Universal Manifest v0.5 — Primer

Informative

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/primer/

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 Primer is the breadth-first entry point to the Universal Manifest v0.5 document set. It is informative: it contains no normative requirements and defines no wire formats. Read it first to understand *what* Universal Manifest is and *how the pieces fit*. Then descend into the normative documents your work requires — starting with the Base specification, which everything else builds on.

Universal Manifest, in one sitting

What it is

A **Universal Manifest** is a portable, signed, self-describing capsule of state about a subject — a person, an app, a venue, a device — that one party can hand to another and the receiving party can verify *on its own*, without phoning home.

Think of it as a sealed envelope that travels with you. Inside are identity references, role permissions, device registrations, consent records, and pointers to authoritative data. Stamped across the outside is a cryptographic signature that proves the contents have not been altered since they were assembled, and a hard expiry that bounds how long anyone may rely on them. A smart display at a venue door, a kiosk, a peer's phone, or a network edge can read that envelope, check the stamp, apply its own policy, and act — even with no network connection at all.

That last property is the whole point. Universal Manifest is designed for **local-first** environments: venue edges, public displays, air-gapped rooms, NFC and BLE taps between two phones. In those settings an evaluator cannot assume a live connection to a cloud service of record. It must be able to decide, from the bytes in front of it plus material it already holds, whether to trust what it is looking at. A Universal Manifest is the standardized shape of those bytes, and this specification is the contract for how to read them.

Conceptually, Universal Manifest is a **hybrid** of two ideas the web already knows. From the *Web Publication Manifest* it borrows semantic linkability — linked-data identity references and pointers to canonical sources, so a manifest can describe a subject by reference rather than by copying everything inline. From the *Web App Manifest* it borrows an applied processing lifecycle — a defined sequence an evaluator runs to turn a received document into a decision. Put those together and you get a state capsule with both meaning and a runtime.

One framing worth keeping in mind, because it explains much of the design: a manifest is a **bag of claims, not a proof of truth**. The signature proves *who assembled the envelope*. It does not, by itself, prove that the subject controls every DID mentioned inside, or that an issuer really issued a claim listed there. The specification is largely about closing that gap honestly — giving evaluators graduated, explicit ways to verify the parts that matter, and to record exactly how far that verification got.

The shape of a manifest

Every manifest, no matter what kind, shares one fixed **envelope**. What makes a particular manifest an identity capsule versus a device descriptor versus a consent record versus a receipt is *which optional members it carries* — not a different schema per kind. The specification calls this a **polymorphic envelope**: one outer shape, many payloads.

The envelope has four layers:

- **Anchor members** identify and semantically ground the manifest: a context reference (which vocabulary version its terms come from), a globally unique opaque `id`, and a `type` that always includes `um:Manifest`. The context reference is what lets two implementations agree on what each term *means*.
- **Identity and lifespan members** say what the manifest is about and for how long: the `subject` (a stable identifier URI, typically a Decentralized Identifier), and the `issuedAt` / `expiresAt` pair that bounds its validity. Expiry is not a hint — it is an absolute rejection gate.
- **Structural state members** carry the actual payload, all optional, each a top-level array:
  - **`facets`** — composable functional blocks. A facet packages one verifiable capability, metadata subset, or configuration module, and may carry an encrypted payload that only intended recipients can open.
  - **`claims`** — statements about the subject, each with a type and an issuer, and optionally proof material that lets an evaluator verify the statement rather than merely read it.
  - **`consents`** — per-facet permission records that say what an evaluator is allowed to *do* with a facet's data, for what purpose, and until when.
  - **`pointers`** — references to canonical external data sources.
  - **`devices`** — registered hardware targets bound to the subject.
- **The integrity proof** — the `signature` member — seals the exact production signing bytes. Profile A (JCS + Ed25519) is mandatory for evaluators; optional Profile B (JCS + ES256) is additive. The v0.5 Profile A rule requires canonical non-identity prime-order points, a canonical scalar and the full-challenge uncofactored equation. The candidate defines this rule; new-version runtime qualification remains separate. See Base §1.6.

