designated evaluators, each decrypting with its own key. The baseline algorithm pair is defined in §2.4.

For the v0.5 inline profile, each recipient MUST carry its own `epk` and `kid` in `recipients[].header`; a shared `epk` is not this profile's recipient representation. Each recipient's ECDH ephemeral key pair MUST be freshly generated for that encryption and MUST NOT be reused across recipients or independently isolated facets. This is a UM profile restriction, not a claim that every shared-header JWE is insecure. Distinct serialized points alone do not prove fresh generation.

Before key use, a consumer MUST validate the original JWE and decoded protected-header JSON, including duplicate members and duplicate or conflicting JOSE header parameters across header locations. It MUST reject malformed inputs before any lossy projection can hide them. Validate `epk` as a public key under the declared algorithm, including key type, curve and point validity; private key material is not a valid `epk`. The selected `kid` MUST identify a recipient key independently authorized for the intended decryption/key-agreement purpose. An identifier or successful unwrap alone does not establish that authorization; no DID or automatic key-discovery method is required. Invalid or tampered recipient/header material MUST NOT yield plaintext through a fallback path. Header-parameter disjointness follows RFC 7516 §7.2.1.

2.3.3. Key Rotation

Holders MAY rotate the content-encryption key by re-encrypting the payload and re-signing the manifest. A `rotationReason` display field MAY record why rotation occurred, and `previousKid` MAY record the `kid` the current content-encryption key replaced — both are display/audit metadata with no processing semantics. Because the manifest signature covers the encrypted payload, any change to the ciphertext requires re-signing.

2.3.4. Recipient Revocation

A holder revokes a recipient by re-encrypting the payload without that recipient's `encrypted_key` entry and re-signing. `revokedRecipientKid` MAY record the `kid` of the removed recipient entry, and `revocationAction` MAY record the action taken (e.g. `"re-encrypted"`) — both are display/audit metadata with no processing semantics. Forward access to subsequently issued manifests is removed; a recipient retains the ability to decrypt manifest instances it already holds

(encryption is not retroactive). Deployments requiring retroactive revocation MUST rely on the manifest TTL and revocation status, not on the encryption layer.

2.3.5. Per-Facet Cryptographic Isolation PREVIEW

A deployment *claims per-facet isolation* for a named set of encrypted facets when it represents — in its conformance claim, in a profile it implements, or to a relying party — that a party able to decrypt one facet in the set is not thereby able to decrypt another. The obligation binds **only a deployment that makes the claim**; it does not outlaw shared-key encryption in general. The claim is carried as a conformance-claim flag and adds no manifest wire member, keeping isolation in symmetry with the discipline (EXT-T1 §T1.3.2) that a *gesture (or identifier) is not a proof of provenance*.

Under an isolation claim, the following structurally-checkable invariants apply across the declared isolated set:

- **No shared decryption-authorizing `kid`.** The same `recipients[].header.kid` MUST NOT authorize decryption of more than one facet in the isolated set. The comparison is across `facets[].entity.recipients[].header.kid` for the facets in the declared set.
- **No shared IV.** Two isolated facets MUST NOT reuse an initialization vector; identical `iv` across two facets in the set is a violation, including when their content-encryption keys and recipient `kid` values differ. This is the isolated-set profile's unconditional distinct-IV rule. The comparison is across `facets[].entity.iv` for the facets in the declared set.

These two are a cross-field structural rule of the same kind as the trust-tier floor check (§6.4.5).

An isolation claim also requires the fresh per-recipient ephemeral-key behavior of §2.3.2. Qualification MUST first demonstrate successful authorized decryption, then exercise every ordered cross-facet key attempt for the declared tested set. Reports MUST distinguish lookup refusal, unsupported algorithm, unwrap/authentication failure, and successful plaintext recovery, and identify the actual implementations and custody boundaries exercised. A lookup miss is not proof of cryptographic isolation; finite software tests do not establish hardware custody or global entropy quality.

Residual gap (necessary, not sufficient). Distinct `kid` strings and distinct `iv` values are NECESSARY but **not sufficient** for true key isolation. Because `kid` is opaque, two distinct `kid`s can still name HKDF- or HD-derived children of one root secret; an evaluator **cannot** verify true key isolation or true derivation-labeling from manifest bytes. True isolation depends on deployment key custody and derivation that Universal Manifest does not attest. This residual is declarable and

audited, never eliminated. The cryptographic property itself — that a grant to decrypt facet A yields no plaintext for facet B — is a behavioral property outside the structural conformance suite.

2.4. JWE Algorithm Constraints

2.4.1. Baseline Algorithm Pair

For encrypted facets (§2.3), conformant implementations MUST implement the algorithm pair **`ECDH-ES+A256KW`** (key management) with **`A256GCM`** (content encryption). This pair is the mandatory-to-implement baseline ensuring interoperable decryption across implementations. Holders encrypting facets MUST use this pair unless an alternative has been negotiated out-of-band.

The wrapping key MUST be derived using the SHA-256 Concat KDF of RFC 7518 §4.6.2. In this wrapping mode, AlgorithmID's length-prefixed ASCII data is the complete `alg` value, **`ECDH-ES+A256KW`**, and the output key length is 256 bits. Using `A256KW` alone, `enc`, or a proprietary KDF domain is not this algorithm. The RFC's direct `ECDH-ES` mode uses `enc` instead and is a different mode. Consumers MUST NOT try an alternate or historical KDF domain after candidate unwrap failure. Migration from experimental ciphertext using a different domain requires explicit authorized recovery, fresh conformant encryption, and re-signing of any containing manifest; existing ciphertext/signatures are not relabeled.

2.4.2. Evaluation Contract

An evaluator that implements encrypted-facet decryption MUST support the baseline pair. An evaluator encountering a JWE that declares an algorithm pair it does not support MUST treat the facet as a sealed entry (present but not read), exactly as it treats a facet for which it lacks a key. Additional algorithm pairs MAY be registered through the profile registration mechanism (EXT-OPT).

Implementation qualification note (informative, 2026-10-16). The detached Rust software candidate supports a finite P-256 subset: exactly `protected`, `recipients`, `iv`, `ciphertext`, and `tag`; protected `alg`/`enc`; recipient `kid`/`epk`; and public EC/P-256 `epk` members `kty`, `crv`, `x`, `y`. It rejects shared headers, alternative placements, unknown parameters, private `epk` material, `apu`/`apv` even when empty, `crit`, `zip`, and external `aad`. It requires canonical unpadded base64url, limits the encoded protected header to 1024 bytes, and accepts 1–256 distinct recipient identifiers of 1–1024 UTF-8 bytes, with distinct ephemeral points within a payload. These are

implementation limits, not new universal JOSE prohibitions or proof of complete JOSE support. The caller must supply authorized keys and appropriate entropy. A consumer that cannot process a supported-JOSE parameter MUST report that limitation and withhold plaintext rather than silently discard it; full implementation claims require evidence for the claimed surface. Primitive interoperability over arbitrary bytes does not establish that the decrypted value is a conformant JSON `um:Entity`.

2.4.3. Algorithm Negotiation

When two parties negotiate an alternative algorithm pair out-of-band (for example, in a bilateral session), each MUST still be able to fall back to the baseline pair, so that a manifest remains decryptable by any conformant recipient that did not participate in the negotiation.

(Worked example: facet with JWE inline encryption — see Cookbook.)

✧ ✧ ✧

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. Implementations MAY short-circuit the sequence at any stage by emitting a rejection receipt (§3.3).

3.1.1. Stage 1: Arrive

The manifest is received and its envelope structure becomes visible. The evaluator MUST consume the representation through a production rule it supports to obtain the abstract manifest, confirm the existence of the required abstract properties (**contextReference**, **id**, **type**, `manifestVersion`, `subject`, `issuedAt`, `expiresAt`, `signature`), and verify that **type** includes `um:Manifest`. The evaluator MUST ignore unknown properties for processing purposes but MUST

NOT remove them prior to signature verification. After verification succeeds, unrecognized properties have no normative semantics and MUST NOT affect evaluation outcomes. If consumption or structural validation fails, the sequence terminates with a `rejected` receipt.

This ensures extension fields survive the verification boundary while having no effect on 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:

1. **Signature verification** over the signing input defined by the production rule and declared signature profile (for the JSON-LD reference encoding under Signature Profile A, the JCS-canonicalized document per §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 (protocol in EXT-OPT).

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

Credential-binding verification sub-steps (Tier 1+). When the manifest relies on the credential-binding mechanisms for Tier 1 or higher assurance, the evaluator MUST additionally perform binding sub-steps 2a–2d (holder-binding, presentation-proof, liveness, cross-DID binding-proof) as part of this stage, each recording its outcome in the credential-binding receipt fields (§3.3.1.1). **These sub-steps are specified in full in EXT-T1.** Evaluators that do not implement credential binding skip them and record the corresponding statuses as `"absent"`, capping relied-upon claims at Tier 0.

Evaluators SHOULD allow a clock-skew tolerance of no more than 60 seconds for `issuedAt`/`expiresAt` comparisons. Manifests with `issuedAt` more than 60 seconds in the future SHOULD be rejected with `rejected` and `freshnessCheck: "stale"`. Deployments without NTP MAY configure wider tolerances but MUST document them 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. This is the bounded v0.5 **selective minimum disclosure** profile: each viewer,

verifier, system, or use case receives only the minimum information it is allowed or needs to see for that interaction. Selective disclosure is holder-controlled: the holder determines which facets are included in a given manifest instance, scoped by the verifier/audience, purpose, use case, and consent context. The holder MUST NOT include unrelated high-sensitivity facets merely because they exist in the holder's source container.

The evaluator MUST NOT assume the manifest contains the complete set of the subject's facets. Facets not included are not absent — they are not projected for this interaction. This bounded Base profile establishes projection, consent gating, sealed-entry handling, and honest receipt reporting. It does **not** by itself claim per-encounter unlinkability, cover-slot indistinguishability, resolver-operator privacy, or absence of residual fingerprinting; those stronger properties are not established by any v0.5 claim.

3.1.4. Stage 4: Consent

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

A purpose-bearing consent authorizes an operation only when the request supplies a purpose matching under §1.4.4. An omitted request purpose MUST NOT match for either disclosure or mutation. A consent missing its required purpose, or carrying an invalid purpose, MUST NOT be interpreted as an unrestricted grant. These rules clarify consent matching only; they add no request carrier, transaction/atomic-write mechanism, or current-state proof.

