OMATrust Specification

Decentralized and Permissionless Trust Layer for the Open Internet

1. Executive Summary

This document proposes a decentralized and permissionless trust layer for the open internet, called OMATrust. It has three main components:

- Application Registry- a registry of tokenized Applications that are identified by DIDs.
- Ownership Resolver- a smart contract that resolves token ownership disputes.
- Reputation System- infrastructure for attestations on DIDs, such as cybersecurity certifications.

The specification details the on-chain and off-chain metadata for tokenized applications, the processes for metadata confirmation and ownership verification, and control policies for versioning. It also outlines the Reputation System, including DID to Index Address mapping, attestation querying, EAS integration, and various schema definitions for linked identifiers, data URL attestations, endorsements, certifications, and user reviews.

The document emphasizes client guidance for adopting an attestation-based trust policy to prevent fraud and ensure user trust in a decentralized environment.

2. Scope

2.1 In-Scope

This document aims to solve the following requirements and use cases described in the Spatial Store RFP.

2.2 Out-of-Scope

Code repositories, tokenomics, business models, and storage implementations are out of scope for this document.

3. References

IWPS Identity Specification: https://github.com/oma3dao/iwps-specification
Spatial Store RFP: https://github.com/oma3dao/spatial-store-rfp
OMATrust Whitepaper: https://github.com/oma3dao/omatrust-docs/blob/main/whitepaper.md
DID Specification: https://www.w3.org/TR/did-core
DID Spec Registries: https://www.w3.org/TR/did-spec-registries/
Metaverse Standards Forum Spatial Store use case
Metaverse Standards Forum Autonomous Payments use case

4. Definitions

4.1 Abbreviations

In the present document, the following abbreviations apply:

DID	Decentralized Identifier
DNS	Domain Name System
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
ID	Identifier
IdP	Identity Provider
RP	Relying Party
TTL	Time To Live
URL	Uniform Resource Locator
IWPS	Inter World Portaling System
VC	(W3C) Verifiable Credential

Relevant External Specification Terminology Sections

DID Spec	https://www.w3.org/TR/did-core/#terminology
VC Spec	https://www.w3.org/TR/vc-data-model-2.0/#terminology

The following applies to the specification:

- “string” means a UTF-8 string
- “URL” means a string in a specific format
- “JSON” means a string in JSON format
- [] is meant to signify an array of whatever is inside the brackets.

4.2 Definitions

The following definitions are used within the present document.

Term	Definition
Application	A software service that can come in many formats, including a self-contained software application on a Device, an API endpoint, or a smart contract.
Client	Software that queries OMATrust to obtain information on an Application.
Decentralized Identifier	An identifier that adheres to the W3C DID standard.
DID Document	A JSON-LD document containing information about the DID, such as public keys and service endpoints.
Owner	The entity that controls the address that owns the app token. Owner may or may not be the same entity that minted the token.
Issuer	An entity that issues credentials as attestations in the Reputation Service.

In addition to the above definitions all OMA3 specifications use requirements language as described in the OMA3 working group process.

5. Specification

5.1 App Registry Contract

5.1.1 Onchain Metadata

The Application Registry contract tokenizes applications on the blockchain as an NFT. This document describes the specifications for this contract in a platform-agnostic manner.

Every app registry NFT stores the following information associated with an application.

Field	Format	Description	Req?	Mutable?

did	string	See Table 3	Y	N
fungibleTokenId	string	CAIP-19 token ID	N	N
contractId	string	CAIP-10 contract address	N	N
versionHistory	[object]	Array of released version structs, append only	Y	Y
status	enum	Active, deprecated, replaced	Y	Y
minter	address	Address of the transaction signer.	Y	N
owner	address	Current owner (built into ERC-721)	Y	Y (soulbound optional)
dataUrl	URL	URL to offchain data	Y	Y
dataHash	string	Hash of the JSON returned by dataUrl (see 5.1.3.2)	Y	Y
dataHashAlgorithm	string	The hash algorithm used to compute dataHash. Values: "keccak256", "sha256"	Y	Y
traitHashes	[string]	A structure of hashed traits. Implementation is different for each VM. Implementations SHOULD cap on-chain traitHashes to ≤ 20 entries to mirror the off-chain keywords cap, and clients MUST NOT assume more than 20 are indexed. See Appendix C.	N	Y
interfaces	[enum]	An unordered set of interface capability codes. Multiple capabilities may be present. Example: bitmap with 0 = human, 2 = api, 4 = smart contract	Y	N

Table 1: Application Registry Onchain Data.

All fields listed above require confirmation according to Section 5.1.3. Confirmation methods vary by field (e.g., ownership confirmation, attestation validation).

5.1.1.1 JSON Format: `versionHistory`

The objects in the **`versionHistory`** array have the following fields:

Value	Format	Required
major	Int	Y
minor	Int	Y
patch	Int	Y

5.1.1.2 JSON Format: `dataUrl`

`dataUrl` points to an endpoint that returns a JSON object with offchain data. The JSON object has the following top level fields depending on the value of the **`type`** field in the NFT contract:

Value	Format	Description	Interface		
			0	**2**	**4**
name	string	App name. Matches ERC-721 metadata extension.	Y	Y	Y
external_url	URL	URL of the app's market website. Matches ERC-721 metadata extension.	O	O	O
image	URL	URL to the application icon. Matches ERC-721 metadata extension.	Y	O	O
description	string	Long description of the application. Max 4000 chars. Matches ERC-721 metadata extension.	Y	Y	Y
publisher	string	Publisher name.	Y	Y	Y
summary	string	Short description of the application. Max 80 chars.	Y	O	O
owner	string	CAIP-10 address of the owner of the app token NFT. Used to confirm ownership of the dataUrl	Y	Y	Y