A useful mental model: the anchor and lifespan members and the signature are the *envelope you can always rely on being there*; facets, claims, consents, pointers, and devices are the *contents you discover by looking inside*. An evaluator that meets a member it does not understand keeps it (so the signature still verifies) but does not act on it (so unknown extensions can never change a

decision). That single rule — **preserve everything, act only on what you recognize** — is what makes the format extensible without becoming brittle.

For a field-by-field scan of the top-level object, see Base §1.1.1. For the serialization-independent property names behind the JSON-LD member names, see EXT-OPT §O1.1.

The lifecycle: how a manifest becomes a decision

The conceptual spine of the whole specification is the **six-stage evaluation sequence**. When any evaluator encounters a manifest, it runs the manifest through these six stages in order. Each stage produces a defined output that feeds the next, and the evaluator may stop early at any stage by emitting a rejection. Told as a story:

1. **Arrive.** The envelope is received and parsed. The evaluator confirms the required members are present and that `type` includes `um:Manifest`. It keeps any members it does not recognize — they must survive untouched through signature verification — but they will have no effect on the outcome.

2. **Verify.** The evaluator checks the seal and the clock. It verifies the signature over the canonical bytes; it rejects anything expired or with an issue-time after its expiry. If the manifest leans on stronger identity binding for higher assurance, this is where those binding checks run, each recording its own outcome. A failed signature ends the story here.

3. **Project.** The evaluator extracts only the facets, claims, pointers, and devices relevant to *its* context. Disclosure is holder-controlled: the holder chose what to include. Crucially, a facet that is absent is *not projected* — it is not evidence of absence. The evaluator must never assume it is seeing the subject's complete state.

4. **Consent.** Before touching any facet's data, the evaluator checks the matching consent record: is my intended operation in the consent's scope, is my purpose within its declared purpose, is it still inside its validity window? No valid consent, no processing. An encrypted facet the evaluator cannot open is acknowledged as a **sealed entry** — recorded as present, never guessed at.

5. **Compose.** The evaluator assembles the result into one of four honest outcomes: `accepted`, `accepted-with-warnings`, `accepted-partial` (some facets processed, others sealed or denied), or `rejected`. The result is machine-readable and carries per-facet status.

6. **Receipt.** The evaluator emits a **structured receipt** — a machine-readable record of what it actually did. The receipt must not hide failed checks or suppress negative outcomes. It is the

audit trail, and in v0.5 it can itself be a signed, chainable manifest.

Two design commitments run through every stage. First, **honesty**: a receipt records the real outcome, including the gap between what a manifest *declared* it needed and what the evaluator could actually *verify*. Second, **fail-safe defaults**: missing consent blocks processing, unverifiable trust caps out low, unknown content is preserved but inert. The evaluator never invents assurance it did not earn.

The core guarantees

Strip the specification to its promises and five remain:

- **Integrity.** A valid signature means the envelope's contents are exactly what the signer assembled. Tampering breaks the seal. (The signature proves authorship of the envelope — not the truth of every claim inside it; see the bag-of-claims framing above.)
- **Freshness.** Expiry is enforced absolutely. A manifest past `expiresAt` must be evicted or rejected. This is the primary defense against replaying a stale capsule.
- **Holder-controlled disclosure.** The holder decides which facets to include in any given presentation. Evaluators cannot infer what was withheld, and selective disclosure is a first-class, supported behavior rather than a leak.
- **Consent-gated use.** Facet data may be acted on only within an explicit, in-window, scope-and-purpose-matching consent. Sealed (encrypted) facets are honored as present-but-unread.
- **Honest, graduated trust.** Claims are verified along an explicit ladder of assurance, and the receipt records exactly which rung was reached. An evaluator never silently upgrades self-asserted data to verified, and never extends trust from one DID to another without binding proof for that specific pair.

These hold for the *baseline* of the specification — the part every conformant implementation must support. Everything beyond the baseline is layered on top in a disciplined way, which is the subject of the next section.

A map of the documents: tiers and extensions at one glance

Universal Manifest v0.5 is organized as a **conformance-tier stack**. The idea is simple: the question that decides what you must read and build is *"what level of assurance am I certifying to?"*

The baseline answers most needs; each higher tier and each optional capability is a self-contained companion document you add only when you need it.