- If a `consents` entry governs the facet, the evaluator MUST verify that consent scope, purpose, and expiry are satisfied: the intended operation MUST appear in the consent's `scope` array, the intended use MUST fall within the consent's declared `purpose`, and the current time MUST be within the validity window (at or after `grantedAt`, before `expiresAt`). If a `withdrawnAt` field is present, whatever its value, the entry is withdrawn and not satisfied. When more than one entry governs a facet, every entry must be satisfied, so a second entry can only narrow what the first allows. A holder that grants two purposes on one facet's data issues two facets.
- If a facet is encrypted (§2.3) and the evaluator lacks a decryption key, the evaluator MUST acknowledge the facet as a sealed entry (present but not read), recorded with the facet status `"opaque"` in the receipt (§3.3.1).
- If no consent entry governs the facet and it is not a sealed entry, the evaluator MUST record the facet status as `"consent-missing"`.

Per-facet floors **PREVIEW** are evaluated together with consent in this stage: for a projected facet carrying a `requiredLiveness` floor (EXT-T1 §T1.3.1) or a `requiredAssuranceClass` floor (EXT-

T1 §T1.3.3), the evaluator MUST verify the floor here and withhold the facet if it is unmet (recording `assuranceInsufficient` for either unmet floor, with a reason identifying the failed floor and actual failure). An unavailable or failed liveness validation cannot satisfy the floor; the evaluator MUST NOT silently downgrade it. When more than one withholding condition applies to the same facet, the evaluator reports the first failing gate in the order sealed-entry → consent → floors.

Evaluators MUST NOT process facet data when required consent is absent, expired, or withdrawn. Consent-evaluation outcomes map to receipt statuses as follows: a matching entry is satisfied (operation in `scope`, use within `purpose`, in-window, no unmet condition) → the facet is processed and its consent status is `"valid"`; no entry matches → `"consent-missing"`; a matching entry exists but operation is not in `scope`, use is not within `purpose`, or a condition is violated/unenforceable → `"consent-denied"` (with consent status `"scope-mismatch"`, `"purpose-mismatch"`, or `"condition-violated"`); a matching entry is expired or withdrawn → `"consent-denied"` with consent status `"expired"` or `"withdrawn"`; the evaluation time is before a matching entry's `grantedAt` → `"consent-denied"` with consent status `"condition-violated"`. The per-consent `consentStatuses` values are therefore `"valid"`, `"scope-mismatch"`, `"purpose-mismatch"`, `"condition-violated"`, `"expired"`, and `"withdrawn"`.

Write authorization. PREVIEW A write, update, or append to a facet is a distinct authorized operation, governed by a **target-facet write-consent**. The evaluator MUST NOT write unless a write operation is literally present in the governing consent's `scope` (consistent with the fail-closed-on-unrecognized-scope rule, §1.4.4), the `purpose` matches, and the consent is active. The governing consent is the one whose `facetRef` matches the *target* facet's `@id`. Absent such a consent, the evaluator MUST fail closed: it refuses the write and records `"consent-missing"` (no governing entry) or `"consent-denied"` with `"scope-mismatch"` (entry exists, write verb not in `scope`). Write authorization is **not a new pipeline stage**: it reuses this Stage-4 consent-matching machinery, applied at write time rather than only over the projected-facet read pass. New consent-`scope` operation tokens follow the `<domain>.<capability>` dot convention of the agent-delegation scope registry (EXT-OPT); legacy bare-word verbs (`read`/`display`/`cache`/`process`/`forward`) remain valid, so a `scope` such as `["read", "write.update"]` carries intentional mixed punctuation. The candidate write-verb token family is: `write.update` (modify an existing facet's content), `write.create` (bring the pre-declared target facet into existence — the creation path below), and `write.append` (add content without modifying what exists); further write verbs register through the scope registry (EXT-OPT) like any other operation token.

When a write would **create** a facet that is not yet present (for example, durable agent memory — see §7.2), there is no `@id` for any existing `consents[].facetRef` to match, so under the plain Stage-4 rule the write is `"consent-missing"` and MUST be refused (fail closed). To authorize

creation, the deployment MUST **pre-declare** a consent whose `facetRef` names the intended target facet `@id` (the holder commits the target `@id` ahead of the write). Write-replay protection uses the same anti-replay model as a presentation proof (EXT-T1): a write operation MUST carry a per-write nonce, or the evaluator MUST dedupe writes; a write performed under an unlock window (EXT-OPT) MUST still be individually replay-guarded. A successful write is recorded with the receipt facet status `"written"` (§3.3.1); a denied write reuses `"consent-denied"`/`"consent-missing"`. The transport by which a write request and its per-write nonce reach the evaluator is deployment/protocol-layer by design; this stage defines the authorization and replay-guard requirements, not the carrier.

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` — accepted but one or more non-critical conditions were noted (e.g., revocation status could not be checked). An unsatisfiable manifest-level floor is **not** a non-critical condition — it produces `rejected` (below), not `accepted-with-warnings`.
- `accepted-partial` — not every facet was processed: at least one was withheld, sealed, rejected, or lacking consent. A result may process no facet at all, so a reader reads each facet's status.
- `rejected` — the manifest failed a mandatory check (signature, freshness, structural validity — including cross-field constraints such as a below-floor tier declaration (§6.4.5) or an expired agent delegation (§6.5.1) — or a malformed consent entry), or an unsatisfied required floor/tier applies at the manifest level and no processable items remain (for example, a manifest whose only facet is withheld because a required assurance/liveness floor or a facet-level `requiredTrustTier` cannot be met — the assurance-floor case is exercised by `valid/manifest-locked-facet-below-floor.jsonld`). Whether an application still *renders* anything after a `rejected` evaluation is an application-layer choice and does not change the receipt outcome.

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 (§3.3) that honestly records what the evaluator actually did, capturing the outcome of each preceding stage. Evaluators MUST NOT omit failed

checks or suppress negative outcomes.

An evaluator MUST NOT report as verified any check it did not perform or could not complete. Presence, successful parsing, structural validation and verification of an outer signature do not establish verification of enclosed evidence. An unsupported or unexecuted check MUST be distinguishable from a successful check wherever the receipt claims an outcome for that evidence.

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 (§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.

3.3.1. Receipt Fields

A receipt MUST include:

- `@type` — MUST include `"um:Receipt"`.
- `manifestId` — the `@id` of the processed manifest.
- `outcome` — one of `"accepted"`, `"accepted-with-warnings"`, `"accepted-partial"`, `"rejected"`.
- `signatureCheck` — `"valid"`, `"invalid"`, `"unsupported-profile"`, or `"not-evaluated"`.
- `freshnessCheck` — `"fresh"`, `"expired"`, `"stale"`, or `"not-evaluated"`. `"stale"` indicates `issuedAt` is in the future relative to the evaluator's clock (beyond clock-skew tolerance).

A manifest rejected during Arrive (structural failure) terminates before signature/freshness checks; stages not reached record `"not-evaluated"`.

A receipt MUST include a `facetStatuses` array (empty when the manifest has zero facets). Each entry is a per-facet status object with `facetId` (matching the facet's `@id`), `status` (`"processed"`, `"opaque"`, `"consent-denied"`, `"consent-missing"`, `"trustTierUnsupported"`, `"assuranceInsufficient"`, `"not-projected"`, or `"written"`), an OPTIONAL `name`, and an OPTIONAL `reason`. `"opaque"` records a **sealed entry** — an encrypted facet the evaluator could not decrypt, acknowledged as present but not read (§2.3, §3.1.4). `"trustTierUnsupported"` records a facet withheld because its `requiredTrustTier` cannot be verified (§6.4.5). `"assuranceInsufficient"` records a facet withheld because its `requiredAssuranceClass` floor (the locked-tier portable-unlock profile — EXT-T1 §T1.3.3) or its `requiredLiveness` floor (EXT-T1 §T1.3.1) is unmet, including when required liveness validation is unavailable or fails. The reason identifies the failed floor and actual failure; it does not imply that unvalidated liveness was verified. A liveness-only refusal does not imply an assurance-class failure or require an `assuranceStatus` echo. The facet's existence may be visible but its content is never released, parallel to `"trustTierUnsupported"`. `"written"` records a successful write authorized under the write-consent rule (§3.1.4); reads keep `"processed"`. `"not-projected"` indicates the evaluator's local policy expected a specific facet absent from the presented manifest; it is OPTIONAL and applies only when the evaluator has prior knowledge of the subject's schema.

A receipt SHOULD include, when applicable, the following fields (the receipt *field surface* a Tier-0/Tier-1 evaluator records): `revocationStatus` (`"active"` / `"revoked"` / `"suspended"` / `"unchecked"`); `revocationReason` (registry-coded); `keyRefResolution` (`"resolved"` / `"unresolved"` / `"not-attempted"`, where `"unresolved"` or `"not-attempted"` caps `keyRef`-derived identity assurance at Tier 0, and `"resolved"` says only that `keyRef` resolved to this key, not that the signer controls the `subject`); `consentStatuses` (per-consent status objects: each entry carries `facetId`, `consentRef`, `status` — one of the §3.1.4 per-consent values — and an OPTIONAL `checkedAt`, the RFC 3339 time the consent was evaluated); `claimStatuses` (per-claim status objects with `claimRef`, `status` ∈ {`"verified"`, `"unverified"`, `"failed"`, `"unprocessable"`, `"trustTierUnsupported"`}, optional `tier`/`reason`); `unprocessedEntries` (structural entries preserved but not acted on); `processedAt`; `warnings` (each with a registry `code` and a `message`); `exchangeId` and `evaluatorId` (REQUIRED on receipts produced in a bilateral session — EXT-OPT); `receiptId`; and `receiptSignature` (optional, following Signature Profile A).

When computing `receiptSignature`, apply the §1.6.3 signing-input procedure to the receipt object with `receiptSignature` taking the place of `signature` — it is removed from its own signing input (avoiding the circularity that a literal reading of §1.6.3, whose exclusion list names only `signature`, would otherwise create for a receipt that has no top-level `signature`). When a receipt

is promoted to a first-class manifest (§3.3.2) and carries both a manifest `signature` and a `receiptSignature`, the `receiptSignature` is computed first and is then covered by the manifest `signature` (which excludes only `signature`, `presentationProof`, and `postQuantumSignature`, not `receiptSignature`).