		JSON (see “Field Confirmation” below).			
screenshotUrls	[URL]	JSON urls field contains an array of URLs to screenshot images.	Y	N	N
videoUrls	[URL]	JSON urls field contains an array of URLs to videos.	O	N	N
3dAssetUrls	[URL]	JSON urls field contains an array of URLs to 3D assets (GLB, USDZ, etc.).	O	N	N
legalUrl	URL	URL to legal agreements like license, terms of service, privacy policy, etc.	O	O	O
supportUrl	URL	URL to get support.	O	O	O
iwpsPortalUrl	URL	See IWPS Spec.	O	O	N
traits	[string]	An array of max 20 traits and the total char count cannot exceed 120. See Appendix C.	O	O	O
interfaceVersions	[string]	Array of supported versions of the interface	N	O	O
platforms	JSON	Object of platforms supported by the app.	Y	N	N
endpoint	JSON	Details on the API endpoint (see below)	N	Y	O
payments	[JSON]	Array of payment mechanisms (see below). If this exists then payment is required to use the service.	O	O	N
artifacts	JSON	Allows clients to verify content (e.g.- binaries)	O	O	N
mcp	JSON	Implements the MCP server specification (see below)	N	O	N
a2a	URL	URL to the ~/well-known/agent-card.json endpoint as defined in the A2A standard	N	O	N

Table 2: Application offchain data.

5.1.2 Offchain Metadata

5.1.2.1 JSON Format: **`dataUrl.platforms`**

If **`interfaces`** = 0, The **`platforms`** JSON object MUST contain one or more of the following fields depending on how the human user interacts with the app:

Value	Format	Description	Required
web	JSON	Web is supported	O
ios	JSON	iOS is supported	O
android	JSON	Android is supported	O
windows	JSON	Windows is supported	O
macos	JSON	MacOS is supported	O
meta	JSON	Meta Quest is supported	O
playstation	JSON	Playstation is supported	O
xbox	JSON	XBox is supported	O
nintendo	JSON	Nintendo is supported	O

A **`platform`** field is a JSON object that has the following fields:

Value	Format	Description	Required
launchUrl	string	URL to launch the app	Y
supported	[string]	iPhone, iPad, x64, arm64, etc.	O
downloadUrl	string	URL to download a binary	O
artifactDid	string	See Appendix A	O

5.1.2.2 JSON Format: **`dataUrl.endpoint`**

The **`endpoint`** JSON object contains the following fields:

Value	Format	Description	Required
url	string	URL of the endpoint	Y
format	string	See Appendix C	O
schemaUrl	string	URL to API	O

		documentation	

For **`interface`** = 4 (contracts), the chain ID is taken from the DID (did:pkh with CAIP-10 ID). Clients can then determine the format the RPC endpoint requires based on the chain ID.

5.1.2.3 JSON Format: **`dataUrl.payments`**

The **`payment`** array contains at least one JSON object if payment is required. The possible JSON objects are differentiated by the **`type`** field. There are currently only two possible values for the **`type`** field: **`x402`** and **`manual`**. The below tables describe these two objects.

Value	Format	Description	Required
type	string	“x402”	Y
url	string	Base URL of the x402 endpoint. Clients SHOULD call GET {url}/supported for live tuples (source of truth). If this field is not present, use *dataUrl.endpoint*.	O
chains	[string]	CAIP-2 chain IDs this service supports.	O

Value	Format	Description	Required
type	string	“manual”	Y
url	string	Describes pricing information and payment mechanics.	O

5.1.2.4 JSON Format: **`dataUrl.mcp`**

This object represents the MCP specification and gives agents the information they need to interface with an MCP server. The reader is referred to the specification of MCP v1.0 (modelcontextprotocol.io, Section 3, Server Metadata), which has descriptions of each field and is incorporated by reference.

The following are the JSON fields for the **`mcp`** field:

Value	Format	Description	Required

tools	[JSON]	See below	Y
resources	[JSON]	See below	Y
prompts	[JSON]	See below	Y
transport	JSON	See below	Y
authentication	JSON	See below	Y

5.1.2.4.1 JSON Format: **`dataUrl.mcp.tools`**

The **`tools`** array contains JSON objects with the following fields:

Value	Format	Description	Required
name	string		Y
description	string		Y
inputSchema	JSON		Y
annotations	JSON		N

5.1.2.4.2 JSON Format: **`dataUrl.mcp.resources`**

The **`resources`** array contains JSON objects with the following fields:

Value	Format	Description	Required
uri	string		Y
name	string		Y
description	string		N
mimeType	string		N

5.1.2.4.3 JSON Format: **`dataUrl.mcp.prompts`**

The **`prompts`** array contains JSON objects with the following fields:

Value	Format	Description	Required
name	string		Y
description	string		Y
arguments	[JSON]		N

5.1.2.4.4 JSON Format: **`dataUrl.mcp.transport`**

The **`transport`** JSON object contains the following fields:

Value	Format	Description	Required
http	JSON		N
stdio	JSON		N

5.1.2.4.5 JSON Format: **`dataUrl.mcp.authentication`**

The **`authentication`** JSON object contains the following fields:

Value	Format	Description	Required
oauth2	JSON		N
blockchain	JSON		N

5.1.2.5 JSON Format: **`dataUrl.artifacts`**

`artifacts` is a JSON map keyed by the value of **`artifactDid`** (see Appendix A). It carries integrity and distribution details for any verifiable payload referenced from **`dataUrl.platforms`**.

Key format: **`did:artifact:<cidv1>`** (CIDv1 base32 of the artifact bytes)

The most prevalent use is for downloadable binaries as it allows clients to check the legitimacy of a binary before downloading it. This object reflects supply-chain lessons baked into Apple notarisation, Windows Authenticode, Sigstore, and SLSA.

The following table shows the common fields in each object in the mapping.

