# Universal Manifest v0.3 Specification

## Published Specification

## 19 May 2026


**This version:** https://universalmanifest.net/spec/v0.3/

**Latest published version:** https://universalmanifest.net/spec/v0.3/

**Previous version:** https://universalmanifest.net/spec/v0.2/

**History:** Changelog



**Editors:** Universal Manifest Working Group



Copyright © 2026 the Contributors to the Universal Manifest Specification.


**Coming from v0.1?** See the v0.1-to-v0.2 and v0.2-to-v0.3 migration paths in Changes from v0.2.

**Coming from v0.2?** See Changes from v0.2 for a summary of additions in v0.3: evaluation sequence, encrypted facets, structured receipts, and normative promotions.

## Abstract


This specification defines a JSON-LD-based file format known as the **Universal Manifest**, a portable state capsule. Formulated as a hybrid synthesis of web publication metadata and web application parameters, this format provides developers and holders with a standardized envelope to convey linked-data identity references, role permissions, device registrations, and pointers to canonical data sources.

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


## Conformance Requirements

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as

described in BCP 14 [RFC 2119] [RFC 8174] when, and only when, they appear in ALL CAPITALS, as shown here.

## Status of This Document

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

Universal Manifest v0.3 is the current published version of the Universal Manifest specification. It builds on the v0.1 base format (core JSON-LD envelope, lifecycle, caching) and v0.2 extensions (normative signature profile, identity binding framework, tiered trust model, `claims[].claimProof`). Version 0.3 introduces a normative evaluation contract (the six-stage evaluation sequence), encrypted inline facets (JWE profile), structured receipts, and promotes cross-DID binding, `requiredTrustTier`, and agent delegation from non-normative conventions to normative requirements.

Because Universal Manifest is still in the v0.x line, minor-version bumps may still include breaking changes when those changes are reflected by a new version folder. Implementors should review the v0.1 to v0.2 migration guide and the Changes from v0.2 section before upgrading.

Future changes are managed through the RFC mechanism and the breaking-change policy. This document uses layout formats established by major web standard groups to present normative requirements cleanly, and may be cited as the published Universal Manifest v0.3 specification.

**For implementers:** use the Implementation Guide, run the Standalone Conformance Suite, review Conformance v0.3, and compare against the TypeScript reference implementation.

## Changes from v0.2

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

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

- **Cross-DID binding promoted** (Section 6.4.4): Promoted from non-normative convention to normative requirement.
- **`requiredTrustTier` promoted** (Section 6.4.5): Promoted from non-normative convention to normative requirement.
- **Agent delegation promoted** (Section 6.5): Promoted from non-normative convention to normative requirement.
- **Tier 2 defined** (Section 6.4.2): Tier 2 (cryptographic binding via zero-knowledge proof of cross-DID control) is now normatively defined. Marked at risk.
- **Evaluator behavior expanded** (Section 4.1): Added evaluation contract obligations referencing the pipeline.
- **Privacy model expanded** (Section 7): Added sealed-entry and selective-disclosure privacy considerations for encrypted facets.
- **Signature limitations softened** (Section 6.1): Reduced v0.1 caveat language to reflect Signature Profile A delivery.
- **Signature integrity updated** (Section 1.5): Removed "reserved for subsequent iterations" language.
- **Informative references** (Section 8.2): Added URLs to all citations.

# 1. Universal Manifest

A **Universal Manifest** is a JSON-LD document acting as a cross-platform data envelope. It blends the semantic linkability of a Web Publication Manifest with the applied processing lifecycle of a Web App Manifest.

## 1.1. Examples

The following is an example of a minimal Universal Manifest:

**Example 1: Minimal Universal Manifest payload**

```
{
  "@context": [
    "https://universalmanifest.net/ns/v0.3"
  ],
  "@id": "urn:uuid:123e4567-e89b-12d3-a456-426614174000",
  "@type": ["um:Manifest"],
  "manifestVersion": "0.3",
  "subject": "did:example:123",
  "issuedAt": "2026-03-01T00:00:00Z",
  "expiresAt": "2026-03-02T00:00:00Z",
  "signature": {
    "algorithm": "Ed25519",
    "canonicalization": "JCS-RFC8785",
    "keyRef": "did:key:z6MkExample#keys-1",
    "publicKeySpkiB64": "MCowBQYDK2VwAyEA...",
    "created": "2026-03-01T00:00:00Z",
    "value": "base64url-encoded-signature-bytes"
  }
}
```

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

## 1.2. JSON-LD Core Members

### 1.2.1. `@context` member

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/v0.3` for this version).

