Build with Universal Manifest (for Agents)

Contents

Build with Universal Manifest (for Agents) **1**
- How to read this kit . . . 1
- Two stability tiers (used throughout) . . . 2
- The single source of truth . . . 2
- 1. What Universal Manifest is, for a builder . . . 2
- 2. The 60-second mental model . . . 2
- 3. Decision: build now vs prepare for . . . 3
- 4. Quickstart paths . . . 4
- 5. Produce a manifest . . . 5
- 6. Consume a manifest . . . 6
- 7. Verify a manifest . . . 6
- 8. Integration patterns index . . . 7
- 9. Versioning and forward-compatibility . . . 8
- 10. Conformance — prove your build is correct . . . 9
- 11. Agent-discovery surfaces (find UM at runtime) . . . 9
- 12. The version index, and where to get updates . . . 10

Build with Universal Manifest (for Agents)

A version-aware integration kit. This guide is written for an AI agent — or the developer driving one — that needs to **produce, consume, or verify** Universal Manifests. It is the *map*: it does not restate the specification or copy the examples. It orients you, then sends you to the canonical source for every concrete artifact.

> **Why "version-aware"?** The Universal Manifest spec is moving and getting better. Two things matter at once: you can **build now** on what is stable, and you can **prepare for** what is coming. Every link in this guide resolves through a single version index (`versions.json`), so this same document keeps pointing at the right spec, schema, and fixtures as the spec advances. You are reading the distribution pinned to: **stable** `v0.3`, **preview** `v0.4-preview`.

How to read this kit

1. If you are deciding *whether and when* to adopt read **The 60-second mental model** then **Decision: build now vs prepare**.
2. If you are *implementing right now* jump to **Produce / Consume / Verify** and the **Quickstart paths**.
3. If you are integrating into a specific ecosystem (spatial fabric, DID/VC, healthcare, education, …) see **Integration patterns**.
4. If you are an RP1/spatial-fabric reader the integration index highlights the RP1 lane, and there is a companion video playlist at https://universalmanifest.net/videos/?topic=rp1-spatial-fabric.

Two stability tiers (used throughout)

Every instruction in this guide is tagged with one of two tiers:

- **[BUILD NOW — `v0.3`]** — the current published, stable spec. Safe to implement against today.
- **[PREPARE FOR — `v0.4-preview`]** — a working draft. Features are marked *[PREVIEW — Working Group Input Requested]* in the spec. Read these to anticipate the next version; do not depend on them in production until they are published.

The single source of truth

This kit **references**, it does not duplicate. The normative truth lives in the specification and the conformance suite; the runnable truth lives in the examples. When this guide says "see X", open X — it is current by construction, because the kit resolves the link through the version index rather than copying X into these pages.

- Stable spec: https://universalmanifest.net/spec/v0.3/
- Preview spec: https://universalmanifest.net/spec/v0.4-preview/
- Latest (version-tolerant) alias: https://universalmanifest.net/spec/latest/
- Version index (the spine): `versions.json` (bundled alongside this guide)

1. What Universal Manifest is, for a builder

A Universal Manifest is a **portable, signed, consented envelope**: one JSON-LD document that carries a subject's identity, credentials, preferences, pointers, and the permissions they set — and that any system can verify and project without phoning home to the issuer.

For a builder, three properties matter:

- **Portable** — it is just structured JSON-LD. No SDK is required to read or write one; libraries are a convenience, not a dependency.
- **Signed** — authenticity is cryptographic (Signature Profile A: JCS canonicalization + Ed25519), so a consumer can trust what it verifies offline.
- **Consented** — data is gated by explicit consent entries; a conformant consumer processes a facet only when consent permits, default-deny.

That is the whole concept at a builder's altitude. For the precise definition, the field model, and the normative envelope, do not rely on this paragraph — read the source:

- Concept, for any reader: https://universalmanifest.net/how-it-works/ and https://universalmanifest.net/about/one-pager/
- The normative spec abstract + envelope: https://universalmanifest.net/spec/v0.3/ (stable `v0.3`) — repo: `docs/W3C-STYLE-SPEC-v03.html`
- The structured artifacts you will actually consume: `spec/v0.3/` (schema, context, conformance).