Value	Format	Description	Required
type	string	"binary", "container", or "website"	Y
sriManifest	JSON	See Appendix B	N
os	string	“windows”, “macos”, “linux”	Y (except website)
architecture	string	“x64”, “arm64”	Y (except website)
mimeType	string		N
downloadUrls	[string]	Array of download URLs	N
sizeBytes	Integer		N
provenanceUrl	URL	SLSA provenance JSON	N
signatureUrls	[JSON]	Example: [{“sigUrl”:”blah”, “sigAlgorithm”:”pgp/x509”}]	N
otherHashes	[JSON]	Example: [{“hash”:”blah”,”algo”:”sha256”}]	N
notarization	TBD		N
transparencyLog	JSON	See below	N

JSON example for the **`artifacts`** field:

JSON

```
{
  "did:artifact:<cidv1>": {
    "type": "binary",
    "os": "windows",
    "architecture": "x64",
    "mimeType": "application/vnd.microsoft.installer",
    "downloadUrls": [
      "https://dl.oma3.dev/pkg/app-v2-windows.exe"
    ],
    "sizeBytes": 1582034,
    "provenanceUrl": "https://dl.oma3.dev/pkg/app-v2-slsa.json",
    "signatureUrls": [
      {
        "sigUrl":
"https://dl.oma3.dev/pkg/app-v2-windows.exe.sig",
        "sigAlgorithm": "x509"
      }
    ],
    "otherHashes": [
      {
        "hash": "e05ac4bb...",
        "algo": "blake3"
      },
      {
        "hash": "8b7e1a...",
        "algo": "sha512"
      }
    ],
    "transparencyLog": {
      "type": "rekor",
      "entryId": "sha256:bf33..."
    }
  }
}
```

5.1.3 Metadata Confirmation

OMATrust uses several mechanisms to confirm the validity of data stored in app tokens. Some of these mechanisms are performed by the registry itself, and other mechanisms are performed by the client.

5.1.3.1 **did** Confirmation

OMATrust requires that the developer minting the NFT controls the **did**. Although this requires more effort from the developer, it results in higher trust in the system. **did** ownership confirmation consists of the following steps:

1. The app owner proves to an approved Issuer that it owns the **did**. See below for proof mechanisms based on the format of the ID.
2. Issuer submits an attestation to the Resolver contract (Section 5.2).
3. Registry contract calls the Resolver contract. If the proper attestations are in place, the token is registered. If not, the registration fails. See Section 5.1.6 for how conflicts, TTL, and revocations are handled.

Clients MAY also check ownership of **did** and other metadata values. The following sections describe how ownership is proven for various types of IDs.

Note: Ownership confirmation occurs at mint time. However, control of a contract or token can change later (for example, a proxy admin may be transferred, or a mint authority may be revoked). Clients that depend on ongoing correctness SHOULD re-verify ownership through the Resolver (see Section 5.1.6) if these changes are relevant to their use case.

5.1.3.1.1 **did:web** Confirmation

If the format of **did** is did:web, the owner MUST support the did:web Method Specification and return a JSON object at the following URL, where url is the URL specified in the **did**: **[url]/.well-known/did.json.**

This object is called a DID Document. Here is an example DID Document:

```json
{
  "@context": [
    "https://www.w3.org/ns/did/v1",
```

```
    "https://w3id.org/security/suites/secp256k1recovery-2020/v2"
  ],
  "id": "did:web:example.com",
  "verificationMethod": [
    {
      "id": "did:web:example.com#app-owner-key",
      "type": "EcdsaSecp256k1RecoveryMethod2020",
      "controller": "did:web:example.com",
      "blockchainAccountId":
"eip155:1:0x89a932207c485f85226d86f7cd486a89a24fcc12"
    }
  ],
  "authentication": [
    "did:web:example.com#app-owner-key"
  ]
}
```

Once this endpoint is implemented, the owner MUST get an attestation from an approved Issuer. The Issuer MUST retrieve the DID Document located at **`[url]/.well-known/did.json`** and verify that the owner address appears as an array element of the **`verificationMethod`** field in the DID Document returned by the endpoint.

The client MAY use this same method to verify the owner controls the **`did`**.

5.1.3.1.2 **`did:pkh`** Confirmation

This DID method is used to tokenize a smart contract application. Smart contracts do not need to be tokenized in order for users to file attestations on them. Attestations can be filed directly with an attestation service using the DID → Index Address Mapping method (see below). Tokenizing a smart contract is primarily for discovery and usage information.

If a smart contract is tokenized, the Issuer MUST confirm that the address minting the smart contract token is controlled by the same entity that either administered or deployed the contract. For EVM contracts:

- If the contract has an admin, the admin control address MUST be used.
- If the contract is immutable, the deploying address MUST be used.

Verification can be done as follows:

1. Determine the controlling address of the contract
    - Upgradeable contracts:
        - Inspect the contract's metadata or storage to locate the address that holds upgrade authority.
        - If an intermediate controller contract exists (e.g., an admin module), resolve the controlling address from that contract.
    - Contracts with explicit owner/admin roles:
        - Query the contract for the address associated with its owner or administrative role.
    - Immutable contracts:
        - If no admin or owner role is detectable, the deploying account from the creation transaction is the controlling address.
          (Example: On Ethereum, this may be read from well-known storage slots such as **`eip1967.proxy.admin`**, or via calls to functions like **`owner()`** or role checks in **`AccessControl`**.)
2. Compare with the minter of the registry token
    - The address that minted the registry token MUST equal the controlling address derived above.
    - If they match, ownership is verified.
3. Fallback: attestation path
    - If the registry token is to be minted by a different address (e.g., a multisig, DAO, or delegated key) than the controlling address, the Issuer MUST make an attestation binding the minting address to the controlling address (Section 5.3.5.1).
    - Clients SHOULD accept such an attestation only if it is issued by a trusted Issuer.
4. Failure to verify
    - If no match is found and no valid attestation is present, clients MUST treat the tokenized contract as unverified.

For non-EVM contracts, the Issuer MUST use the appropriate verification mechanism for the appropriate non-EVM virtual machine.