Because `receiptSignature` is removed from its own signing input, its `created` and `keyRef` members are not authenticated by it, and a change to either does not invalidate it. An evaluator or consumer MUST NOT use `receiptSignature.created` as evidence of when the receipt was signed, and MUST NOT attribute a receipt to an author on the strength of `receiptSignature.keyRef`; receipt authorship requires a separate check of the verifying key against the author's authority. These statements concern the signature only: `receiptSignature.created` keeps the RFC 3339 `date-time` rule of this specification, and a value that does not conform fails validation of the receipt, whether or not the signature verifies.

Receipt signing is RECOMMENDED for accountability use cases but is not required for v0.5 conformance. Unsigned receipts are valid evaluation outputs but provide weaker non-repudiation.

3.3.1.1. Credential Binding Status Fields

When the evaluator processes credential-binding mechanisms (Tier 1+, specified in EXT-T1), the receipt SHOULD record their outcomes using `holderBindingStatus`, `presentationProofStatus`, `livenessStatus`, `crossDidBindingStatus`, and `effectiveTrustTier`. When a facet carries a `requiredAssuranceClass` floor (the locked-tier portable-unlock profile — EXT-T1 §T1.3.3), its corresponding `facetStatuses[]` entry SHOULD additionally record `assuranceStatus` — an object echoing the evaluator-determined `assertedClass` (one of `"software"`, `"hardware-uv"`, `"hardware-bound"`) and `met` (a boolean: whether it satisfied that facet's floor). Each echo describes its own facet, rather than substituting a receipt-wide value for different facet results. This extends the consent-record lineage with the authenticator-strength dimension, beside the existing `livenessStatus` freshness echo. A genuinely absent presentation proof records `presentationProofStatus: "absent"`. A present proof whose verification was not executed records `"not-evaluated"`, never absence or success. Interactive verification requiring that proof MUST refuse an absent, failed or not-evaluated proof. Other unsupported binding mechanisms remain distinguishable from verified outcomes; these diagnostics do not establish proof execution or identity authority. **`effectiveTrustTier` is the highest trust tier the evaluator actually verified** for the claims it relied on, per §6.4.2; a claim without a verified `holderBinding` contributes at most Tier 0. `effectiveTrustTier` is independent of the declared `requiredTrustTier`; the evaluator MUST NOT act on any item whose `requiredTrustTier` exceeds the `effectiveTrustTier` it achieved. Recording both makes the

gap between declared and verified trust explicit. The exact value sets for these fields are specified in EXT-T1.

Preview decision: One open decision remains on `effectiveTrustTier`'s status — REQUIRED on every v0.5 receipt (recommended direction) versus SHOULD conditional on credential-binding support, as built here. Flag to revise.

(Worked example: structured receipt — see Cookbook.)

3.3.2. Receipt as a First-Class Manifest Class PREVIEW

Beyond its role as the terminal output of a single evaluation, a receipt is itself a **first-class manifest class** (§1.0.1): a signed, portable record that can be chained, retained, and independently verified, carrying the common envelope members with `@type` including both `um:Manifest` and `um:Receipt`. This promotion is additive — an evaluator that only emits inline receipts remains conformant. The chain-integrity members (`chainId`/`seq`/`prevHash`), the typed event vocabulary, the structured reason registry, the session-scoped signing-key authorization, and the optional transparency-log anchoring profile are specified in full in EXT-OPT (O3, Receipt as a First-Class Manifest Class).

3.4. statusRef Resolution Schema

`signature.statusRef` (§1.6.2) resolves to a **status endpoint** that reports whether a manifest instance is `active`, `revoked`, or `suspended`. The HTTP resolution procedure, response schema, status semantics, conditional-request/caching behavior, and error taxonomy are specified in EXT-OPT (O2, statusRef Resolution Schema). In the Base, an evaluator that implements revocation-aware verification follows that protocol; an evaluator that does not records `revocationStatus: "unchecked"` (§1.6.6).

3.4.1. Current Policy-State Resolution Before Governed Action (Informative Composition)

This subsection is informative. It names a composition of existing normative machinery; the requirements themselves live in the cited sections and are not changed here.

Before a governed asset is used or a governed or agentic action proceeds, an evaluator often needs an answer to a plain-language question: **is the consent, authorization, or status state**

carried by this manifest still current? Consent and authorization change over time — through renewal, withdrawal, expiry, supersession, and revocation — and a cached record can go stale. The composition below supplies bounded checks, conditional on independently established authority and target binding; a valid envelope signature and an HTTPS status response alone do not establish current permission. The composition applies to any governed asset or action — 3D assets, scans, media, digital replicas, embodied-person representations, and agentic actions alike — not to any single asset kind.

- **Freshness bounds.** `issuedAt`/`expiresAt` TTL enforcement (§3.1.2) bounds how stale any carried state can be; an expired manifest is rejected.
- **Manifest-instance status.** `signature.statusRef` resolution (§3.4, EXT-OPT) can report `active`, `revoked`, or `suspended` only with the trusted authority/target validation in O2.1. Opaque cursors have no implicit ordering, and `nextCheck` is a revalidation hint rather than permission to extend evidence validity. A superseding record can be represented by revoking the old manifest instance and issuing a new one; neither a successor locator nor an HTTP cache validator authorizes that successor.
- **Per-record status.** A `claimProof` entry may carry its own `statusRef` (EXT-OPT), and key state is resolved through the key's DID document (§1.6.4); manifest status, claim status, and key status are distinct objects of resolution and are not conflated.
- **Consent currency.** Consent entries carry validity windows and withdrawal (`grantedAt` / `expiresAt` / `withdrawnAt`, §1.4.4); expired or withdrawn consent is not satisfied, and processing fails closed (§3.1.4).
- **Delegation currency.** An agent delegation expires at its `expiresAt` and may expose a `livenessEndpoint` for real-time delegation status (§6.5); absent that endpoint, the delegation is static for the manifest's TTL.
- **Fail-closed / policy-gated behavior.** When revocation status cannot be determined and local policy requires active status, the manifest is rejected for use (§1.6.6); an unreachable status endpoint is recorded as `unchecked` with a structured reason (EXT-OPT), and a facet whose consent cannot be verified is not processed (§3.1.4).
- **Receipt evidence.** The receipt records the resolution outcome — `revocationStatus`, per-consent `consentStatuses`, structured reason codes, and typed events such as `consent-withdrawn` / `consent-expired` (§3.3, EXT-OPT) — so the disposition of a governed action is auditable.

This composition is the bounded v0.5 behavior for current policy-state resolution. It does **not** yet define a unified policy-record resolution profile beyond these mechanisms — for example, a general supersession pointer between consent or authorization records, a normative policy-window

taxonomy for deny (fail-closed) / escalate dispositions, or receipt reason codes specific to `superseded` and `escalated` outcomes. A candidate registered profile covering exactly those three pieces is drafted as the **Current Policy-State Profile** in EXT-OPT (PREVIEW, design-stage: its wire shape may change and it carries no conformance tests until finalized); deployments MAY also carry equivalent semantics as profile-defined extensions through the profile registration mechanism (EXT-OPT).

An authoritative source may check current local permission without publishing a queryable status pointer; a stale replica is not that authority. The protected signed-status/current-state mechanism, its authenticated carrier and any high-privacy resolution suite remain unresolved here. No new signature suite, wire member, or private permission mechanism is selected by this clarification; when required current permission cannot be established, the governed action remains denied.

3.5. Bilateral Session Model PREVIEW

In a bilateral exchange both parties present and evaluate manifests (§6.4.6). The **bilateral session model** — session objects, paired-receipt correlation via `exchangeId`, session identifiers (`sessionId`), session lifecycle states, and transport independence — is a protocol-layer concern specified in EXT-OPT. The Base carries the bilateral *field surface* it depends on: `exchangeId` and `evaluatorId` on receipts (§3.3.1), and the bilateral-exchange security model (§6.4.6).

A distinct single-subject construct, the **unlock window** — a time-bounded grant opening a holder-enumerated set of facets, explicitly *not* a `um:BilateralSession` — is also specified in EXT-OPT. An unlock window never lowers a per-facet `requiredLiveness` floor (EXT-T1 §T1.3) or a per-facet isolation obligation (§2.3.5), and a write performed under one is individually replay-guarded (§3.1.4).

✧ ✧ ✧

4. Conformance

Conformance is defined against the abstract data model, 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; the JSON-LD reference encoding is the encoding implementations are expected to support for interoperation. The behavioral requirements below apply to the abstract manifest regardless of encoding (§4.5).

4.1. Evaluator Behavior

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

Evaluators claiming v0.5 conformance MUST implement the six-stage evaluation sequence (§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 (§3.3) for every processed manifest.
3. Treat encrypted facets as sealed entries when it lacks a decryption key, recording them as present.
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 SHOULD bound `expiresAt` to a sensible interaction lifetime (typically hours or days); a "sensible" lifetime is a policy judgment, not a testable bound, and legitimate long-lived manifests exist (venue, device, registered-service, and asset-bound manifests are not re-signed hourly). The enforceable staleness controls remain mandatory regardless of window length — expiry checking (§3.1.2), revocation via `statusRef` (§1.6.6), and freshness checks — and an evaluator MAY reject an over-long window by local policy. Holders MUST sign every manifest prior to distribution using Signature Profile A (§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 key resolution succeeds at the edge and identity assurance is not capped at Tier 0.

Identity across a change of control (informative). The `subject` identifier, not `@id`, carries identity across issuances. A subject keeps its identifier through a change of control only if its identifier method lets the authorized keys change, as an updatable DID document does. The new controller's key then appears under the relationship that EXT-T1 §T1.5.3 requires, and the new controller signs later manifests. An identifier derived from a key, such as a `did:key` subject, cannot

do this: a new key means a new identifier. An `identity.crossDidBinding` claim (§6.4.4) is the available way to relate the two identifiers, with the limits in EXT-T1 §T1.6.3. This version defines no handover operation: how control moves and who authorizes it belong to the identifier method and the deployment. A new manifest does not withdraw an earlier one, which stays usable until its `expiresAt` unless a status or key-revocation check says otherwise. A new manifest holds only the consent entries its signer includes, and an entry left out is absent consent.

v0.5 adds these holder obligations:

- Holders encrypting facets MUST use the baseline JWE algorithm pair (`ECDH-ES+A256KW` / `A256GCM`, §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 (EXT-T1).
- When presenting a manifest in response to a verifier-issued challenge (an interactive presentation), the presenter MUST include a `presentationProof` (EXT-T1).