  [BUILD NOW — `v0.3`] The required envelope is small. A minimal manifest is seven required members plus zero-or-more facets, consents, and pointers. Rather than reproduce the field list here (it changes per version), read the version-resolved minimal fixture in **§5 Produce**.

2. The 60-second mental model

If you remember one thing: **a manifest is a portable, signed, consented envelope, and a conformant consumer always evaluates it through the same fixed sequence of stages.** Produce one; hand it over; the other side runs the sequence and hands you back a receipt.

[BUILD NOW — `v0.3`] The evaluation sequence. Every conformant evaluator runs these stages **in order** for every manifest it processes. Each stage feeds the next:

1. **Arrive** — parse the JSON; confirm the required envelope members; ignore but *preserve* unknown fields (forward-compatibility).
2. **Verify** — verify the signature against the JCS-canonicalized payload; enforce freshness (TTL, with a small clock-skew tolerance); optionally resolve revocation status.
3. **Project** — extract only the facets, claims, and pointers relevant to your processing context. What you do not project is *not projected* — not the same as *absent*.
4. **Consent** — match consent entries against each projected facet; check scope, purpose, expiry, and withdrawal. Default-deny: no matching consent means do not process the facet's data.
5. **Compose** — produce one outcome: accepted, accepted-with-warnings, accepted-partial, or rejected — with a per-facet status.
6. **Receipt** — emit a structured receipt that honestly records what you did.

This summary is deliberately non-normative and version-light. The **normative** stage definitions, the receipt schema, the outcome and facet-status enumerations, and the clock-skew tolerance all live in the spec — read them there so you are never working from a stale paraphrase:

- Sequence + summary: `spec/v0.3/README.md` ("Six-stage evaluation sequence")
- Normative detail: https://universalmanifest.net/spec/v0.3/ (spec text §3.1, receipts §3.3)
- Evaluator conformance checklist: `spec/v0.3/CONFORMANCE.md`

The trust tiers (one line each). Relying parties pick a tier by threat model; tiers are strictly additive:

- **Tier 0** — signature-only (zero friction; self-asserted claims).
- **Tier 1** — attested / proof-backed (VP/attestation or cross-DID binding).
- **Tier 2** — cryptographic binding (ZKP). *Normatively named in `v0.3`; the proof-system profile is `v0.4-preview` work — see §3.*
- **Tier 3** — multi-party ceremony. *`v0.4-preview` and beyond.*

Read the normative tier definitions in `spec/v0.3/README.md` ("Tiered trust model") and the spec text — do not implement tiers from these one-liners.

3. Decision: build now vs prepare for

This is the section the rest of the kit hangs on. Universal Manifest is a living spec. You do not have to wait for it to be "finished" — you can ship against the **stable** version today, and you can design your data model so the **preview** version is a small step, not a rewrite.

Build now on the stable version — `v0.3`

Treat everything in the stable spec as safe to implement. As of this distribution, the stable line gives you a complete, conformance-testable base: the six-stage evaluation sequence, structured receipts, normative schemas for the core arrays, encrypted facets (JWE inline profile), `requiredTrustTier`, cross-DID binding (Tier 1), and an agent-delegation pointer. For the authoritative "what's in the stable version" list, read `spec/v0.3/README.md` — it is kept current with the spec; this guide intentionally does not duplicate it.

If you are building today, pin to `v0.3`:

- Use the stable schema + context: `/ns/universal-manifest/v0.3/schema.json` and `/ns/universal-manifest/v0.3/`
- Round-trip the stable fixtures: `examples/v0.3/`.
- Gate releases on the stable conformance fixtures: `conformance/v0.3/`.

Prepare for the preview version — `v0.4-preview`

The preview line is where the working group is actively designing. **Every preview feature is marked *[PREVIEW — Working Group Input Requested]* in the spec** — that label is your signal: read it to anticipate, prototype behind a flag, and give feedback, but do not make production depend on it until it is published as the next stable version.