5.1.3.2 **`contractId`** and **`fungibleTokenId`** Confirmation

These CAIP-10 fields SHOULD be confirmed by the client using the same mechanisms the Issuer follows when confirming a did:pkh DID as described in Section 5.1.3.1.2.

5.1.3.3 **`dataURL`** Confirmation

5.1.3.3.1 **`dataHash`** Check

Clients MUST fetch **`dataUrl`**, canonicalize the returned JSON (see 5.1.3.4.2), compute the digest of the the resulting UTF-8 bytes using the on-chain **`dataHashAlgorithm`** (see 5.1.3.4.3), and compare it to the on-chain **`dataHash`** value. If they differ, clients MUST treat the manifest as unverified and SHOULD display a warning or hide the app according to UI policy

5.1.3.3.2 **`dataURL`** Ownership

The JSON returned by the **`dataUrl`** API endpoint MUST contain the **`owner`** field, and its value MUST match the NFT owner address. Furthermore, the owner SHOULD ensure the existence of a trusted third-party attestation that verifies certain dataUrl fields such as:

- name
- publisher
- summary
- description
- image
- screenshotUrls
- videoUrls
- 3dAssetUrls
- external_url

5.1.3.3.3 Other URL Confirmation

The dataUrl object could contain URL fields. The client MAY confirm the ownership of these URLs as well using one of the following mechanisms:

- URLs that use a domain that has already been verified in Section 5.1.3.1.1 or Section 5.1.3.3.2 do not need to be re-verified.

- All other URLs MAY be verified using the same mechanism to verify did:web DIDs or by checking a Linked Identifier attestation (Section 5.3.5.1).

For URLs that point to media:

- Clients SHOULD place higher trust in content addressable URLs such as IPFS or Filecoin URLs, as the content in these URLs cannot be changed without changing the URL.

- If the URL returns a media file, the media file MAY have the owner address embedded in the file in some manner.

5.1.3.4 JSON Policies

5.1.3.4.1 JSON Parsing

Parsing JSON objects MUST conform to RFC 8259 (ECMA-404). Inputs that contain comments, single quotes, NaN/Infinity, trailing commas, or other non-standard extensions MUST be rejected.

5.1.3.4.2 JSON Canonicalization

To ensure deterministic **`dataHash`** values across implementations, the **`dataUrl`** MUST be canonicalized using JCS (RFC 8785). Canonicalization (JCS, RFC 8785) summary:

- Object member names are sorted lexicographically (by Unicode code point).
- Objects are emitted in sorted order; array element order is preserved.
- Numbers are emitted in their minimal form (no leading zeros, no superfluous decimal points or exponents; −0 normalizes to 0).
- Strings are escaped exactly as specified in RFC 8785; no additional Unicode normalization is applied.
- All insignificant whitespace is removed.

5.1.3.4.3 JSON Hashing

`dataHash = HASH(canonicalUtf8Bytes)` where **`HASH`** is the algorithm specified in **`dataHashAlgorithm`**. The algorithm used MUST be recorded in **`dataHashAlgorithm`** and the digest MUST be encoded as a 0x-prefixed lowercase hex string.

5.1.3.4.3 Other Guidance

- Prefer integers and strings over floats where precision matters to avoid cross-runtime number formatting issues.
- Implementers SHOULD publish golden test vectors for canonicalization+hashing (covering numbers, escape sequences, object key ordering, and nested structures) and verify cross-runtime determinism to ensure consistency across implementations. Clients are encouraged to use these vectors to validate their consumption of OMATrust data. See Appendix D for example test vectors to guide implementation.

Note: Contracts do not validate JCS; they only store the algorithm + digest. JCS compliance and hash correctness are enforced off-chain by clients and indexers.

5.1.4 Control Policy

App Registry contracts MUST manage when version updates are required and when new NFT mints are required. These policies are based on policies of existing app stores that have years of experience mitigating fraudulent behavior.

- Each DID requires a different token in the Registry. It is the equivalent of Apple's Bundle ID.
- Different major versions of a specific DID also require a new token.

The following table details versioning rules for certain onchain fields.

Desired Change	On-chain rule
Move from **`(did, major)`** → **`(did, major+i)`**	**Must mint a new NFT**
Edit **`interfaces`**	Interfaces change requires **`minor+i`** and must be additive only
Edit **`dataUrl`** or **`traitHashes`**	Requires **`patch+i`** or **`minor+i`**
Edit **`fungibleTokenId`**	Must mint a new DID
Edit **`contractId`**	Not allowed
Transfer NFT ownership	Allowed without version changes

Table 3: Allowed DID methods for Application Registry DID field.

Control Policy Justification

- Apple's App Store ties immutability to the Bundle ID, forbidding any Bundle ID change once an app is live, so users always know they're running the same canonical app. Google Play enforces the same rule on the Android Package Name, requiring publishers to create an entirely new listing if they change it.
- Swapping the fungibleTokenId on-chain is equivalent to issuing an infinite-mint ERC-20 rug pull, so we freeze that field and demand a new DID/NFT to alter it.

- Because CAIP-19 encodes the contract address in the asset ID, any breaking executable change must mint a new NFT keyed by (did,major) to maintain deterministic asset lookups.
- Non-breaking API additions only require a minor bump in version number, signaling backward compatibility, while metadata or binary tweaks controlled by dataHash require only a patch bump.

5.1.5 Optional Soulbound Mode

When enabled, transfers and approvals MUST be rejected; minting and burning remain allowed.

API impact: No new fields are required (mode can be implicit or a boolean per token), but behavior MUST be observable via a read method, e.g., isSoulbound(tokenId).

UI guidance: Stores SHOULD visually label soulbound apps.