> **Note:** The namespace URI is versioned. Manifests conforming to this specification **MUST** use `https://universalmanifest.net/ns/v0.3`. Evaluators encountering earlier namespace versions (e.g., `v0.1`, `v0.2`) **SHOULD** process them according to the referenced version's rules.

#### 1.2.2. `@id` member

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

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 v0.3 conformant manifests, the value **MUST** be `"0.3"`. Evaluators **MUST** check that they support the declared version before processing the manifest through the evaluation sequence.

#### 1.3.2. `subject` member

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

#### 1.3.3. `issuedAt` and `expiresAt` members

Both `issuedAt` and `expiresAt` are **REQUIRED**. They formulate the bounding constraints (TTL) for the manifest's validity. Both parameters are formatted as ISO 8601 / RFC 3339 date-time strings.

### 1.4. Structural State Members

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

#### 1.4.1. `facets` member

The `facets` member organizes extended functional blocks (`um:Facet`), packaging specific verifiable capabilities, metadata subsets, or configuration modules. The `facets` member, when present, **MUST** be a JSON array of facet objects.

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

These arrays group specific operational contexts representing permissions, deployed hardware targets, and external data reference pointers connected to the `subject`. Each array is a top-level structural member of the manifest. The following subsections define the base schema for each array's entries.

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

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

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

#### 1.4.3. `claims` Array Schema

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

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

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

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

- `subject` — DID or URI of the entity the claim is about. When absent, the manifest-level `subject` is implied.
- `issuedAt` — ISO 8601 date-time when the claim was issued.
- `expiresAt` — ISO 8601 date-time when the claim expires. Evaluators **SHOULD** reject expired claims.
- `claimProof` — A Verifiable Presentation, attestation proof object, URI reference, or array of proof entries enabling Tier 1 verification (see Section 6.4.3). When an array, each entry is independently verifiable and **MAY** carry its own `proofType` and `proofPurpose` fields.
- `requiredTrustTier` — Integer (0–3) declaring the minimum trust tier for this claim (see Section 6.4.5).

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

**Example 8: Base claim object**

```
{
  "claims": [
    {
      "@type": "membership.organization",
      "issuer": "did:web:example-org.com",
      "subject": "did:key:z6MkAlice",
      "issuedAt": "2026-03-01T00:00:00Z",
      "expiresAt": "2027-03-01T00:00:00Z"
    },
    {
      "@type": "identity.crossDidBinding",
      "issuer": "did:web:verify.example",
      "boundDids": ["did:key:z6MkAlice", "did:plc:alice-bsky"],
      "attester": "did:web:verify.example",
      "attestationMethod": "AT Protocol handle resolution",
      "attestedAt": "2026-03-15T10:30:00Z",
      "claimProof": [
        {
          "proofType": "VerifiablePresentation",
          "proofPurpose": "assertionMethod",
          "@type": "VerifiablePresentation",
          "issuer": "did:web:verify.example",
          "verifiableCredential": ["https://verify.example/attestations/abc1
        }
      ]
    }
  ]
}
```

### 1.4.4. `consents` Array Schema

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

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

- `@id` — A URI uniquely identifying this consent entry. **MUST** be present. Consent entries are identified by `@id` for receipt recording.
- `@type` — **MUST** be `"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. An evaluator determines which consent entry governs a facet by matching `facetRef` to the facet's `@id`.
- `scope` — An array of scope strings defining what operations are authorized (e.g., `"read"`, `"display"`, `"cache"`, `"process"`, `"forward"`). Evaluators **MUST NOT** perform operations outside the declared scope.
- `purpose` — A string declaring the stated purpose for the data use (e.g., `"session-personalization"`, `"age-verification"`, `"access-control"`). Evaluators **SHOULD** verify that their intended use falls within the declared purpose before processing the facet. Purpose matching semantics are deployment-specific; this specification does not define a purpose vocabulary or matching algorithm.
- `grantedAt` — ISO 8601 date-time when the consent was granted.
- `expiresAt` — ISO 8601 date-time when the consent expires. Evaluators **MUST** treat expired consent as absent consent.