What the preview line is shaping (read the preview spec for the current, authoritative set — not this list, which the version index keeps honest):

- the Tier 2 ZKP proof profile and the Tier 3 ceremony,
- JWE algorithm constraints and a statusRef resolution schema,
- a bilateral session model and profile registration,
- post-quantum signatures and federation.

To prepare:

- Read the preview spec: https://universalmanifest.net/spec/v0.4-preview/ (repo: `docs/W3C-STYLE-SPEC-v04-preview`
- Inspect preview manifests: `examples/v0.4/` and the preview schema `/ns/universal-manifest/v0.4/schema.json`.
- Design for forward-compatibility now (see §9) so a `v0.3` → `v0.4-preview` move is additive.

The rule of thumb

You are…	Pin to	Read for the future
Shipping a real integration	`v0.3` (stable)	the preview spec, occasionally
Prototyping / giving WG feedback	`v0.4-preview` (preview)	both
Writing version-tolerant code	`/spec/latest/` alias + capability checks	§9 Versioning

When stability matters, **pin to a concrete version**. When you want to follow the head and you have handled unknown fields gracefully, reference the version-tolerant alias https://universalmanifest.net/spec/latest/. How that alias behaves (it 301-redirects to a concrete version, never serves mutable content) is defined in the versioning policy: `docs/PUBLISHING-AND-VERSIONING.md`.

4. Quickstart paths

There are runnable, canonical examples in the repository. This kit does not re-teach them — it sequences them so an agent knows what to read in what order. All paths are in the repo; the code examples live under `examples/code/` (index: `examples/code/README.md`).

The 30-second path

```
git clone https://github.com/grigb/universal-manifest.git
cd universal-manifest/examples/code/01-hello-world
node create-manifest.mjs | node validate-manifest.mjs
```

That creates and validates a manifest with nothing but Node.js 18+.

Suggested reading order for an agent

Order	Example (under `examples/code/`)	Why read it
1	`01-hello-world`	The minimal create + validate loop; the envelope shape.
2	`04-facets-and-pointers`	Compose embedded data (facets) vs external references (pointers).

Order	Example (under `examples/code/`)	Why read it
3	`09-unknown-field-tolerance`	The forward-compatibility guarantee — critical for version-tolerant code (§9).
4	`02-validate-and-check-ttl`	Structural validation vs freshness (TTL).
5	`03-create-signed-manifest-v02`	The Ed25519 signature lifecycle + JCS canonicalization.
6	`08-consent-enforcement`	Default-deny consent gating (the Consent stage).
7	`06-social-profile-projection`	A realistic projection: manifest consented HTML view.
8	`10-multi-manifest-aggregation`	Assembling data from multiple manifests.

Cross-language: `python-validate` (Python 3.8+) and `browser-validate` (no Node) demonstrate that conformance is language-agnostic.

Then choose an implementation path

When you are ready to build something real, the adopter guide gives you two routes (do not duplicate it — read it): `docs/IMPLEMENTING-UM.md`.

- **[BUILD NOW — `v0.3`] Path A — fork the reference implementation.** Start from a working TypeScript artifact (https://github.com/grigb/um-typescript), swap internals to your runtime, keep the conformance adapter compatible. Fastest to a green conformance run.
- **[BUILD NOW — `v0.3`] Path B — build from scratch, any language.** Use the published spec artifacts + the standalone conformance suite as the only inputs. No language constraint.

Both converge on the same gate: **a passing conformance report** (see §10).

5. Produce a manifest

The agent task: emit a valid, signed manifest for a subject.

This kit gives you the canonical artifacts to work from rather than a JSON listing that would drift from the schema. Resolve everything through the version you are building against (default: stable `v0.3`).