5.1.6 App Registry API

The Application Registry API allows developers to access the Application Registry. The API provides native blockchain JSON-RPC smart contract APIs (recommended) and a web2 API. The OMATrust Registry contract supports the following functions:

- **`mint`**: mints an Application.
- **`updateStatus`**: updates the status of an Application.
- **`getAppsByStatus`**: gets all the Applications (with pagination) of a certain status.
- **`getApps`**: gets all Applications with ACTIVE status.
- **`getAppDIDsByStatus`**: returns a list of Application DIDs of a certain status.
- **`getAppDIDs`**: returns a list of Application DIDs with ACTIVE status.
- **`getAppsByMinter`**: gets all the Applications minted by an address.
- **`getApp`**: returns an Application object given a DID.
- **`getDIDDocument`**: returns an Application is DID Document format given a DID.

To put data on the Application Registry a developer must send a signed transaction to the Application Registry's **`mint`** smart contract function through a native RFC endpoint of the contract (most often through an OMA3 front end website). The transaction MUST include the required parameters in Table 1. The transaction MAY include one or more of the optional parameters in Table 1.

The Application Registry DID standard API returns a DID Document using a DID Resolver. Here's an example of a DID Resolver request to a REST API using curl asking for a DID Document given an Application DID:

Shell

```
curl -X GET
https://appresolver.oma3.org/1.0/identifiers/did:web:upland.
me
```

The request returns the following DID Document:

JSON

```
{
  "@context": [
    "https://www.w3.org/ns/did/v1"
  ],
  "id": "did:web:upland.me",
  "verificationMethod": [
    {
      "id": "did:web:upland.me",
      "type": "EcdsaSecp256k1VerificationKey2019",
      "controller": "did:web:123456789abcdef",
      "publicKeyBase58": "04a34b8b56cf4d... "
    }
  ],
  "authentication": [
    "did:ethr:0x123456789abcdef#key-2"
  ],
  "service": [
    {
      "id": "did:example:123456789abcdef#metadata",
      "type": "MetadataService",
      "serviceEndpoint": {
        "name": "Upland",
        "version": 1.0,
        "icon-url": "https://upland.me/icon",
        "status": "active",
        "iwps-portal-url": "https://upland.me/iwps-portal-api",
        "description": "A web3 property game",
```

```
        "screenshots": [
          "https://example.com/screenshot1.png",
          "https://example.com/screenshot2.png"
        ],
        "owner-address": "did:ons:upland.realm",
        "application-token": "8g82898g98234-ba..."
      }
    }
  ]
}
```

Here's an example of the same resolver request to an EVM JSON-RPC node at localhost 127.0.0.1 assuming the Application Resolver smart contract is located at address **`0x9fE46736679d2D9a65F0992F2272dE9f3c7fa6e0`** using curl:

Shell

```
curl -H "Content-Type: application/json" -X POST --data
'{"jsonrpc":"2.0","method":"eth_call","params":[0:{"to":"0x9fE467
36679d2D9a65F0992F2272dE9f3c7fa6e0","value":"did:web:upland.me"},
1:"latest"]}' 127.0.0.1:8545
```

The request returns the same DID Document as the REST API call.

Reputation Integration: TBD

5.2 Ownership Resolver Contract

DID ownership is verified at token minting time. OMA3 uses a dedicated Resolver contract to confirm ownership and arbitrate conflicts when multiple parties claim to own the same DID.

Process

1. **First Attestation:** The owner of a DID gets an attestation from an approved Issuer to that confirms ownership of the DID. This confirmation process can be manual (e.g.- in

conjunction with a cybersecurity audit) or automatic (e.g.- a server that checks **`.well-known/did.json`** programmatically). The Resolver contract holds the attestation onchain.

2. **First Mint:** With the ownership attestation in place, the owner mints the application with the wallet address in the attestation. The Resolver contract checks the attestation before confirming the mint.

3. **Challenge:** A challenger may attempt to rebind a DID by minting the same DID/version combination.
    - The challenger MUST either ask the original Issuer to reverse the attestation (issuing a new attestation) or enlist at least two other approved issuers to attest to the challenger's ownership of the DID in question. The Resolver compares the challenger's attestation score against the incumbent's.
    - Scores are based on the count of valid, non-revoked attestations from an approved list of issuers.
    - Only attestations older than a global maturation delay (e.g., 72h) count toward scores, creating a rolling challenge window.

4. **Resolution:** If the number of challenger attestations strictly exceed incumbent's, ownership flips. Otherwise, the challenge fails.

Attestation model

- Attestations are EIP-712 signed by approved attesters (starting with the OMA3 verification server).
- Each *(***`attester, did, epoch, claimer`***)* has only one active attestation, which can be updated or revoked.
- Events record every **`AttestationUpdated`***,* **`ConflictAttempted`**, and **`OwnershipChanged`**.

Notification

- Incumbents receive notice via events; off-chain watchers (or push notification services) subscribe to these logs.
- An optional notifier callback can be registered by incumbents for on-chain hooks.

This model ensures predictable resolution without upgradeable registries, while allowing flexible evolution of conflict policy.

5.3 Reputation Service

A permissionless app registry without third party attestations leaves users open to fraud. Adversaries can register malicious, fraudulent, misleading, and counterfeit apps. To address this problem OMATrust leverages the Reputation System.

5.3.1 OMATrust Reputation System

The OMATrust Reputation System leverages and augments existing services like Ethereum Attestation Service. It is comprised of the following components:

1. Attestation Schemas: OMA3 defines several schemas for different attestations, from user reviews to cybersecurity certifications. These are listed below.

2. Cross Chain Addresses: Instead of using chain-specific blockchain addresses to identify the attestation subject, OMATrust uses hashed DIDs in the same format as addresses (see DID → Index Address Mapping below), which support web domains as well as blockchain addresses.

3. Resolver: the Resolver contract stores onchain attestations related to ownership, as described above.

5.3.2 DID → Index Address Mapping and Searching

To enable efficient per-DID discovery in contexts where a Solidity address is used as an index key (including the **`recipient`** field in EAS), OMATrust defines a deterministic mapping from a DID to an Index Address.