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

- `grantor` — DID or URI of the entity who granted consent. When absent, the manifest `subject` is implied.
- `withdrawnAt` — ISO 8601 date-time when consent was withdrawn. When this field is present, the consent is no longer active regardless of `expiresAt`. Evaluators **MUST** treat a consent entry with a `withdrawnAt` value as absent consent.
- `conditions` — An array of condition strings imposing additional constraints (e.g., `"offline-only"`, `"no-third-party-sharing"`).

At most one consent entry **SHOULD** reference a given facet `@id` via `facetRef`. If multiple consent entries reference the same facet, evaluators **MUST** treat them as conjunctive — all matching consent entries must individually pass validation (not expired, not withdrawn, scope and purpose satisfied) for the facet to be considered consented.

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

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

**Example 9: Consent entry**

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

### 1.4.5. `pointers` Array Schema

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

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

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

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

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

- `@id` — An opaque identifier for this pointer entry.
- `label` — A human-readable label describing the pointer's purpose.
- `createdAt` — ISO 8601 date-time when the pointer was created.
- `expiresAt` — ISO 8601 date-time when the pointer expires. Evaluators **SHOULD** ignore expired pointers.

The following pointer types are defined in this specification:

- `um:agentDelegation` — Declares delegated session authority. See Section 6.5.1 for the complete field specification. The `um:agentDelegation` type replaces the base `target` field with type-specific required fields (`delegatedBy`, `delegateType`, `delegatedAt`, `expiresAt`) and optional fields (`delegateId`, `scope`).

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

### 1.4.6. `devices` Array Schema

> **Note (Reserved):** The `devices` array is a reserved structural member. It is intended to register hardware endpoints (e.g., NFC readers, smart displays, wearable sensors) associated with the manifest `subject`. The base schema for device entries is not yet specified. Implementations **MUST** accept and preserve a `devices` array if present but **MUST NOT** assign semantic meaning to its contents until a future version of this specification defines the device entry schema.

When a `devices` array is present, evaluators **MUST** include it in the signing input (it is part of the manifest payload) and **MUST** carry it through the evaluation sequence without modification.

Evaluators **MUST NOT** discard the `devices` array during the Arrive stage's unknown-field handling, because it is a named structural member.

## 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 v0.3 conformant manifest **MUST** include a `signature` member conforming to the JCS + Ed25519 Signature Profile (Signature Profile A, Section 1.6). Future spec versions **MAY** introduce additional normative profiles; evaluators **MUST** reject manifests whose signature profile they do not support.

> **Note:** Unsigned manifests **MAY** exist as development artifacts but are not v0.3 conformant and **MUST** be rejected by conformant evaluators.

## 1.6. Signature Profile A: JCS + Ed25519

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

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

Signature profiles are additive. Future versions **MAY** introduce additional profiles (e.g., post-quantum algorithms or W3C Data Integrity proofs). Evaluators verify the profiles they support and safely ignore unknown profiles.

### 1.6.1. Canonicalization and Algorithm

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

This combination provides deterministic signing input without requiring JSON-LD expansion or RDF canonicalization, matching the specification's "state capsule" usage where the JSON shape is the primary contract.

### 1.6.2. Signature Shape

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

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

The following fields are **OPTIONAL**:

- `signature.publicKeySpkiB64` — base64-encoded SPKI DER public key bytes for offline/fixture/local-first verification.
- `signature.created` — ISO 8601 date-time indicating when the signature was produced.
- `signature.statusRef` — URI to revocation/status material for this signature.
- `signature.revocationCursor` — monotonic status cursor/version string for cache-aware revocation checks.