4.3. Bilateral Participant Behavior

A Conformant Bilateral Participant MUST implement both Evaluator Behavior (§4.1) and Holder Behavior (§4.2). In a bilateral exchange (§6.4.6), both parties independently evaluate the other's manifest. The *interaction tier floor* is the maximum of either party's `requiredTrustTier` — a negotiated floor distinct from the `effectiveTrustTier` each party records (§3.3.1.1).

4.4. Standalone Conformance Suite

Implementations validate conformance claims natively by testing against the official `conformance/` suite — fixture validation (accepting valid stubs and correctly isolating/flagging malformed artifacts like missing contexts or expired manifests). Implementations claiming v0.5 conformance MUST satisfy the normative requirements of this specification. The standalone conformance suite is the canonical evidence mechanism; implementations SHOULD publish their suite results and claimed level per the conformance profile served at https://universalmanifest.net/conformance/v0.5/ (the same suite is `conformance/v0.5/` in the specification repository). The v0.5 suite extends the v0.3 suite with tests for newly normative features. Design-stage preview profiles (EXT-OPT O2.6/O12/O13/O14) have no conformance tests until finalized; other PREVIEW features may carry structural (shape-only) fixtures while their enforcement profiles remain PREVIEW, as itemized in the conformance summary (`spec/v0.5/CONFORMANCE.md` §9 in the specification repository — the two-grade PREVIEW policy).

4.5. Conformance to the Abstract Data Model

The requirements in §4.1–§4.3 are stated in terms of the abstract data model and apply identically regardless of which production rule serialized a manifest. A conformant implementation MUST support at least one production rule and, for interoperation, SHOULD support the JSON-LD reference encoding; MUST consume supported representations into the abstract data model before applying behavioral requirements, and MUST reject an unsupported encoding rather than misinterpreting it; MUST preserve every abstract property across a production/consumption round-trip when it both produces and consumes an encoding; and MUST verify the integrity proof against the bytes of the production it consumed (a signature is bound to one encoding). **Supporting only the JSON-LD reference encoding is sufficient for full conformance**; supporting CBOR-LD (EXT-OPT) is OPTIONAL. The full ADM conformance detail is in EXT-OPT.

4.6. Status Endpoint Conformance

A **conformant status endpoint** is the conformance class that responds to `signature.statusRef` resolution. Its obligations (response schema, status semantics, conditional requests) are specified with the statusRef protocol in EXT-OPT. Resolver conformance (the federated-status conformance class) is deferred pending the open decision on whether federation moves to a companion specification.

4.7. Optional-Feature Matrix

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

Feature	Conformance	Mandatory baseline when not implemented
Signature Profile B (JCS + ES256 — EXT-OPT §O15)	OPTIONAL	Verify Signature Profile A, which is mandatory to implement (§1.6); reject a Profile-B-signed manifest, recording `signatureCheck: "unsupported-profile"` (§1.6.5); never reinterpret it under Profile A.
Encrypted-facet decryption (§2.3)	OPTIONAL	Sealed-entry handling is REQUIRED: record the facet as present-but-sealed;

Feature	Conformance	Mandatory baseline when not implemented
		never infer its content.
Revocation-aware verification (EXT-OPT)	RECOMMENDED	Record `revocationStatus: "unchecked"`; do not grant revocation-gated trust.
Credential binding (EXT-T1)	OPTIONAL (REQUIRED for Tier 1+)	Record `holderBindingStatus: "absent"`; cap relied-upon claims at Tier 0.
Per-facet floors (`requiredLiveness`, `requiredAssuranceClass` — EXT-T1 §T1.3.1 / §T1.3.3) PREVIEW	OPTIONAL (PREVIEW)	A facet carrying a floor MUST be withheld (recorded `assuranceInsufficient` for either unmet floor, with the actual failure reason), never processed, by an evaluator that does not implement floor enforcement — fail closed, without silent downgrade.
CBOR-LD encoding (EXT-OPT)	OPTIONAL	Support the JSON-LD reference encoding; reject unsupported encodings rather than misinterpreting them.
Transparency anchoring (EXT-OPT)	OPTIONAL	Verify the manifest signature and proofs directly; do not require a transparency log.

The complete optional-feature matrix, including modules introduced by extensions, is consolidated in EXT-OPT. Claims MUST distinguish implemented, partial, and not-implemented obligations with actual executed evidence and the mandatory baseline for every unmet check, as specified in EXT-OPT O11.1. A partial module declaration cannot waive a requirement of the conformance class or raise verification, assurance, or trust tier.

✧ ✧ ✧

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 (§3.1.1), domain-specific profiles do not compromise cross-system interoperability.

The extension model in brief. Universal Manifest grows by *profiles*: self-contained documents that add manifest classes, claim types, signature profiles, trust profiles, or binding profiles, each

registered under a canonical `um:profile:<category>:<name>` identifier and each constrained never to redefine a common envelope member. The formal **profile registration mechanism** (the five-step IANA-style process, the category scheme, conflict resolution, deprecation, and registry hosting) is specified in EXT-OPT.

Extensions Index. This document set comprises the Base plus the following companion documents:

Document	Scope	Registry category emphasis
EXT-T1 — Tier-1 Binding Profile	Credential-binding mechanics that satisfy the Tier-1 requirement: holder binding, presentation proof, liveness, the end-to-end claim-proof path, Stage-2 sub-steps, binding receipt fields.	`binding`, `trust`
EXT-T2 — Tier-2 Cryptographic-Binding Profile	Zero-knowledge cross-DID proof profiles (2A/2B), the BBS+ selective-disclosure track, and post-quantum signature migration.	`trust`, `signature`
EXT-T3 — Tier-3 Multi-Party Ceremony Profile	The multi-party ceremony model, attester-role taxonomy, and threshold-protocol guidance.	`trust`
EXT-OPT — Optional-Feature Profiles	Tier-orthogonal optional capabilities: abstract data model & CBOR-LD, statusRef protocol, receipts-as-class, bilateral sessions, federation, profile registration, `trustWeight`/category trust, extended `devices`/`actorState`/sensor-consent schemas, crypto-requirements summary, class registry snapshot, and the design-stage candidate profiles for current policy-state resolution, asset-bound represented-person consent, signed referral-route attribution, and GeoPose primary pose values.	`domain`, `signature`, all
Cookbook	All worked examples, by scenario, tagged by tier. (Informative.)	—



6. Security Considerations

Universal Manifest v0.5 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. The cryptographic *constructions* that realize the higher tiers are

specified in the tier extensions; this section states the properties and the Tier-0/Tier-1 requirement surface.

6.1. Signature Limitations

Implementers MUST use Signature Profile A (§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 is the primary defense against presentation replay spoofing. For interactive presentations, the `presentationProof` mechanism (EXT-T1) additionally binds a presentation to a specific verifier and challenge, preventing replay within the TTL window.

Clock trust. Every time check in this specification compares against the evaluation time: freshness, consent windows, liveness, unlock windows, delegation expiry and claim expiry. These checks are only as strong as the clock that supplies that time. On a platform with no trusted or attested time source, a party that controls the local clock can defeat expiry, and a receipt does not show otherwise. The Rust reference evaluator takes the evaluation time only as an explicit input and never reads an ambient clock. The TypeScript reference evaluator reads the host clock when the caller supplies no evaluation time; a caller that needs an auditable time passes it explicitly.

6.3. Resource Limits

Denial-of-Service vectors from inflated arrays or recursion SHOULD be countered with hard limits on payload ingestion. Evaluators SHOULD enforce at least the following defaults 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. A megabyte here is 1,000,000 octets, measured over the manifest's JCS-canonical form. Depth counts containers, with the manifest object as level 1. The array limit applies to every array.

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), or that an issuer actually issued a listed claim (claim authenticity) — `claims[].issuer` 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, and 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** (EXT-T1): `claimProof` proves issuance, while `holderBinding` proves the presenting subject is the credentialed holder. (Exception: on an `identity.crossDidBinding` claim whose `boundDids` includes the manifest `subject`, presenter binding is established by the subject-signature key-authorization check together with attester or proof verification, so a separate `holderBinding` is OPTIONAL on such claims — EXT-T1 §T1.1.) Suitable for medium-stakes use cases (social identity, reputation, basic access control). *(Binding mechanics: EXT-T1.)*

Tier 2 — Cryptographic binding. Medium friction. Cross-DID control is cryptographically proven via zero-knowledge proof, without revealing private key material. Evaluators claiming Tier 2 assurance MUST verify the ZK proof before granting Tier 2 trust. Suitable for high-stakes Sybil-resistance. *(Proof profile: EXT-T2.)*

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. *(Ceremony model: EXT-T3.)*

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.

Informative: the trust tiers grade binding strength — how strongly a claim is cryptographically tied to the presenting subject — not identity-proofing assurance. They are not equivalents of Levels of Assurance or of the NIST SP 800-63 IAL/AAL classes, and a tier MUST NOT be reported as satisfying such a level. (An orthogonal `requiredAssuranceClass` floor grades authenticator strength; its informative crosswalk to external assurance frameworks is in EXT-T1 §T1.3.3.)

6.4.3. `claims[].claimProof` Field

The `claimProof` field is an OPTIONAL property on any claim object, carrying proof material demonstrating claim issuance to the manifest subject. This field enables Tier 1 verification. It is named `claimProof` rather than `evidence` to avoid collision with the W3C Verifiable Credentials Data Model v2.0 [VC-DATA-MODEL] `evidence` property.

`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 (supporting claims backed by multiple independent proofs). Existing manifests with a single-value `claimProof` remain valid; the array form is additive and non-breaking.

When `claimProof` is an object or array element, each proof entry MAY carry `proofType` (RECOMMENDED values: `VerifiablePresentation`, `DataIntegrityProof`, `sd-jwt-kb`, `pop-jws`, `evidence-pointer`; if absent, inferred from structure) and `proofPurpose` (per W3C DID Core §5.3 [DID-CORE]; if absent, assume `assertionMethod`). Proof entries MAY contain additional properties (`@type`, `verifiableCredential`, `proofValue`, `verificationMethod`, `statusRef`); evaluators MUST preserve unrecognized properties.