[BUILD NOW — `v0.3`] Steps:

1. **Start from the minimal fixture, not from memory.** Read the version-resolved minimal manifest and copy its shape: `examples/v0.3/minimal-signed-manifest.jsonld`. It is the ground truth for the required envelope members.
2. **Declare the right context.** Manifests use the `@context` value `https://universalmanifest.net/ns/v0.3`. The machine-readable context document is `/ns/universal-manifest/v0.3/schema.jsonld`; the structural schema is `/ns/universal-manifest/v0.3/schema.json`.
3. **Add facets, consents, and pointers as needed.** See `examples/code/04-facets-and-pointers` for the composition pattern. Keep bulk data in facets or behind pointers, not inflated into the envelope.
4. **Sign it (Signature Profile A).** JCS-canonicalize the payload and sign with Ed25519. The lifecycle is demonstrated in `examples/code/03-create-signed-manifest-v02`; the profile text is `spec/v0.2/SIGNATURE-PROFILE.md`.
5. **Validate before you ship.** Run structural validation + a TTL check: `examples/code/01-hello-world` and `02-validate-and-check-ttl`.

Do **not** transcribe schema field tables into your prompt or your code comments — they are versioned. Point your build at `/ns/universal-manifest/v0.3/schema.json` and the minimal fixture, and you will track the spec automatically.

[PREPARE FOR — `v0.4-preview`] If you are prototyping against the preview line, start instead from `examples/v0.4/minimal-signed-manifest.jsonld` and validate against `/ns/universal-manifest/v0.4/schema.json`. Treat any preview-only members as experimental.

6. Consume a manifest

The agent task: receive a manifest, project the parts relevant to your context, and gate them by consent — default-deny.

Consuming is stages 3–4 of the evaluation sequence (Project, Consent). Run them in order, after you have verified (§7).

[BUILD NOW — `v0.3`] Steps:

1. **Project, do not slurp.** Extract only the facets, claims, and pointers your processing context needs. Record what you did not project as *not projected*, distinct from *absent* — this honesty matters for the receipt.
2. **Consent-gate every projected facet.** Match each projected facet against the manifest's consent entries; check scope, purpose, expiry, and withdrawal. If there is no matching consent, the facet status is consent-missing and you MUST NOT process its data. Treat expired or withdrawn consents as absent. The normative rules are in `spec/v0.3/CONFORMANCE.md` (evaluator requirements) and spec text §3.1 — read them; do not reimplement from this summary.
3. **Handle sealed (encrypted) facets gracefully.** If a facet is encrypted (JWE inline profile) and you lack the key, treat it as a sealed entry — acknowledge it, do not process it, and **do not reject the manifest** just because you cannot decrypt it.

Worked references (do not duplicate — read):

- Default-deny consent gating: `examples/code/08-consent-enforcement`.
- A realistic projection (manifest → consented profile view): `examples/code/06-social-profile-projection`.
- Multi-source assembly: `examples/code/10-multi-manifest-aggregation`.
- Canonical consented fixture to test against: `examples/v0.3/manifest-with-facets-and-consents.jsonld`.

Why projection + default-deny is the point. The same manifest can serve a social platform, a headset, and a spatial-fabric runtime — each projects a different, consented view. That is how one envelope stays portable across systems without over-sharing. The spatial-fabric case is worked in §8.

7. Verify a manifest

The agent task: before you trust anything in a manifest, verify its signature and freshness, and emit a structured receipt of what you did.

Verifying is stage 2 (Verify) and stage 6 (Receipt) of the sequence. Verify *first*, project *after*.

[BUILD NOW — `v0.3`] Steps:

1. **Verify the signature.** Re-canonicalize the payload with JCS and verify the Ed25519 signature against the declared key (Signature Profile A). Enforce the `keyRef` ↔ inline-public-key consistency check — skipping key resolution is non-conformant and opens a key-substitution attack. Profile text: `spec/v0.2/SIGNATURE-PROFILE.md`; lifecycle demo: `examples/code/03-create-signed-manifest-v02`.
2. **Enforce freshness.** Reject expired manifests (TTL: issuedAt → expiresAt), allowing the small clock-skew tolerance the spec defines. Demo: `examples/code/02-validate-and-check-ttl`.
3. **Optionally resolve revocation status** if the manifest carries revocation metadata. Fixture: `examples/v0.3/manifest-with-revocation-metadata.jsonld`.