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

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

**Example 2: Signature Profile A object**

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

### 1.6.3. Signing Input Procedure

To compute the signature for this profile:

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

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

### 1.6.4. Verifier Checklist

A verifier implementing this profile **MUST**:

1. Confirm the document is a v0.x Universal Manifest (required fields present and `@type` includes `um:Manifest`).
2. Enforce TTL: reject for use if `now > expiresAt`; sanity-check `issuedAt <= expiresAt`.
3. Determine signature profile support: require `signature.algorithm === "Ed25519"`, `signature.canonicalization === "JCS-RFC8785"`, and a non-empty `signature.value`.
4. Resolve the verification key. The evaluator **MUST** resolve `keyRef` to obtain the public key material (method-specific, e.g., DID resolution or HTTPS fetch). If `signature.publicKeySpkiB64` is present, the evaluator **MAY** use it for verification but **MUST** confirm that it is consistent with the key resolved via `keyRef` — specifically, the resolved public key material **MUST** be byte-identical to the decoded `publicKeySpkiB64` value. If the values do not match, the evaluator **MUST** reject the manifest with outcome `rejected` and `signatureCheck: "invalid"`. If `publicKeySpkiB64` is absent, the evaluator resolves `keyRef` to obtain the public key material. In offline or local-first environments where `keyRef` cannot be resolved, the evaluator **MAY** use `publicKeySpkiB64` as the sole key source; the consistency check is deferred until connectivity is available. Evaluators operating in this mode **SHOULD** record `"keyResolution": "deferred"` in the receipt.
5. Recompute the signing input (remove `signature`, JCS canonicalize).
6. Verify the Ed25519 signature over the canonical bytes.

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

> **Security Note:** Without the key-matching check in step 4, a malicious holder can bundle a `keyRef` pointing to a high-reputation DID while supplying their own key material in `publicKeySpkiB64`, enabling key substitution attacks. Evaluators that skip `keyRef` resolution when `publicKeySpkiB64` is present are vulnerable to this vector.

#### 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 when strict verification is required by policy.
- Evaluators **MUST NOT** reinterpret unknown pairs as the baseline profile.

#### 1.6.6. Revocation-Aware Verification Extension

For evaluators claiming revocation-aware verification:

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

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

## 2. Entities and Facets

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

## 2.1. um:Facet Module

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

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

A facet object **SHOULD** contain the following field:

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

A facet object **MAY** contain the following fields:

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

> **Note:** 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.

**Example 9: Plaintext facet with all fields**

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

## 2.2. um:Entity Base

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

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

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

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

**Example 10: Plaintext entity**

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

## 2.3. Encrypted Facets (JWE Inline Profile)

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

### 2.3.1. Declaration

To declare an encrypted facet, the facet **MUST** include the field `encryptionProfile` with the value `"jwe-inline-v1"`. When `encryptionProfile` is present, the `entity` field **MUST** contain a JWE

JSON Serialization object instead of a plain `um:Entity`.

### 2.3.2. JWE Structure

The `entity` value for an encrypted facet **MUST** conform to the following structure:

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

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

**Example 3: Facet with JWE inline encryption**

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

### 2.3.3. Key Rotation

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

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

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

#### 2.3.4. Recipient Revocation

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

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

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

## 3. Manifest Lifecycle and Caching

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

### 3.1. Evaluation Sequence

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

#### 3.1.1. Stage 1: Arrive

The manifest is received and its envelope structure becomes visible to the evaluator. The evaluator **MUST** parse the JSON document, confirm the existence of required properties (`@context`, `@id`, `@type`, `manifestVersion`, `subject`, `issuedAt`, `expiresAt`, `signature`), and verify that `@type` includes `um:Manifest` (per Section 1.2.3). The evaluator **MUST** ignore unknown fields for processing purposes but **MUST NOT** remove them from the payload prior to signature verification. The full JSON object (minus only the `signature` property per Section 1.6.3) is the input to JCS canonicalization in the Verify stage. After verification succeeds, unrecognized fields have no normative semantics and **MUST NOT** affect evaluation sequence outcomes. If parsing or structural validation fails, the pipeline terminates with a `rejected` receipt.

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

#### 3.1.2. Stage 2: Verify

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

1. Signature verification against the JCS-canonicalized payload per the declared signature profile (see Section 1.6).
2. Freshness enforcement: reject if `now > expiresAt` or if `issuedAt > expiresAt`.
3. Revocation status resolution, if `signature.statusRef` is present and the evaluator implements revocation-aware verification.