Computing the Index Address:

For any **`did`**, the corresponding Index Address is derived as follows:

```
None
library DidIndex {
    /// @notice Compute the DID Index Address used for EAS recipient or other
address-keyed indexes.
    /// @dev didHash = keccak256(canonicalizeDID(did))
    function toAddress(bytes32 didHash) internal pure returns (address) {
        // Domain-separated, versioned prefix for portability and clarity.
        bytes32 h = keccak256(abi.encodePacked("DID:Solidity:Address:v1:",
didHash));
        return address(uint160(uint256(h)));
    }
```

```
}
```

Where:

- **`canonicalizeDID(did)`** applies normalization rules defined by the DID method. For example:
  - **did:web**: lowercase the host, apply IDNA/punycode for international domains, preserve the path.
  - **did:pkh**: use the canonical chain/account encoding.
- **`didHash`** is the keccak256 digest of the canonicalized DID string.

Design Rationale:

- **Domain separation & versioning:** The ASCII prefix **`DID:Solidity:Address:v1:`** ensures compatibility and clear separation from other schemes. Future revisions can use updated prefixes (e.g., **`…:v2:`**).
- **Portability:** The Index Address can be used anywhere an **`address`** index is expected—EAS recipients, event partition keys, or contract mappings.
- **Semantics:** The Index Address is a label for indexing only. It is not a wallet and does not imply control or ownership.
- **Collision risk:** Negligible ($\approx 1 / 2^{160}$). The prefix further reduces overlap with unrelated mappings.

5.3.3 Attestation Querying

Clients can retrieve attestations related to a DID by computing its Index Address and filtering attestations accordingly.

Example query flow:

1. Compute **`indexAddress(did)`** using the algorithm above.
2. Query EAS for attestations with **`recipient = indexAddress(did)`** and a given schema UID (e.g., **`Oma3UserReview@1`**).
3. Within each attestation payload, confirm that **`subjectDidHash`** matches the DID hash used to derive the recipient. This prevents mismatches or spoofing.

5.3.4 EAS Integration: Recipient Rule

When storing an attestation about a DID in EAS:

- The **`recipient`** field MUST equal the computed **`indexAddress(did)`**.
- The attestation payload MUST include **`subjectDidHash`**, which is exactly the **`didHash`** derived during Index Address computation (see Section 5.1.3.4.3)

Example Schema: **`Oma3UserReview@1`**

A review attestation payload might be structured as:

```
None
struct UserReviewPayload {
    uint8 rating; // Rating: 0–100 (basis points) or scaled 0–5 × 20
    bytes32 contentHash; // Hash or CID digest of the review content
    bytes5 locale; // Optional: BCP-47 locale code (e.g., "en-US")
    uint16 version; // Optional: schema/content version indicator
    bytes32 subjectDidHash; // MUST equal the didHash used in recip...
}
```

Discovery Pattern

To fetch reviews about a DID:

```
None
address recipient = indexAddress(did);
// Query EAS for attestations where:
// schemaUID == Oma3UserReview@1
// recipient == indexAddress(did)
```

Then decode the attestation payloads and resolve **`contentHash`** for off-chain content when needed.

Clients can retrieve attestations related to a DID by computing its Index Address and filtering attestations accordingly.

5.3.5 Schema Definitions

5.3.5.1 Linked Identifier Schema

This schema provides additional attestations on control of an ID by another ID. For example, attesting that the entity that controls one blockchain address also controls a different blockchain address. Examples of IDs that can be entered into a Linked Identifier attestation:

- DID: DIDs need an attestation signed by the wallet that is associated with the DID.
- Blockchain address: Similar to DIDs, blockchain addresses need an attestation signed by the blockchain address.
- Web3 names: web3 domains need an attestation that the developer owns the wallet address that controls the web3 domain.
- URL: URLs need an attestation of a third party that uses a mechanism to prove the developer controls the domain.

In some cases the Issuer of an attestation can be automated. It is easy for software to get signatures from two different blockchain accounts to prove common control. Others, such as offchain IDs like passport numbers, may require manual confirmation.

Below are some Linked Identifier schema field values and verification processes that Issuers could implement before issuing a Linked Identifier attestation.

Field	Value	Description
Issuer ID	Wallet address	ID of the attester/Issuer
Holder ID	DID, wallet address, or Web3 domain	ID of the Application
Linked ID	Wallet address	Wallet address of Application owner
AttestationDate	block.timestamp	
Method	enum	Method used to verify the metadata

For verifying that a Holder ID wallet address also controls a URL domain, a third party Issuer could follow this procedure, which is similar to proving domain ownership for a website SSL certificate:

1. Developer goes to the Issuer's website and enters the domain they are trying to verify.
2. Developer uses the website to sign a transaction with the same wallet they used with the Application Registry.
3. Issuer website gives developer a random value.
4. Developer stores the value in the domain's DNS TXT Record.
5. Developer tells the Issuer website to check the DNS records.

6. If the Issuer finds records contain the value, the Issuer sends a signed transaction attestation to the blockchain with the above format.

To bind a did:web identifier to a blockchain wallet address, the schema requires an attestation that proves control of both:

- Holder ID: **`did:web:example.com`**
- Linked ID: **`eip155:1:0xabc...`**

Verification flow:

1. The developer serves a DID document at **`https://example.com/.well-known/did.json`** containing a **`verificationMethod`** entry with **`blockchainAccountId`** equal to the wallet address.

2. The developer signs a challenge message with that wallet.

3. The Issuer (e.g., OMA3 verification server) validates both:

   - the DID document includes the wallet address, and
   - the wallet produced a valid signature.

4. The Issuer then issues a Linked Identifier attestation with fields:

   - Issuer ID: attester wallet
   - Holder ID: did:web:example.com
   - Linked ID: eip155:1:0xabc...
   - AttestationDate: block.timestamp
   - Method: didWebToWalletSignature

Revocation/updates: A new attestation overwrites the old one, or the Issuer can revoke it explicitly.