Verification. At Tier 0 proof checks are OPTIONAL. At Tier 1+ the evaluator MUST verify the proof — the VP proof chain for an embedded object, fetch-and-verify (or record `"unverified"`) for a URI string, and the conjunction of all entries for an array — and MUST enforce size limits of at most 50 KB per embedded VP and 500 KB total VP payload across all claims. **The full verification procedure — the VP proof-chain steps, the claimproof-unresolved handling, the audience/challenge replay protections, and the end-to-end claim-proof path with key-authorization checks against DID Core verification relationships — is specified in EXT-T1.** It is the mechanic that turns the Tier-1 requirement stated here into running code.

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.

A `crossDidBinding` claim MUST contain `@type` (`"identity.crossDidBinding"`), `issuer` (inherited from the base claim schema), and `boundDids` (an array of DID strings asserted as controlled by the same entity; MUST contain at least 2 DIDs, one of which MUST match the manifest `subject`).

An **attester-asserted** binding — offered for Tier 1 evaluation on the strength of an attester's assertion rather than a cryptographic proof — MUST additionally contain `attester` (DID or URI of the attesting entity), `attestationMethod` (human-readable description of the verification method), and `attestedAt` (RFC 3339). When the claim instead carries a cryptographic `bindingProof` (EXT-T2) or `ceremonyProof` (EXT-T3), these three are OPTIONAL: the proof object carries the binding.

A `crossDidBinding` claim MAY contain `claimProof` (URI/object/array pointing to the attestation proof), `expiresAt` (attestation expiry; evaluators SHOULD reject expired attestations), `bindingProof` (a Tier 2 ZK proof — EXT-T2), and `ceremonyProof` (a Tier 3 ceremony proof — EXT-T3).

When `bindingProof` is present, `boundDids` MUST contain exactly two DIDs; the v0.5 schema applies that upper bound only under this condition. A ceremony-only claim may bind more than two DIDs. For `ceremonyProof`, M and N in `threshold` MUST be positive integers with M ≤ N, and the attester IDs MUST be unique with M ≤ distinct attesters ≤ N (EXT-T3 §T3.1). Passing these shape and threshold checks does not select a protocol, verify an aggregate, authorize an attester or establish common control.

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. Evaluators SHOULD maintain a configurable attester trust list. Multiple binding claims for overlapping DID sets are independent assertions, not cumulative proof.

(Worked example: cross-DID binding claim — see Cookbook.)

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 — 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. The manifest-level value sets the floor; claim/facet-level values can only raise it, not lower it. A claim- or facet-level `requiredTrustTier` lower than the manifest-level value is a cross-field structural violation: evaluators MUST reject the manifest (`rejected`), exactly as for other structural-validity failures (§3.1.5).

If a manifest, claim, or facet specifies a `requiredTrustTier` for which the evaluator has no implemented verification profile (e.g., Tier 3, where the ceremony profile is under development — EXT-T3), the evaluator MUST treat the item as unverifiable and record it as `"trustTierUnsupported"` — in `claimStatuses` for claims, `facetStatuses` for facets, and `crossDidBindingStatus` for binding claims. The evaluator MUST NOT downgrade to a lower tier. The overall outcome is `"accepted-partial"` if other items can still be processed, or `"rejected"` if the unsupported tier applies at the manifest level or no processable items remain (§3.1.5).

(Worked example: manifest with requiredTrustTier — see Cookbook.)

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 and the venue verifies Alice's

claims at the tier the venue requires; the venue presents its UM to Alice and Alice verifies the venue's claims at the 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 — a negotiated floor, distinct from the `effectiveTrustTier` each party verifies and records (§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 (one party cannot satisfy the other'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.

*(The bilateral **session protocol** — session objects, lifecycle, paired-receipt correlation — is specified in EXT-OPT.)*

6.4.7–6.4.9. Advanced Binding Profiles (named; specified in extensions)

- **§6.4.7 Tier 2 ZKP Proof Profile (PREVIEW)** — the zero-knowledge proof profiles (Profile 2A: BBS+ linked-secret; Profile 2B: HD-derivation via Groth16) for cryptographic cross-DID binding, carried in the `bindingProof` field. Specified in EXT-T2.
- **§6.4.8 Tier 3 Multi-Party Ceremony (PREVIEW)** — the ceremony model, attester-role taxonomy, and threshold-protocol guidance, carried in the `ceremonyProof` field. Specified in EXT-T3.
- **§6.4.9 Claim Proof Process — End-to-End Verification Path** — the 7-step verification path from claim to trust decision, including key-authorization checks against W3C DID Core verification relationships. Specified in EXT-T1 (it is the connective procedure that ties §6.4.3, §6.4.4, and the Stage-2 sub-steps together).

6.5. Agent Delegation

The `um:agentDelegation` pointer (§1.4.5) declares that the manifest holder has delegated session authority to an agent (a bot, assistant, or automated process) acting on the subject's behalf.

6.5.1. Structure

A `um:agentDelegation` pointer MUST contain `@type` (`"um:agentDelegation"`), `delegateType` (one of `"ai-agent"`, `"bot"`, `"proxy"`, `"human-delegate"`), `delegatedBy` (the DID of the delegating subject; MUST match the manifest `subject`), `delegatedAt` (RFC 3339, when delegation was granted), and `expiresAt` (RFC 3339). An expired delegation is a structural violation of the delegation pointer: evaluators MUST NOT honor it, and MUST reject the manifest (`rejected`) rather than process it with the delegation ignored — an expired grant of authority is not safely severable from a manifest presented under it (§3.1.5). This is an explicit exception to the base-pointer rule that expired pointers are ignored (§1.4.5). It MAY contain `delegateId` (the DID/identifier of the delegate; REQUIRED whenever the delegation is intended to be exercised), `scope` (an array of capability strings the delegate may exercise), and `livenessEndpoint` (a URI for real-time liveness/delegation status). **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 the reference resolves.

6.5.2. Platform Guidance

Evaluators SHOULD display delegation status to other users when present, MAY require human-only sessions for high-stakes actions (financial, governance), and 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. Platforms SHOULD also surface the operating executor to the relying party (see the `actorState` member, §1.4.7). The standardized **agent-delegation scope registry** (the namespace convention, the six core scope values, evaluator behavior for unrecognized scopes, and the registration procedure) is specified in EXT-OPT.

(Worked example: agent delegation pointer — see Cookbook.)

6.6. Credential Binding Security Considerations

Credential binding is the set of mechanisms that turn the **Tier-1 requirement** into a verifiable property: `holderBinding` (proving the presenting subject is the credentialed holder), `presentationProof` (proof-of-possession binding a presentation to a specific verifier and moment), and `livenessAttestation` (evidence of recent interactive human authentication), together with their security considerations (BBS+ pseudonym scope, ZK trusted-setup, reciprocal-

control limitations, and v0.3 backwards compatibility). **This entire subsystem is specified in EXT-T1.** The Base names it, requires it for Tier 1+ (§6.4.2, §4.2), and carries its field surface on claims (§1.4.3) and receipts (§3.3.1.1). One signature-coverage property deserves explicit statement: within a `livenessAttestation`, the `validUntil`, `method`, and `userVerified` members are covered by the holder's manifest signature, not by the attester's proof (EXT-T1 §T1.3); an evaluator MUST treat them as holder-asserted except where the declared `proofType`'s mechanics attest them. *(Wire-freeze rider, recorded for the schema-lock decision set: fold `validUntil` into the attester-signed content so the attestation's validity window is attester-controlled.)*

A facet MAY additionally declare an OPTIONAL **`requiredLiveness`** floor — a per-facet minimum liveness freshness (and optional user-verification requirement) that gates processing of that facet. It is a sibling of the optional `requiredTrustTier` facet member (§2.1) but, unlike the closed integer `requiredTrustTier`, is a compound object (a freshness class plus a boolean). Its shape, the floor comparison order, and the fail-closed sequencing are specified in EXT-T1 §T1.3; the floor is enforced in the Consent stage (§3.1.4) and recorded in the receipt.

A facet MAY also declare an OPTIONAL **`requiredAssuranceClass`** floor — a per-facet minimum *authenticator assurance class* (a closed, UM-neutral ordered enum `software` < `hardware-uv` < `hardware-bound`, default `software`) that, like `requiredTrustTier`, can only raise the bar and is fail-closed (a facet below its floor is withheld, recorded `assuranceInsufficient` — §3.3.1). It is another sibling of the `requiredTrustTier` facet member (§2.1) and is orthogonal to it (no formal mapping between trust tier and assurance class). It is part of the locked-tier portable-unlock profile and is specified in EXT-T1 §T1.3.3.

6.7. Post-Quantum Signatures PREVIEW

Post-quantum signature migration (candidate algorithms ML-DSA, SLH-DSA, FN-DSA, and a dual-signature migration path carrying a `postQuantumSignature` alongside the classical signature) is forward-looking cryptography for high-assurance deployments. It is specified in EXT-T2. The Base reserves the behavior at naming altitude: a `postQuantumSignature`, when present, is excluded from the signing input (§1.6.3) and is safely ignored by an evaluator that does not implement the profile (§1.6).

6.8. Category Trust and `trustWeight` PREVIEW

The typed `trustWeight` field (replacing the deprecated `interpretedAs: "hint-only"` enum) and the category-trust claim split (operator identity, service-category claim, category attestation, urgency claim) are specified in EXT-OPT. The Base records only that `interpretedAs` is removed

and replaced by `trustWeight` (a v0.3 manifest carrying `interpretedAs` degrades safely, with `trustWeight` defaulting to `"hint"`).

6.9. Cryptographic Requirements Summary PREVIEW

The consolidated mandatory-to-implement / recommended / optional / future table of all algorithms and cryptosuites referenced across the document set is maintained in EXT-OPT, so it can be updated as profiles are added without editing the Base. The Base's own mandatory-to-implement floor is: **Ed25519 + JCS** for signatures (§1.6) and **`ECDH-ES+A256KW` / `A256GCM`** for encrypted facets (§2.4.1).

6.10. Context Integrity

Because the meaning of every term in a manifest is fixed by its `@context` (§1.2.1), an evaluator MUST resolve the declared version namespace to a **pinned, content-hashed context document** rather than fetching an arbitrary remote context at evaluation time. The normative context document for this version, with its content hash, is reproduced in Appendix B. An evaluator MUST NOT alter term semantics based on an unpinned or substituted context, and MUST reject a manifest whose `@context` cannot be matched to a context it trusts. (Context-integrity *production* details — how the hash is computed and carried across encodings — are in EXT-OPT.)