4. **Emit a structured receipt.** Record the outcome (accepted / accepted-with-warnings / accepted-partial / rejected), the signature and freshness check results, and per-facet statuses. The required receipt members and enumerations are normative — read `spec/v0.3/CONFORMANCE.md` (it lists exactly what the receipt MUST include) rather than inventing the shape.

Trust tier enforcement. If the manifest (or a claim/facet) declares a `requiredTrustTier`, enforce it raise-only — a claim/facet floor cannot lower the manifest floor. If you have no verification profile for the required tier (e.g. Tier 3 under `v0.3`), record `trustTierUnsupported` and **do not downgrade**. Normative detail: `spec/v0.3/README.md` ("Tiered trust model") and the spec text.

[PREPARE FOR — `v0.4-preview`] The Tier 2 ZKP proof profile and the Tier 3 ceremony — the verification machinery for the higher tiers — are being defined in the preview line. Inspect `examples/v0.4/manifest-with-zkp-binding` and `manifest-with-ceremony-proof.jsonld` to see the direction.

8. Integration patterns index

The repository ships non-normative integration guidance per ecosystem under `integrations/` (each follows `integrations/TEMPLATE.md` and references real fixtures). This kit indexes them — read the lane you need; do not paste its prose.

Lane (`integrations/`)	One-line intent
`rp1-spatial-fabric.md`	**UM as the portable identity/consent/pointer envelope for RP1/MSF-style spatial fabrics** — runtime does live scope traversal; UM stays transportable. (Highlighted below.)
`metaverse.md`	Portaling and digital-world interoperability — carrying self + assets between worlds.
`smart-glasses.md`	A social layer for smart glasses / headsets projecting a consented view.
`smart-home.md`	Smart-home device management with trust levels.
`did-vc.md`	DID + Verifiable Credential lane — binding manifests to DIDs and presenting VCs.
`proof-of-personhood.md`	Multi-provider proof-of-personhood without lock-in.
`delegated-agent-provenance.md`	Delegated-agent provenance + credential-presentation profile (agent acting for a subject).
`oma-trust.md`	OMATrust integration lane.
`healthcare-patient-consent.md`	Patient-consent lane mirroring health data-sharing models.
`education-credentials.md`	Education credentials lane.
`gpc-global-privacy-control.md`	Honoring Global Privacy Control signals.
`data-firewall-ux.md`	Data-firewall UX patterns for bounded disclosure.
`portable-identity-profile-xr.md`	Portable identity profile for XR.
`social.md`	Social/profile projection notes.
`reference-runtime.md` / `runtime-profile.md`	Reference-implementation integration + profile conventions.
`public-discovery-publication-policy.md`	Public discovery + publication policy profile.

Highlighted for the RP1 / spatial-fabric reader

If you build on RP1 or an MSF-style spatial fabric, **Universal Manifest is a companion, not a replacement.** The fabric keeps doing what it does best — live scope traversal, topology, anchor services, asset loading. UM rides alongside as the **portable, signed, consented envelope** for the things that must survive *outside*

a single runtime session: subject identity and consent, pointer routing to fabric/anchor/place/attachment resources, compact attachment-policy summaries, asset-profile hints, and post-exchange freshness evidence.

- Lane guidance (suggested pointer names like `rp1.fabric`, `rp1.anchorSet`, `rp1.attachmentIndex`, `rp1.sessionContext`; consent keys; fail-closed guardrails): `integrations/rp1-spatial-fabric.md`.
- Worked fixtures (baseline visitor tier + a near-real multi-fabric cross-world attachment): `examples/integrations/rp1-s`
- Companion explainer videos: https://universalmanifest.net/videos/?topic=rp1-spatial-fabric.

The design rule the lane insists on: keep RP1/MSF semantics out of core required UM fields, use pointers (not embedded scope trees or 3D assets), and **fail closed** — stale attachment evidence blocks child-scope traversal; an expired/revoked session pointer is never replayed. Read the lane doc for the normative-vs-non-normative boundary; the contract itself stays in `spec/` and `conformance/`.