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

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

#### 3.1.3. Stage 3: Project

The evaluator **MUST** extract only the facets, claims, and pointers relevant to its processing context. Selective disclosure is holder-controlled: the manifest holder determines which facets are included in a given manifest instance. The evaluator **MUST NOT** assume that the manifest contains the complete set of the subject's facets. Facets not included in the manifest are not absent — they are not projected for this interaction.

#### 3.1.4. Stage 4: Consent

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

Section 1.4.4). For each facet:

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

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

#### 3.1.5. Stage 5: Compose

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

- `accepted` — all projected facets processed, all consent requirements satisfied, signature valid.
- `accepted-with-warnings` — manifest accepted but one or more non-critical conditions were noted (e.g., revocation status could not be checked).
- `accepted-partial` — some facets were processed; others were rejected, sealed (encrypted and undecryptable), or lacked consent.
- `rejected` — the manifest failed a mandatory check (signature, freshness, structural validity, required consent, or a manifest-level `requiredTrustTier` that the evaluator cannot satisfy).

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

#### 3.1.6. Stage 6: Receipt

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

## 3.2. Caching Formulation

For constrained devices and public displays:

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

## 3.3. Structured Receipts

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

### 3.3.1. Receipt Fields

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

- `@type` — **MUST** include `"um:Receipt"`.
- `manifestId` — the `@id` of the processed manifest.
- `outcome` — one of `"accepted"`, `"accepted-with-warnings"`, `"accepted-partial"`, or `"rejected"`.
- `signatureCheck` — result of signature verification: `"valid"`, `"invalid"`, or `"unsupported-profile"`.
- `freshnessCheck` — result of TTL enforcement: `"fresh"`, `"expired"`, or `"stale"`. The `"stale"` value indicates that the manifest's `issuedAt` is in the future relative to the evaluator's clock (beyond the permitted clock-skew tolerance).

A receipt **MUST** include a `facetStatuses` array when the manifest contains one or more facets. If the manifest contains zero facets, `facetStatuses` **MUST** be an empty array. Each entry in the array is a per-facet status object containing a `facetId` (matching the facet's `@id`), `status` (`"processed"`, `"opaque"`, `"consent-denied"`, `"consent-missing"`, or `"not-projected"`), an **OPTIONAL** `name` (the facet's display label, if present), and an optional `reason` string.

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

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

- `revocationStatus` — `"active"`, `"revoked"`, or `"unchecked"`.
- `consentStatuses` — array of per-consent status objects. Each entry **MUST** contain: `facetId` (the `@id` of the governed facet), `consentRef` (reference to the consent entry), `status` (one of `"valid"`, `"expired"`, `"withdrawn"`, or `"missing"`), and `checkedAt` (ISO 8601 date-time when the consent status was evaluated).
- `processedAt` — ISO 8601 date-time when the receipt was produced.
- `warnings` — array of warning strings for the `accepted-with-warnings` outcome.
- `receiptId` — unique identifier for the receipt (**SHOULD** be an opaque URI, e.g., `urn:uuid:...`).
- `evaluatorId` — DID or identifier of the evaluator that produced the receipt.
- `receiptSignature` — optional signature over the receipt using the evaluator's key, following the same profile as manifest signatures (Signature Profile A).

> **Note:** Receipt signing is **RECOMMENDED** for accountability use cases but is not required for v0.3 conformance. Unsigned receipts are valid evaluation sequence outputs but provide weaker non-repudiation guarantees.

**Example 4: Structured receipt**

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

# 4. Conformance

## 4.1. Evaluator Behavior

An evaluator parsing the manifest **MUST** validate structural parameters and securely ignore unknown elements without raising fatal invocation errors. Freshness (via `expiresAt` TTL constraints) is an absolute rejection gateway. Implementers **MUST** verify `issuedAt <= expiresAt`.

Evaluators claiming v0.3 conformance **MUST** implement the six-stage evaluation sequence defined in Section 3.1. Specifically, a conformant evaluator **MUST**:

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

### 4.2. Holder Behavior

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

### 4.3. Bilateral Participant Behavior

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

### 4.4. Standalone Conformance Suite

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

Implementations claiming v0.3 conformance **MUST** pass the standalone conformance suite and should publish their claimed level using the guidance at https://universalmanifest.net/conformance/v0.3/.