Usage: Clients and registries SHOULD only consider a did:web binding valid if a live Linked Identifier attestation exists proving control of both identifiers.

5.3.5.2 DataURL Attestation Schema

Every app token stores two on-chain pointers to its mutable metadata:

Slot	Purpose

dataUrl (string)	HTTP / IPFS endpoint that returns the JSON manifest for this token.
dataHash (bytes32)	Hash of the JCS-canonicalized bytes of the JSON served at dataUrl.

To prevent silent or misleading edits, each manifest must be covered by a third-party attestation whose schema includes the same **`dataHash`**. The three artefacts work together as follows:

Step	Actor	Action
1. Publish manifest	App developer	Upload the JSON to **`dataUrl`**; compute **`dataHash = hash(bytes(JSON))`**.
2. Request attestation	Human reviewer (attester)	Manually inspect the JSON (name, publisher, screenshots, etc.). • If compliant, issue an EAS-compatible attestation: **`{tokenId, dataHash, schema: NAME_COMPLIANCE, issuedAt}`**
3. Register hash	App developer	Call **`updateMetadata(tokenId, dataUrl, dataHash);`** the contract rejects the call unless a matching attestation exists.
4. Client verification	Wallet / marketplace	a) Fetch **`dataUrl`** JSON. b) Compute **`hash(JSON)`** and compare it to on-chain **`dataHash`**. c) Confirm an attestation referencing that same hash is still valid. d) Only display the app if both checks pass.

Why this is secure

- Immutability guarantee – Any byte-level change (even a new screenshot) alters the hash; the old attestation no longer matches, so UIs hide the app until a new review is issued.

- Reviewer accountability – The attestation is signed by an address in the front-end's trusted-attester list; multiple independent attesters can coexist, avoiding central gate-keeping.

- Audit trail – Every approved version leaves an on-chain record (`dataHash` + attestationId), giving provable history for compliance audits.

Manual review requirements

A reviewer MUST verify at minimum:

1. Brand & trademark – No infringing or misleading app/publisher names.
2. Visual assets – Screenshots, icons, videos accurately represent the product.
3. Policy compliance – No illicit content, malware links, or disallowed keywords.

Once satisfied, the reviewer signs the attestation that binds *their* approval to the specific **`dataHash`**. Any later change triggers the whole cycle again, ensuring continuous trust in all mutable data while keeping the registry fully decentralised.

5.3.5.3 Endorsement Schema

An Endorsement attestation is a simple way for an organization to vouch for a DID subject.

5.3.5.4 Certification Schema

A certification attestation is a more structured type of endorsement that reflects the typical certification model used by certification bodies such as the Federal Trade Commission, Common Criteria, FIPS, Underwriter Laboratories, SunSpec Alliance, and countless others. In these programs there are four main actors:

- Product- the subject that is getting certification.
- Developer- the entity responsible for the product.
- Assessor- a trusted third party that analyzes the product against a requirements document and test procedure.
- Certification Body- an entity (such as an industry consortium or government body) that creates and administers the certification program.

The use case flow using a certification schema is as follows:

1. Developer signs agreement with the Assessor to test the Product for certification.
2. Assessor tests the product using documents ratified by the Certification Body.
3. Assessor sends test results to the Certification Body.
4. Certification Body reviews the test results. If it passes, it signs the Certification attestation transaction. If not, it sends the failure reasons to the Assessor.

5.3.5.5 User Review Schema

User reviews are critical for any app's reputation. The User Review attestation allows anyone to post a review on a DID.

The problem with permissionless reviews is that a developer can review their own app by spinning up multiple wallets. Users can also spam bad reviews on an app. The User Review attestation relies on KYC and KYB attestations as well as app stores requiring these attestations to prevent these issues from making user reviews meaningless. For example, an app store SHOULD only incorporate user review attestations created by wallets that have been verified (e.g.- KYC). Applications MAY also require uniqueness attestations (e.g.- Worldcoin) as well to prevent sybil attacks. See below for more app store guidance on leveraging attestations to protect users.

Specifications on the below schemas can be found in the OMA3 Reputation Service draft proposal.

5.3.6 Client Guidance

To ensure integrity and prevent user harm from fraudulent or misleading app metadata, all clients consuming data from OMATrust SHOULD adopt an attestation-based trust policy.
An app store SHOULD NOT display any application token unless the following conditions are met:

1. **`dataUrl`** resolves to valid JSON conforming to the current offchain schema.
2. The SHA-256 (or Keccak-256) hash of the fetched JSON exactly matches the on-chain **`dataHash`** for the token.
3. A valid third-party attestation has been published referencing that specific **`dataHash`** and using an accepted schema(s).

Each app store may define its own attestation trust model. For example:

Model	Description
Single attester	A centralized store operator acts as sole reviewer (like Apple).
Multi-attester	Accept attestations from any approved third party (e.g., Mozilla, OMA3, community DAOs).

AI + human fallback	Require attestation from an automated reviewer unless the confidence score is low, in which case fallback to a human attester.
Community attestations	Support attestations from public reviewers with onchain reputation systems.

If an application does not have a valid attestation:

- It SHOULD be hidden by default from public listings.
- It MAY be shown with a warning or behind a user-controlled setting (e.g., “Show unverified apps” toggle).
- It SHOULD NOT be eligible for ranking, recommendation, or featuring.

This approach balances decentralized publishing with user trust, enabling permissionless participation while minimizing abuse.

Change History

Version	Date	Comments
0.1	2025-09-25	Initial draft - Alfred Tom
0.2	2025-09-28	Clarified data confirmation mechanisms

Appendix A

did:artifact Specification

Provisional DID method used within this specification to identify verifiable payloads (binaries, containers, and website proof files such as SRI manifests or site snapshots). A standalone method spec and registry entry will follow.

A.1 Overview

`did:artifact` is a content-addressable identifier. The method-specific ID is a CIDv1 (multibase base32-lower) whose multihash encodes the hash algorithm and digest of the artifact's bytes.