9. Versioning and forward-compatibility

The single most important habit for building on a moving spec: **tolerate what you do not understand.**

[BUILD NOW — `v0.3`] Unknown-field tolerance. A conformant consumer ignores but *preserves* fields it does not recognize (stage 1, Arrive). This is the forward-compatibility guarantee: a manifest written for a newer version still round-trips through your older consumer without breaking, and you do not drop data you simply do not process. Worked example: `examples/code/09-unknown-field-tolerance`.

Pin vs follow-the-head

- **Pin to a concrete version when stability matters.** Reference `/ns/universal-manifest/v0.3/schema.json` / `/ns/universal-manifest/v0.3/schema.jsonld` and the version id `v0.3` directly. Your build is reproducible and immune to spec movement until you choose to bump.
- **Follow the head with the alias when you want the latest.** Reference the version-tolerant alias https://universalmanifest.net/spec/latest/ (and the schema/context `latest/` aliases). The alias **301-redirects to a concrete version and never serves mutable content** — so "latest" is always a real, immutable version under the hood. The policy: `docs/PUBLISHING-AND-VERSIONING.md`.

How the spec moves (so you can track it)

- Versions are **immutable** once published. A change ships as a *new* version folder (`v0.3` → the next one), never an in-place edit. During the v0.x line, breaking changes are allowed but always arrive via a new version folder with a migration note. Full rules: `docs/PUBLISHING-AND-VERSIONING.md` and the fixture-mirror policy `docs/FIXTURE-SOURCE-OF-TRUTH-AND-PUBLISHED-MIRROR-POLICY.md`.
- **To track movement programmatically:** watch the version index (`versions.json` — this kit's spine), or resolve `/spec/latest/` and compare the concrete version it redirects to against your pinned version.

Writing a version-tolerant integration (checklist)

- ☐ Ignore-and-preserve unknown members (do not hard-fail on new fields).
- ☐ Resolve schema/context/examples through a single indirection (a version index), not scattered hard-coded URLs — so a bump is one edit.
- ☐ Default-deny on consent and fail-closed on freshness, so a newer manifest cannot trick an older consumer into over-sharing.
- ☐ Re-run conformance (§10) against the spec's `main` on a schedule to catch upstream drift within days, not at complaint time.

This kit eats its own cooking: the guide and skill you are reading contain **no hard-coded version strings** in their body. Every `v0.3`/`v0.4-preview` reference and every spec/schema/example link

is resolved from `versions.json` at assemble time. Bumping the spec version is one edit to that file.

10. Conformance — prove your build is correct

Conformance is **language-agnostic**. The spec repo treats the standalone conformance suite as the contract; any implementation in any language that passes it is conformant. This kit points you at the suite — it does not restate the contract.

[BUILD NOW — `v0.3`] The gate. A passing conformance report against the stable fixtures. Both implementation paths (§4) converge here.

1. **Read the conformance contract.** `spec/v0.3/CONFORMANCE.md` defines the evaluator / issuer / bilateral-participant requirements. The suite layout and expected-results schema are in `conformance/README.md`.
2. **Run against the stable fixtures.** Valid + invalid fixtures live at `conformance/v0.3/` (with `expected.json` listing the expected accept/reject outcome and category for each). Invalid fixtures MUST be rejected; valid fixtures MUST round-trip.
3. **Wire an adapter.** The standalone runner (`conformance/runner/`) speaks two adapter modes — command (subprocess over stdin/stdout) or HTTP. Use whichever suits your language. The canonical example adapter and the full invocation are documented in `docs/IMPLEMENTING-UM.md` (Path B).
4. **Run it on a schedule.** Re-run against the spec repo's `main` weekly so you catch upstream contract changes within days (drift detection). The reference implementation does exactly this.
5. **Declare conformance.** Publish your `conformance-report.json` and, optionally, consume a badge from `conformance/badges/`.

Reference path. The TypeScript reference implementation (https://github.com/grigb/um-typescript, repo helper `packages/universal-manifest/`) is *implementation #1*, not "the official one" — multiple independent implementations are the goal. Fork it (Path A) for the fastest green run, or use it only as an oracle while you build from scratch (Path B).