✧ ✧ ✧

7. Privacy Considerations

Universal Manifest is built so that a holder reveals only what an interaction requires, and so that the act of revealing does not become a durable tracking signal. The baseline privacy model rests on several properties already normative in this document:

- **Opaque, rotating identifiers.** The manifest `@id` MUST be opaque and free of PII (§1.2.2) and SHOULD be rotated per issuance (§3.2), so manifest instances are not trivially linkable.
- **Holder-controlled minimal disclosure.** Disclosure is holder-controlled and projection-scoped (§3.1.3): a holder includes only the facets an interaction needs, and a withheld facet is *not projected*, never evidence of absence.
- **Sealed entries.** Encrypted facets (§2.3) let a holder carry data that only designated recipients can read; every other evaluator records the facet as present-but-sealed and MUST NOT infer

its content.

- **The privacy–binding tension.** Stronger identity binding (Tier 1+) improves assurance but can increase correlatability. Holders SHOULD prefer the least-linkable binding adequate to the interaction, and SHOULD omit long-lived correlators (such as a hardware `deviceAttestation`) from session-only or pseudonymous manifests (§1.4.6). EXT-T2 specifies a BBS+ selective-disclosure track as a PREVIEW. It removes one repeated value, the issuer's signature, and does not establish unlinkability in this version.
- **Consent as a privacy control.** The Consent stage (§3.1.4) is itself a privacy mechanism: facet data may be used only within an explicit, in-window, scope-and-purpose-matched grant, and purpose binding prevents silent re-use of data across purposes.

On top of those properties, the following baseline privacy obligations apply to any conformant deployment:

- **Sealed-entry forward secrecy.** Recipient revocation (§2.3.4) governs *future* issuances only: re-encrypting on revocation cannot un-share copies already distributed, and 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` (§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 (EXT-OPT), by spreading queries across federated resolvers (EXT-OPT), and by tolerating offline operation rather than blocking on a live status fetch. These privacy measures remain subject to the authority, target-binding, and current-permission limits in EXT-OPT O2.1; neither caching nor offline operation permits a required check to be treated as satisfied.
- **Receipt minimization.** Receipts (§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 (EXT-OPT) to avoid creating a long-lived correlator across unrelated interactions.
- **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.

7.1. Selective Minimum Disclosure: Bounded Base Profile

Selective minimum disclosure means each viewer, verifier, system, or use case receives only the minimum information it is allowed or needs to see. In v0.5, the bounded Base profile is the combination of holder-controlled projection (§3.1.3), consent gating (§3.1.4), sealed-entry handling (§2.3), and honest receipt reporting (§3.3). A holder builds a manifest instance for the interaction; an evaluator processes only the projected items; and unprojected facets are not evidence of absence.

This bounded profile is sufficient to state the core Universal Manifest privacy requirement in v0.5 without pretending the harder privacy properties are solved. It does **not** guarantee verifier-verifier unlinkability, issuer-verifier unlinkability, cover-slot indistinguishability, resolver-operator privacy, or resistance to all residual fingerprinting. No v0.5 conformance claim establishes those stronger properties. The higher-privacy tracks below are PREVIEW and establish none of them in this version.

High-privacy mode limits. High-privacy mode is an unresolved draft in this version, which selects no such mechanism. In the draft, a presentation shows a new holder key for each check in place of a stable holder. The mode does not make a holder unlinkable, and this version gives no measured result that it hides a holder or keeps two checks apart. A deployment that offers such a mode:

- MUST use it for one-off checks only.
- MUST refuse it for delegated use and for long-lived use, and MUST state that those uses are linkable.
- MUST state that the proof shows only that someone knows the credential's secret. It does not show who. The draft gives a verifier no way to detect or limit lending.
- MUST NOT promise the mode for a check where lending defeats the purpose, such as an age check. This version offers no age check for real use.

Values a verifier sets can link one holder's visits to that verifier, and a holder cannot make those values fresh.

7.1.1. Selective Disclosure: Dual-Track Model PREVIEW

Manifest-level selective disclosure is offered along two tracks: **Track A (baseline)** — SD-JWT-style selective disclosure, the interoperable default — and **Track B** — BBS+ derived proofs for privacy-critical deployments, which remove one repeated value, the issuer's signature. Other carried values can still link presentations, so Track B alone does not prevent verifier-to-verifier correlation.

The Base establishes Track A as the baseline selective-disclosure behavior consistent with holder-controlled projection above; the full dual-track model, and Track B's derived-proof mechanics, are specified in EXT-T2.

7.2. Private-Data Handling: Derived Writes and Honest Limits

PREVIEW

Terminology. A *sensitivity label* ranks how sensitive a facet's content is. A set of labels equipped with a *partial order* (some pairs comparable, some not) forms a lattice fragment; the *join* of two labels is their *least upper bound* — the least-sensitive label that is at least as sensitive as both. A *join-semilattice* is an ordering in which a join is always defined for every pair. To *down-label* is to assign an output a label less sensitive than its sources — the move this section prohibits.

Derived-write monotonicity. This is a new construct analogous in spirit to the most-specific-key down-label prohibition of the derived-variant sensor consent rule (EXT-OPT §O9.1), not a generalization of it. Where a profile supplies a facet sensitivity ordering, an evaluator that produces an output derived from one or more source facets MUST assign that output **at least the join** of the source labels, MUST NOT write a derived output into a lower-sensitivity facet unless it re-gates the write under a consent valid for the output label, and MUST NOT silently down-label; otherwise it discards the output. A profile that supplies a sensitivity ordering MUST supply a join-semilattice (a least upper bound is always defined), **or**, where no join is defined for a pair of labels, the evaluator MUST fail closed — assign the output a label that is an upper bound of all the source labels, or discard the derived write; where the ordering defines no such upper bound (an ordering need not have a top element), the output MUST be discarded. Behavior for incomparable labels is thus defined.

Profile-supplied ordering (supply model). A facet sensitivity ordering is supplied by a **registered profile** (EXT-OPT §O6) — never by the manifest under evaluation. The profile's specification publishes the ordering as two parts: `labels` — the finite set of label strings — and `order` — a set of covering pairs `[a, b]`, each stating that label `a` is ordered strictly below label `b`; the effective ordering is the **reflexive–transitive closure** of those pairs, which MUST be a partial order (the closure admits no cycle) and SHOULD be a join-semilattice (otherwise the fail-closed branch above governs each join-less pair). Deployments implementing this section pin the profile's ordering exactly as they pin the context document (the EXT-OPT §O1.7 discipline): resolve and fix it at configuration time from the profile's versioned registry entry, never fetch it from an untrusted resolver at evaluation time — offline operation is preserved. **Label binding:** facets bind to labels by a mapping defined by the profile or by the deployment configuration operating under it (for example, by facet `name`, by entity `@type`, or by an explicit per-deployment table); no v0.5 facet

member carries a label — a per-facet label member is a wire-freeze decision. A source facet the mapping assigns no label yields no join for any pair that includes it, so a derivation reading it fails closed under the rule above. **Trust posture:** the ordering polices the producer's own derived-write behavior, and an artifact MUST NOT supply the lattice that polices it — producers and evaluators MUST NOT source the operative ordering from any member of the manifest under evaluation.

Precedence with the sensor specificity rule. When a sensor-derived write is governed by *both* the most-specific-key specificity rule (EXT-OPT §O9.1) and this join rule, they compose and the write MUST satisfy both: the join governs the *output label* assigned to the written facet, while the specificity rule governs *which consent key authorizes the source sensor read* that feeds the derivation. The two rules act on different objects (output label vs. source-read authorization), so there is no precedence conflict.

Agent-derived memory. When a profile designates a facet as durable agent-memory, a store of agent-derived memory retained across sessions is a **facet** under Universal Manifest, not a separate primitive, and is governed by consent like any other facet. A write into it is a derived write and MUST carry at least the join of the labels of the source facets that contributed to it (for a sensor-derived write, the precedence rule above applies). Reading the memory facet is an ordinary consent-gated read of *that* facet at *its* recorded write-time-join label and confers **no implicit grant** on any source facet. Memory content MUST NOT be used to satisfy or bypass any source facet's consent or floor, and **re-derivation from memory is itself a derived operation** subject to this section. Creating the memory facet by writing is authorized through the pre-declared write-consent mechanism (§3.1.4).

Receipt-as-derived-output. A receipt is itself a derived output. This section and Receipt Minimization (above) apply to receipts: `subjectRef`/`reason` MUST carry only opaque identifiers, never source-facet content, and a receipt's effective sensitivity is the join of the facets it records. This closes the receipt-chain laundering vector. Forwarding re-presents signed manifest bytes and adds no derivation; a derived output retains the label assigned at write time, and later forwarding does not launder that label (consistent with Manifest Forwarding, EXT-OPT).

Honest limits (residual risk — bounded and audited, never eliminated).

- **Off-by-default derived-write protection.** Derived-write monotonicity is a producer-internal behavior with **no wire-checkable evidence**, and **absent a profile-supplied ordering there is NO protection** (the protection is off by default). A consuming evaluator cannot verify from manifest bytes that a producer actually applied the join.
- **Consent withdrawal is forward-only.** Consent withdrawal (§1.4.4) governs only *future* processing of the governed facet. It does not relabel, revoke, or erase outputs already derived under a then-valid consent, nor already-written agent memory.

- **Shred is assertion, not erasure.** A `facet-key-shredded` event (EXT-OPT §O3.2) is the emitter's assertion and timing of key destruction, not cryptographic proof copies are gone; it is not retroactive revocation.
- **In-use plaintext.** Once a facet is decrypted for an authorized use, the plaintext is in use; no manifest mechanism eliminates in-use plaintext risk. An unlock window bounds the authorization window, not in-use plaintext (EXT-OPT).
- **Unverifiable key isolation.** Distinct `kid`/`iv` are necessary but not sufficient for true key isolation; true isolation depends on deployment key custody that is not attestable from manifest bytes (§2.3.5).