> **Note:** The v0.3 conformance suite extends v0.2 with evaluation sequence, encrypted facet, and receipt schema validation tests. Full receipt behavioral tests (verifying that receipts honestly reflect pipeline execution) require a test oracle and are planned for v0.3.1.

## 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`. Evaluators ignoring unrecognized properties guarantees that domain-specific profiles do not compromise cross-system interoperability.

# 6. Security Considerations

Universal Manifest v0.3 defines a mandatory signature profile, an additive tiered trust model, and resource-limit guidance. This section specifies the security properties these mechanisms provide and the threats they mitigate.

## 6.1. Signature Limitations

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

## 6.2. TTL Enforcement

Bounding the `expiresAt` timeline dictates the primary line of defense against presentation replay spoofing.

## 6.3. Resource Limits

Denial-of-Service vectors originating from disproportionately inflated arrays or recursion logic **SHOULD** be countered with hard limits on payload ingestion:

- Maximum memory serialization threshold: 1 MB
- Maximum recursion/nesting depth: 10 layers
- Array length maximums: 1,000 bounds

## 6.4. Identity Binding and Claim Authenticity

### 6.4.1. Bag of Claims Limitation

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

- That the signer controls the `subject` DID (subject-signer binding).
- That the `subject` controls all DIDs mentioned in claims or facets (cross-DID control).

- That an issuer actually issued a claim listed in the manifest (claim authenticity). The `claims[].issuer` field is a string assertion, not a verified provenance chain.

Relying parties 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 higher-tier manifest also satisfies all lower-tier requirements. Higher tiers provide stronger guarantees but impose more user ceremony. The specification does NOT mandate a minimum tier; relying parties 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 relying party 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`). Relying parties can verify specific claims against their issuers or evaluate attester trust. Evaluators claiming Tier 1 assurance MUST enforce attester trust policy and freshness/expiry checks on the proof material used for Tier 1 acceptance. Evaluators MUST validate the `claimProof` proof chain or the attester's cross-DID binding attestation before granting Tier 1 trust. Suitable for medium-stakes use cases (social identity, reputation, basic access control).

**Tier 2 — Cryptographic binding.** Medium friction. Cross-DID control is cryptographically proven via zero-knowledge proof of cross-DID control. The subject demonstrates control of multiple DIDs without revealing private key material, using a ZK proof that the same entity controls the keys behind each DID. Evaluators claiming Tier 2 assurance MUST verify the ZK proof before granting Tier 2 trust. Suitable for high-stakes Sybil-resistance use cases. At risk: this feature may be removed if implementations do not materialize before v0.4.

**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. (Future: v0.4+.)

Relying parties MUST define their required trust tier based on their threat model. Relying parties MUST NOT extend trust from one DID in a manifest to another DID in the same manifest unless binding

proof material (Tier 1 or Tier 2) is present for that specific DID pair.

#### 6.4.3. `claims[].claimProof` Field

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

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

`claimProof` **MAY** be:

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

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

##### *6.4.3.1. Proof Entry Fields*

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

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

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

#### *6.4.3.2. Verification Procedure*

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

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

When present as a URI string, the evaluator **MAY** fetch the VP for verification when network access is available. Evaluators that cannot resolve the URI **SHOULD** report the claim as `claimProof-unresolved` rather than `verified`.

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

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

Size limits: maximum 50 KB per embedded VP; maximum 500 KB total VP payload across all claims in one manifest.

### 6.4.4. `identity.crossDidBinding` Claim

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

**Example 5: Cross-DID binding claim**

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

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

- `@type` — **MUST** be `"identity.crossDidBinding"`.
- `issuer` — DID or URI identifying the entity that asserts the claim. **MUST** be present (inherited from the base claim schema, Section 1.4.3).
- `boundDids` — Array of DID strings asserted as controlled by the same entity. **MUST** contain at least 2 DIDs. One **MUST** match the manifest `subject`.
- `attester` — DID or URI of the entity attesting the binding. **MUST** be present.
- `attestationMethod` — Human-readable description of the verification method used. **MUST** be present.
- `attestedAt` — ISO 8601 timestamp of when the attestation was produced. **MUST** be present.