[PREPARE FOR — `v0.4-preview`] Preview fixtures, where present, live at `conformance/v0.4/`. Treat preview conformance as a moving target until the preview line is published as stable.

11. Agent-discovery surfaces (find UM at runtime)

If you are an agent encountering Universal Manifest in the wild, the site exposes machine-readable entry points. Use these to discover the canonical artifacts at runtime instead of hard-coding them.

- `/llms.txt` (https://universalmanifest.net/llms.txt) — the canonical agent onboarding file: discovery order, machine-readable artifact URLs, tool surfaces, and the interaction model. Start here.
- `/.well-known/universal-manifest.json` (https://universalmanifest.net/.well-known/universal-manifest.json) — the well-known descriptor for the standards host.
- `/agent/fixture-catalog.json` (https://universalmanifest.net/agent/fixture-catalog.json) — a machine-readable inventory of published fixture files (path, authored source path, surface, version, kind).
- `/agent/sandbox-scenarios.json` (https://universalmanifest.net/agent/sandbox-scenarios.json) — a machine-readable inventory of public sandbox scenarios (id, title, category, spec version, route, fixture URL).
- **Resolver:** `https://myum.net/{UMID}` resolves a manifest by its Universal Manifest Identifier at runtime (separate from spec hosting).

  Version note for agents. At the time this distribution was assembled, the `llms.txt`, fixture-catalog, and sandbox-scenarios surfaces advertised an older `currentSpecVersion` than the stable spec this kit pins to (`v0.3`). Treat the **spec directory and the version index as authoritative for "what is stable"**, and use these discovery surfaces for *locating* fixtures, scenarios, and tool endpoints. (This drift is a known item for the site maintainers.)

The site also exposes interactive tool surfaces — a sandbox, a proof harness, a workbench, and a resolver UI — linked from `/llms.txt`. They are useful for a human or agent to exercise manifests without writing code.

12. The version index, and where to get updates

This kit is built on one idea: **a single switch controls every version-specific reference.** That switch is `versions.json`, bundled alongside this guide (and inside the skill).

What `versions.json` holds

- `stable` and `preview` — the two version ids this kit tracks.
- `stabilityTiers` — the BUILD NOW / PREPARE FOR mapping used throughout the guide.
- `versions.<id>` — for each version: its spec URL, schema, JSON-LD context, example directory, conformance fixtures, minimal fixture, and signature profile.
- `sharedSources` — version-independent canonical paths (the implementation guide, code examples, integration lanes, conformance runner, reference implementation, versioning policy, discovery surfaces, resolver).
- `latestAlias` and the schema/context `latest/` aliases — the version-tolerant references.

How resolution works (for the agent driving the kit)

Every link in this guide is written in the *source* as a placeholder token (a stable token, a preview token, a latest-alias token, and per-field version / shared-source tokens) and is **resolved from `versions.json` when the guide is assembled**. The published guide you are reading has those tokens already filled in for the pinned versions. If you have the *source* guide and the version index, re-running the assemble step re-points everything — there are no version strings hand-written in the body to hunt down.

Bumping to a new spec version

When the spec advances:

1. Add the new version's entry under `versions.<id>` in `versions.json`.
2. Move `stable` (and/or `preview`) to the new id; update `stabilityTiers`.
3. Re-run `assemble.sh`.

That is the entire change. The guide, the PDF, the skill, and the site page all re-point to the new spec, schema, fixtures, and conformance suite — with **zero hand-edited duplication**. This mirrors the proven spec build pattern (`tools/spec/update-spec-version.sh`): one canonical source, many derived distributions.

Where to get updates

- The living spec: https://universalmanifest.net/spec/latest/ (always redirects to the current stable concrete version).
- The version index in this kit: `versions.json`.
- The site's agent-discovery surfaces (§11) for runtime discovery.

This guide is the map. The territory — the normative spec, the runnable examples, the conformance contract — lives in the Universal Manifest repository and on https://universalmanifest.net. Build now on `v0.3`; prepare for `v0.4-preview`.