✧ ✧ ✧

8. Federation PREVIEW

Federation — resolver coordination, status-check distribution, cache invalidation, availability and failover, and manifest forwarding across multiple resolver operators — is a protocol-layer infrastructure concern that evolves independently of the envelope. It is specified in EXT-OPT. The Base depends on it only at naming altitude: a `signature.statusRef` (§1.6.2) may resolve through a federated resolver, but a Tier-0/Tier-1 evaluator treats status resolution as the single-endpoint protocol referenced in §3.4.

✧ ✧ ✧

Acknowledgements

The Universal Manifest builds on the Web Publication Manifest and Web Application Manifest traditions, the W3C Decentralized Identifiers and Verifiable Credentials work, and the IETF JOSE/COSE and SD-JWT efforts. The editors thank the reviewers and integration-profile contributors.

9. References

This Base cites the references required to implement the Tier-0/Tier-1 surface. Extension-only references travel with their extension.

9.1. Normative References

- **[RFC 2119]** Key words for use in RFCs. https://www.rfc-editor.org/rfc/rfc2119
- **[RFC 8174]** Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words. https://www.rfc-editor.org/rfc/rfc8174
- **[RFC 3339]** Date and Time on the Internet: Timestamps. https://www.rfc-editor.org/rfc/rfc3339
- **[RFC 8032]** Edwards-Curve Digital Signature Algorithm (EdDSA). https://www.rfc-editor.org/rfc/rfc8032
- **[RFC 8785]** JSON Canonicalization Scheme (JCS). https://www.rfc-editor.org/rfc/rfc8785
- **[RFC 7493]** The I-JSON Message Format. https://www.rfc-editor.org/rfc/rfc7493
- **[RFC 4648]** The Base16, Base32, and Base64 Data Encodings. https://www.rfc-editor.org/rfc/rfc4648
- **[RFC 7516]** JSON Web Encryption (JWE). https://www.rfc-editor.org/rfc/rfc7516
- **[JSON-LD11]** JSON-LD 1.1. https://www.w3.org/TR/json-ld11/
- **[DID-CORE]** Decentralized Identifiers (DIDs) v1.0. https://www.w3.org/TR/did-core/

9.2. Informative References

- **[DPV]** W3C Data Privacy Vocabulary. https://w3c.github.io/dpv/
- **[VC-DATA-MODEL]** Verifiable Credentials Data Model v2.0. https://www.w3.org/TR/vc-data-model-2.0/

Additional references (BBS+, SD-JWT, FROST, ML-DSA/SLH-DSA/FN-DSA, MULTIFORMATS, OPENXR, WEBAUTHN, TPM2, and others) are cited where used in EXT-T1, EXT-T2, EXT-T3, and EXT-OPT.

10. IANA Considerations

10.1. Media Type Registrations

This specification registers the media type `application/um+ld+json` for the JSON-LD reference encoding of a Universal Manifest, with the versioned context namespace as the controlling vocabulary; the generic `application/ld+json` remains an acceptable alias (EXT-OPT §O1.3).

The compact-encoding media type for CBOR-LD is registered with the CBOR-LD production rule in EXT-OPT.

(Note: the `+ld+json` multiple-suffix form depends on IETF media-type suffix policy that is not yet settled — the W3C VC WG registered `application/vc` for this reason. If registration of `application/um+ld+json` is refused, the fallback identifier is `application/um`, with `application/ld+json` retained as the alias; this note flags a registrability risk in the suffix form only and does not reopen the namespace decision above. The full RFC 6838 registration template — type name, subtype, required/optional parameters, encoding, security considerations, contact, change controller — is to be added as an appendix before CR-equivalent maturity.)

10.2. Registry Change Control

Change control for Universal Manifest media types and for the profile registry rests with the maintainers of the canonical `UniversalManifest/universal-manifest` repository, through its public process, per the profile registration mechanism (EXT-OPT).

✧ ✧ ✧

Appendix B. Versioned Context Document (Normative Reference)

The full term definitions for v0.5 are fixed by the versioned context document identified by the namespace URI `https://universalmanifest.net/ns/v0.5` (and served there; the same normative document is `spec/v0.5/schema.jsonld` in the specification repository). That document, together with its content hash, is the **normative reference for context integrity** (§6.10): an evaluator pins this content-hashed copy rather than fetching an arbitrary remote context at evaluation time. The context document defines the `um:` term set (`um:Manifest`, `um:Facet`, `um:Entity`, `um:Consent`, `um:Receipt`, `um:Device`, the `um:reason:` and `um:profile:` registries, and the structural member terms) over the base IRI `https://universalmanifest.net/ns/um#`.

The versioned namespace URI identifies an **immutable document**, not a mutable network resource (EXT-OPT). Its normative body is the JSON-LD context document reproduced verbatim below; the same document is carried unchanged as the machine-readable artifact `schema.jsonld` in the v0.5 artifact directory (`spec/v0.5/schema.jsonld` in the specification repository, also served at `https://universalmanifest.net/ns/universal-manifest/v0.5/schema.jsonld`). The reproduction below, that artifact, and any copy retrieved from the namespace URI are the

same byte sequence: UTF-8 text with LF (U+000A) line endings and a single trailing LF. Trust in the term definitions derives from the content hash, never from the retrieval channel.

The content hash of the v0.5 context document is the SHA-256 multihash computed over exactly those bytes, per the computation rule in EXT-OPT:

- Multibase base58btc: `zQmR1W9vsPrdbXmhH38jdNhdEhBB1C7JUL1Z9kZMi6u16i7`
- SHA-256: `27ab5a3cc491cee7a5947b73f550c07ecbff285a81a6ba801aa98f60046dcea4`
- Multihash bytes (hex): `122027ab5a3cc491cee7a5947b73f550c07ecbff285a81a6ba801aa98f60046dcea4`

Implementations MUST verify this hash before trusting the term definitions; a manifest whose `@context` cannot be matched to a context the evaluator trusts MUST be rejected (§6.10).

The v0.5 URI intentionally identifies the same exact context bytes as the accepted local v0.4 editorial candidate: its definitions use the version-independent `um:` IRI and contain no v0.4 namespace URI requiring replacement. Equal content pins do not make wire `0.4` and `0.5` documents interchangeable; the signed envelope carries a different namespace and version. Published v0.4 context bytes and their old pin remain preserved, as described in migration.