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

- `claimProof` — URI, structured object, or array of proof entries pointing to the attestation proof (see Section 6.4.3).
- `expiresAt` — Attestation expiry. Evaluators **SHOULD** reject expired attestations.

Evaluators **MUST NOT** treat the presence of a binding claim as proof of common control unless they trust the attester. Evaluators **SHOULD** maintain a configurable attester trust list. Multiple binding claims for overlapping DID sets are independent assertions, not cumulative proof.

#### 6.4.5. `requiredTrustTier` Declaration

A manifest MAY declare the minimum trust tier required for specific claims, facets, or the manifest as a whole via the `requiredTrustTier` field. This field is an integer (0–3) indicating the minimum verification tier a relying party 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 relying party can only verify at Tier 1, the relying party MUST treat that claim as unverified. If absent, the default is 0 (no minimum required). The manifest-level value sets the floor; claim/facet-level values can only raise it, not lower it.

If a manifest or claim specifies a `requiredTrustTier` value for which the evaluator has no implemented verification profile (e.g., Tier 3 in v0.3, where no ceremony profile is defined), the evaluator MUST treat the item as unverifiable and record it in the receipt as `"trustTierUnsupported"`. The evaluator MUST NOT downgrade to a lower tier. The pipeline outcome for the overall manifest is `"accepted-partial"` if other items can still be processed, or `"rejected"` if the unsupported tier applies at the manifest level.

**Example 6: Manifest with requiredTrustTier**

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

#### 6.4.6. Bilateral Exchange

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

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

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

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

## 6.5. Agent Delegation

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

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

### 6.5.1. Structure

Required fields:

- `@type` — **MUST** be `"um:agentDelegation"`.
- `delegateType` — One of `"ai-agent"`, `"bot"`, `"proxy"`, or `"human-delegate"`.
- `delegatedBy` — DID of the delegating subject. **MUST** match the manifest `subject`.
- `delegatedAt` — ISO 8601 date-time when delegation was granted.
- `expiresAt` — ISO 8601 date-time when delegation expires. Platforms **MUST** reject expired delegations.

Optional fields:

- `delegateId` — DID or identifier of the delegate entity.
- `scope` — Array of capability strings the delegate may exercise.
- `livenessEndpoint` — URI for real-time liveness/delegation status queries.

**Example 7: Agent delegation pointer**

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

#### 6.5.2. Platform Guidance

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

## 7. Privacy Considerations

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

**Privacy-Binding Tension.** Universal Manifest supports both privacy-preserving pairwise DIDs and cross-DID binding mechanisms. These goals are in fundamental tension: stronger binding enables

correlation, while stronger privacy prevents binding verification. The specification does not fully address this tension at the protocol level. Instead, it provides mechanisms for both and requires relying parties to define their trust tier based on their specific threat model. Relying parties MUST NOT require cross-DID binding unless their use case demands Sybil resistance or trust transitivity. Subjects SHOULD use the minimum binding tier that satisfies their relying parties' requirements.

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

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

# 8. References

## 8.1. Normative References


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

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

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

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

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

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

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

**[RFC3339]**
G. Klyne, C. Newman. *Date and Time on the Internet: Timestamps*. IETF RFC 3339, July 2002.

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

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

## 8.2. Informative References

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

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

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

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

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

**[DID-CONFIG]**

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

**[RFC9449]**

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

**[VC-STATUS]**

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

**[VC-DATA-MODEL]**

Verifiable Credentials Data Model v2.0. W3C Recommendation, March 2025. Core data model for VCs and VPs.

**[ISO-29115]**

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

**[UM-BREAKING-CHANGE]**

Universal Manifest Breaking-Change Policy. Project governance document.

**[UM-DEPRECATION]**

Universal Manifest Deprecation Policy. Project governance document.

**[UM-INCIDENT]**

Universal Manifest Incident Response and Rollback. Project governance document.

**[UM-RELEASE]**

Universal Manifest Release Cadence. Project governance document.

**[UM-RFC]**

Universal Manifest Spec Improvement Queue (RFC mechanism). Project governance document.