The trust ladder. Trust tiers are *strictly additive* — a claim verified at a higher tier automatically satisfies every lower tier. Higher tiers buy stronger guarantees at the cost of more user ceremony. The specification mandates no minimum; an evaluator picks its tier from its own threat model.

- **Tier 0 — Signature-only.** Zero friction. Claims are self-asserted by the signer; the only cryptographic check is the manifest signature. Right for low-stakes interactions where trust in the signer is established out of band. *Never sufficient for Sybil-critical decisions.*
- **Tier 1 — Attested / proof-backed.** Low friction. Claims carry external proof of issuance (a Verifiable Presentation, or an attested cross-DID binding), plus a **holder binding** proving the party presenting the manifest is the same party the credential was issued to. Right for social identity, reputation, and basic access control.
- **Tier 2 — Cryptographic binding.** Medium friction. Control of multiple DIDs is proven in zero knowledge, without revealing key material. Right for high-stakes Sybil resistance.
- **Tier 3 — Multi-party ceremony.** High friction. Multiple keyholders — potentially different people in different places — co-sign, like a multi-signature wallet. Right for the highest-stakes organizational and financial contexts.

How that ladder becomes documents:

- **The Base** — *everything required to build a conformant Tier-0 / Tier-1 evaluator and holder.* The envelope, the six-stage lifecycle, the baseline signature profile, encrypted facets, receipts, conformance, and the base security and privacy model. The Base also *names* the Tier-1 binding requirement and points to its profile for the mechanics, so it stays at "what conformance demands" altitude. A developer can ship an interoperable baseline from the Base alone — the same test v0.3 passed. **Read this first; everything else assumes it.**

- **Extension — Tier-1 Binding Profile** — the mechanics that turn the Tier-1 *requirement* into running code: holder binding, presentation proofs, liveness attestation, the end-to-end claim-proof verification path, and the binding sub-steps and receipt fields. Read this when you implement Tier 1.

- **Extension — Tier-2 Cryptographic-Binding Profile** — zero-knowledge cross-DID proof profiles, the BBS+ selective-disclosure track that shares the same cryptography, and the post-quantum migration path. Read this when you implement Sybil-resistant high assurance.

- **Extension — Tier-3 Multi-Party Ceremony Profile** — the ceremony model, attester-role taxonomy, and threshold-protocol guidance. Read this for co-signed, highest-stakes

deployments.

- **Extension — Optional-Feature Profiles** — capabilities that are not rungs on the tier ladder but are independently optional: the format-independent abstract data model and its second (CBOR-LD) encoding, the `statusRef` revocation protocol, receipts-as-a-class with hash-chaining, the bilateral session model, federation, the profile registration mechanism, `trustWeight` and category trust, extended `devices` / `actorState` / sensor-consent schemas, and the cryptographic-requirements summary. Read the parts that match the feature you are adding.

- **The Cookbook** — every worked example, consolidated and organized by scenario and tagged by the tier it exercises. The normative documents keep one anchoring example per concept; the rest live here, with narrative connective tissue. Read this when you want a concrete payload to copy.

Reading paths, by who you are:

- *Evaluating Tier-0 signatures only?* Base, and you are done.
- *Building Tier-1 attested identity?* Base + Tier-1 Binding Profile.
- *Deploying Sybil-resistant assurance?* Base + Tier-1 + Tier-2 (and Tier-3 for co-signing).
- *Adding revocation, federation, sessions, or a new manifest class?* Base + the relevant parts of Optional-Feature Profiles.
- *Just want to see real manifests?* Primer → Cookbook.

The stable-versus-preview line. The Base is the production-candidate core, on track to harden toward v1.0. The advanced tiers and several optional features are marked **PREVIEW** — under active review, built into the draft on the editors' recommended defaults, and fully revisable before the schema locks. Naming a capability in the Base is breadth; specifying it in an extension is depth. That separation is what keeps the Base digestible while the frontier keeps moving — and it is exactly the discipline that kept v0.3 readable.

✧ ✧ ✧

This Primer summarizes; it does not bind. Where it and a normative document appear to differ, the normative document governs. Start with the Base specification.