The verbatim body of the versioned context document:

```
{
  "@context": {
    "@version": 1.1,
    "um": "https://universalmanifest.net/ns/um#",
    "schema": "https://schema.org/",
    "xsd": "http://www.w3.org/2001/XMLSchema#",
    "id": "@id",
    "type": "@type",
    "Manifest": "um:Manifest",
    "Facet": "um:Facet",
    "Entity": "um:Entity",
    "Consent": "um:Consent",
    "Receipt": "um:Receipt",
    "Device": "um:Device",
    "BilateralSession": "um:BilateralSession",
    "agentDelegation": "um:agentDelegation",
    "manifestVersion": "um:manifestVersion",
    "subject": {
      "@id": "um:subject",
```

```
      "@type": "@id"
    },
    "issuedAt": {
      "@id": "um:issuedAt",
      "@type": "xsd:dateTime"
    },
    "expiresAt": {
      "@id": "um:expiresAt",
      "@type": "xsd:dateTime"
    },
    "requiredTrustTier": {
      "@id": "um:requiredTrustTier",
      "@type": "xsd:integer"
    },
    "requiredLiveness": {
      "@id": "um:requiredLiveness"
    },
    "minFreshness": "um:minFreshness",
    "userVerified": {
      "@id": "um:userVerified",
      "@type": "xsd:boolean"
    },
    "requiredAssuranceClass": "um:requiredAssuranceClass",
    "facets": {
      "@id": "um:facets",
      "@container": "@set"
    },
    "claims": {
      "@id": "um:claims",
      "@container": "@set"
    },
    "consents": {
      "@id": "um:consents",
      "@container": "@set"
    },
    "devices": {
      "@id": "um:devices",
      "@container": "@set"
    },
    "pointers": {
      "@id": "um:pointers",
      "@container": "@set"
    },
```

```
    "actorState": {
      "@id": "um:actorState"
    },
    "name": "schema:name",
    "description": "schema:description",
    "ref": {
      "@id": "um:ref",
      "@type": "@id"
    },
    "entity": {
      "@id": "um:entity"
    },
    "encryptionProfile": "um:encryptionProfile",
    "protected": "um:jweProtected",
    "recipients": {
      "@id": "um:jweRecipients",
      "@container": "@set"
    },
    "header": "um:jweHeader",
    "kid": "um:jweKid",
    "encrypted_key": "um:jweEncryptedKey",
    "iv": "um:jweIv",
    "ciphertext": "um:jweCiphertext",
    "tag": "um:jweTag",
    "previousKid": "um:jwePreviousKid",
    "rotationReason": "um:jweRotationReason",
    "revokedRecipientKid": "um:jweRevokedRecipientKid",
    "revocationAction": "um:jweRevocationAction",
    "issuer": {
      "@id": "um:issuer",
      "@type": "@id"
    },
    "claimProof": {
      "@id": "um:claimProof"
    },
    "boundDids": {
      "@id": "um:boundDids",
      "@container": "@set"
    },
    "attester": {
      "@id": "um:attester",
      "@type": "@id"
    },
```

```
    "attestationMethod": "um:attestationMethod",
    "attestedAt": {
      "@id": "um:attestedAt",
      "@type": "xsd:dateTime"
    },
    "facetRef": {
      "@id": "um:facetRef",
      "@type": "@id"
    },
    "scope": {
      "@id": "um:scope",
      "@container": "@set"
    },
    "purpose": "um:purpose",
    "grantedAt": {
      "@id": "um:grantedAt",
      "@type": "xsd:dateTime"
    },
    "grantor": {
      "@id": "um:grantor",
      "@type": "@id"
    },
    "withdrawnAt": {
      "@id": "um:withdrawnAt",
      "@type": "xsd:dateTime"
    },
    "conditions": {
      "@id": "um:conditions",
      "@container": "@set"
    },
    "unlockWindowFacets": {
      "@id": "um:unlockWindowFacets",
      "@container": "@set"
    },
    "idleLifetime": {
      "@id": "um:idleLifetime",
      "@type": "xsd:duration"
    },
    "absoluteLifetime": {
      "@id": "um:absoluteLifetime",
      "@type": "xsd:duration"
    },
    "lockOnSleep": {
```

```
      "@id": "um:lockOnSleep",
      "@type": "xsd:boolean"
    },
    "target": {
      "@id": "um:target",
      "@type": "@id"
    },
    "label": "um:label",
    "createdAt": {
      "@id": "um:createdAt",
      "@type": "xsd:dateTime"
    },
    "delegateType": "um:delegateType",
    "delegatedBy": {
      "@id": "um:delegatedBy",
      "@type": "@id"
    },
    "delegatedAt": {
      "@id": "um:delegatedAt",
      "@type": "xsd:dateTime"
    },
    "delegateId": {
      "@id": "um:delegateId",
      "@type": "@id"
    },
    "livenessEndpoint": {
      "@id": "um:livenessEndpoint",
      "@type": "@id"
    },
    "holderBinding": {
      "@id": "um:holderBinding"
    },
    "mode": "um:bindingMode",
    "cnfThumbprint": "um:cnfThumbprint",
    "commitment": "um:commitment",
    "pseudonymScope": "um:pseudonymScope",
    "boundDid": {
      "@id": "um:boundDid",
      "@type": "@id"
    },
    "subjectProof": "um:subjectProof",
    "boundDidProof": "um:boundDidProof",
    "bindingProof": {
```

```
      "@id": "um:bindingProof"
    },
    "cryptosuite": "um:cryptosuite",
    "proofSystem": "um:proofSystem",
    "circuit": "um:circuit",
    "publicInputs": {
      "@id": "um:publicInputs"
    },
    "commitmentA": "um:commitmentA",
    "commitmentB": "um:commitmentB",
    "publicKeyA": "um:publicKeyA",
    "publicKeyB": "um:publicKeyB",
    "derivationPathA": "um:derivationPathA",
    "derivationPathB": "um:derivationPathB",
    "ceremonyProof": {
      "@id": "um:ceremonyProof"
    },
    "threshold": "um:threshold",
    "attesters": {
      "@id": "um:attesters",
      "@container": "@set"
    },
    "ceremonyId": {
      "@id": "um:ceremonyId",
      "@type": "@id"
    },
    "aggregateProof": "um:aggregateProof",
    "presentationProof": {
      "@id": "um:presentationProof"
    },
    "proofType": "um:proofType",
    "challenge": "um:challenge",
    "audience": {
      "@id": "um:audience",
      "@type": "@id"
    },
    "proofPurpose": "um:proofPurpose",
    "proofValue": "um:proofValue",
    "verificationMethod": {
      "@id": "um:verificationMethod",
      "@type": "@id"
    },
    "livenessAttestation": {
```

```
      "@id": "um:livenessAttestation"
    },
    "validUntil": {
      "@id": "um:validUntil",
      "@type": "xsd:dateTime"
    },
    "method": "um:method",
    "signature": {
      "@id": "um:signature"
    },
    "algorithm": "um:algorithm",
    "canonicalization": "um:canonicalization",
    "keyRef": {
      "@id": "um:keyRef",
      "@type": "@id"
    },
    "publicKeySpkiB64": "um:publicKeySpkiB64",
    "created": {
      "@id": "um:created",
      "@type": "xsd:dateTime"
    },
    "value": "um:value",
    "statusRef": {
      "@id": "um:statusRef",
      "@type": "@id"
    },
    "revocationCursor": "um:revocationCursor",
    "manifestId": {
      "@id": "um:manifestId",
      "@type": "@id"
    },
    "outcome": "um:outcome",
    "signatureCheck": "um:signatureCheck",
    "freshnessCheck": "um:freshnessCheck",
    "revocationStatus": "um:revocationStatus",
    "revocationReason": "um:revocationReason",
    "keyRefResolution": "um:keyRefResolution",
    "facetStatuses": {
      "@id": "um:facetStatuses",
      "@container": "@set"
    },
    "facetId": {
      "@id": "um:facetId",
```

```
      "@type": "@id"
    },
    "status": "um:status",
    "reason": "um:reason",
    "consentStatuses": {
      "@id": "um:consentStatuses",
      "@container": "@set"
    },
    "consentRef": {
      "@id": "um:consentRef",
      "@type": "@id"
    },
    "claimStatuses": {
      "@id": "um:claimStatuses",
      "@container": "@set"
    },
    "claimRef": {
      "@id": "um:claimRef",
      "@type": "@id"
    },
    "tier": {
      "@id": "um:tier",
      "@type": "xsd:integer"
    },
    "effectiveTrustTier": {
      "@id": "um:effectiveTrustTier",
      "@type": "xsd:integer"
    },
    "holderBindingStatus": "um:holderBindingStatus",
    "presentationProofStatus": "um:presentationProofStatus",
    "livenessStatus": "um:livenessStatus",
    "crossDidBindingStatus": "um:crossDidBindingStatus",
    "assuranceStatus": {
      "@id": "um:assuranceStatus"
    },
    "assertedClass": "um:assertedClass",
    "met": {
      "@id": "um:met",
      "@type": "xsd:boolean"
    },
    "unprocessedEntries": {
      "@id": "um:unprocessedEntries",
      "@container": "@set"
```

```
    },
    "checkedAt": {
      "@id": "um:checkedAt",
      "@type": "xsd:dateTime"
    },
    "processedAt": {
      "@id": "um:processedAt",
      "@type": "xsd:dateTime"
    },
    "warnings": {
      "@id": "um:warnings",
      "@container": "@set"
    },
    "code": "um:code",
    "message": "um:message",
    "receiptId": {
      "@id": "um:receiptId",
      "@type": "@id"
    },
    "evaluatorId": {
      "@id": "um:evaluatorId",
      "@type": "@id"
    },
    "exchangeId": {
      "@id": "um:exchangeId",
      "@type": "@id"
    },
    "seq": {
      "@id": "um:seq",
      "@type": "xsd:integer"
    },
    "prevHash": "um:prevHash",
    "chainId": {
      "@id": "um:chainId",
      "@type": "@id"
    },
    "events": {
      "@id": "um:events",
      "@container": "@set"
    },
    "eventType": "um:eventType",
    "at": {
      "@id": "um:at",
```

```
      "@type": "xsd:dateTime"
    },
    "subjectRef": {
      "@id": "um:subjectRef",
      "@type": "@id"
    },
    "facetKeyRef": {
      "@id": "um:facetKeyRef",
      "@type": "@id"
    },
    "receiptSignature": {
      "@id": "um:receiptSignature"
    },
    "sessionId": {
      "@id": "um:sessionId",
      "@type": "@id"
    },
    "participants": {
      "@id": "um:participants",
      "@container": "@set"
    },
    "did": {
      "@id": "um:did",
      "@type": "@id"
    },
    "role": "um:role",
    "initiatedAt": {
      "@id": "um:initiatedAt",
      "@type": "xsd:dateTime"
    },
    "state": "um:sessionState",
    "updatedAt": {
      "@id": "um:updatedAt",
      "@type": "xsd:dateTime"
    },
    "cursor": "um:cursor",
    "nextCheck": {
      "@id": "um:nextCheck",
      "@type": "xsd:duration"
    }
  }
}
```

B.1. Term-Inclusion Rule

Which members hold a term in this versioned context is governed by a stated rule, not by accumulation:

1. **Production-candidate wire surface.** The context defines a term for every member of the production-candidate surface: the Base envelope, facet, claim, consent, pointer, and device members; the signature members; and every field of the receipt field surface (§3.3). Receipts are first-class manifests, so receipt fields are wire members — this class includes the credential-binding status fields (`holderBindingStatus`, `presentationProofStatus`, `livenessStatus`, `crossDidBindingStatus`), `revocationReason`, the warnings entry members `code` and `message`, and the holder-binding mode members (including the `bbs-holder-commitment` mode's `commitment`).

2. **Wire-exercised PREVIEW members.** A member belonging to a PREVIEW feature holds a term when inherited v0.4 editorial wire artifacts already carry it — a conformance fixture, the behavioral suite, or a Cookbook worked example — so that those artifacts expand losslessly under this context. A term in this class pins the member's *expansion*, not the feature's *maturity*: the feature stays PREVIEW and revisable. This is why `unlockWindowFacets` (PREVIEW unlock windows, carried by the `consent-unlock-window` and `manifest-locked-facet-below-floor` fixtures), the per-facet floor members (`requiredLiveness`, `minFreshness`, `userVerified`, `requiredAssuranceClass`, and the `assuranceStatus` echo members), the key-lifecycle members (`events`, `eventType`, `facetKeyRef`), the Tier-2 and Tier-3 proof members, the bilateral-session members, and the statusRef response members hold terms today.

3. **Design-stage members stay out.** Members of design-stage candidate profiles (EXT-OPT Sections O2.6, O12, O13, O14) and PREVIEW members whose designs are not yet wire-frozen (for example `transparencyAnchor` and its sub-members, `sessionKeyAuthorization`, the `actorState` and device sub-members, `postQuantumSignature`) enter the context only at their wire freeze, together with their fixtures and schema coverage (§4.4).

The transitional carriages that previously sat outside these classes have been resolved in the accepted local v0.4 editorial candidate at its wire freeze and are recorded here rather than silently dropped. First, the manifest-carried sensitivity-ordering member set (`facetSensitivityOrdering`, `profile`, `labels`, `order`) **came off the wire** at the wire freeze under decision UM-v04-DEC-03: the ordering is profile-supplied — it comes from a registered profile the deployment pins, not from the manifest under evaluation — so the member set, its fixture, its conformance listing, and its guard entry were removed together in one coordinated cleanup and the content hash rolled once. The derived-write join rule (§7.2) remains normative and unchanged; only the carrier moved.

Second, of the two terms seeded with their structural definitions ahead of wire exercise, `trustWeight` (EXT-OPT Section O8) **came off the wire** at the same freeze — it is exercised by no fixture, example, or worked example, and its value shape (closed enum versus extensible vocabulary) is an open question, so under the third class above it re-enters only with fixture and schema coverage once that question is settled — while the `actorState` container term (§1.4.7) **stays**, having since gained a Cookbook worked example (agent executor) and so qualifying under the second class above. Any term addition or removal changes the context bytes and therefore the content hash, so changes are batched and the hash rolls once per batch.

(Appendix A — Manifest Class Registry Snapshot — is informative and PREVIEW; it is relocated to EXT-OPT. Appendix C — Complete Worked Example — is informative and relocated to the Cookbook.)



End of Base. To implement above Tier 0, continue with EXT-T1. For concrete payloads, see the Cookbook.