V1 requirement: the multihash MUST be SHA-256 (32-byte digest- see Section 5.1.3.4.3).

- Binaries/containers: hash the file/image bytes.
- Websites: hash a proof artifact (e.g., JCS-canonicalized SRI manifest JSON, or a site snapshot archive).
- Each artifact gets its own DID; different files/proofs → different DIDs.

A.2 Identifier Syntax

```
None
did:artifact:<cidv1>
```

- **`<cidv1>`** MUST be CIDv1 encoded with multibase base32-lower.
- The CID's multihash MUST use SHA-256 under this spec version (5.1.3.4.3).
- The multicodec SHOULD be **`raw`** for opaque bytes. Using a more specific codec does not change verification semantics.

A.3 Computing `artifactDid`

Common procedure (all artifact types):

1. Obtain the exact artifact bytes (after any required canonicalization for that type).
2. Compute SHA-256 over those bytes (5.1.3.4.3).
3. Wrap as multihash (function code + length + digest).
4. Build **`CIDv1`** (multicodec **`raw`** unless specified otherwise).

5. Multibase-encode (base32-lower) → prepend **`did:artifact:`**.

Type notes:

- **Binary / Installer / Archive:** use bytes exactly as distributed (no repacking). Apply common procedure.
- **Container (OCI):** use the OCI image manifest bytes as stored in the registry (**`application/vnd.oci.image.manifest.v1+json`**). Apply common procedure.
- **Website SRI manifest (JSON):** canonicalize with JCS (JSON Canonicalization Scheme), then apply common procedure to the canonical UTF-8 bytes.
    - Paths in the manifest MUST be same-origin absolute paths (no query/fragment). Producer normalization: single percent-decode, Unicode NFC, collapse `//`, strip trailing `/` (except root), preserve case.
- **Website snapshot (archive):** create a deterministic archive when possible (fixed owner/mode/timestamps). Apply common procedure to the archive bytes.

A.4 Verification (Client Requirements)

When a record references an **`artifactDid`**, verifiers MUST:

1. Fetch bytes from any location (HTTP(S), ipfs://, local, etc.).
2. Recompute CIDv1(SHA-256- Section 5.1.3.4.3) per A.3.
3. Require equality with the referenced **`artifactDid`**. If different → invalid, regardless of URL or signature.

Websites:

- If the artifact is an SRI manifest, verifiers SHOULD also validate loaded assets against the per-file SRI hashes.
- If the artifact is a snapshot, verifiers MAY present a “verified snapshot” badge when the content matches.

No trust from URLs. Links are advisory for discovery/distribution only.

A.5 Data Model Integration (normative hooks)

- **`dataUrl.platforms`**
    - **`launchUrl`** (required) is the user-facing deep link/store page.
    - **`artifactDid`** (optional) SHOULD be present when a platform ships verifiable bytes (desktop installer/CLI/container) or a website proof artifact.
    - If **`artifactDid`** is present, a matching entry MUST exist at **`dataUrl.artifacts[artifactDid]`**. See A.6.

- **`dataUrl.artifacts`** (map keyed by **`artifactDid`**)
    - Key: **`did:artifact:<cidv1>`** per this appendix.
    - Type-specific:
        - binary/container: **`os`**, **`arch`**, optional **`libc`**, **`variant`**; container may include **`ociDigest`**.
        - website: **`originDid`** (**`did:web:<apex>`**), and either inline **`sriManifest`** (the canonicalized JSON used for hashing) **or** pointers **`manifestDid`**/**`snapshotDid`** (both **`did:artifact:*`**).

A.6 Policy (V1)

- **Allowed hash:** only SHA-256 is permitted for producing **`did:artifact`** values under this spec version. Verifiers MUST read the multihash algorithm but MUST reject non-permitted algorithms.
- **Website scope:** website proof artifacts MUST be scoped to an apex origin; cross-origin redirects MUST NOT be followed during attestation/verification.
- **Caching:** verifiers MAY cache computed CIDs; any new download MUST be re-verified.

A.7 Security Considerations

- **Redirect/mirror safety:** content identity derives solely from **`artifactDid`**; mirrors/CDNs are acceptable.
- **Dynamic content:** SRI manifests cover only listed assets; non-listed dynamic responses are **out of scope** and should be treated as unverified.
- **Determinism:** prefer deterministic packaging to avoid unintentional hash churn.
- **Keyed proofs:** signatures/SBOM/provenance strengthen trust but do not replace byte-level verification against **`artifactDid`**.
- **DID Index Address semantics**: `indexAddress(did)` (see §{#did-index-address}) is a deterministic **indexing label** for discovery and partitioning, not proof of control. Do **not** interpret it as a signer/owner; never send assets to it. The chance a real EOA equals this address is negligible (≈ 1 / 2^160), and the versioned prefix prevents cross-scheme overlap.

A.8 DID Index Address Helper

Reference Solidity helper for computing the Index Address from a `didHash`:

None

```
library DidIndex {
    /// @notice Compute the DID Index Address used for EAS recipient or
other address-keyed indexes.
    /// @dev didHash = keccak256(canonicalizeDID(did))
    function toAddress(bytes32 didHash) internal pure returns (address)
{
        // Domain-separated, versioned prefix for portability and
clarity.
        bytes32 h =
keccak256(abi.encodePacked("DID:Solidity:Address:v1:", didHash));
        return address(uint160(uint256(h)));
    }
}
```

Safety: The returned address is an *index label*, not a controller. Never infer control or send assets to it. Always include **`subjectDidHash`** in payloads and verify consistency with the derived **`recipient`**.

A.9 Forthcoming Method Registration (informative)

This appendix will be extracted into a standalone **`did:artifact`** method specification and registered in the W3C DID Spec Registries. The standalone spec will define a minimal DID Document and JSON-LD context (e.g., **`https://w3id.org/did-artifact/v1`**). Until then, this appendix is the normative definition for use within this specification.

A.10 Examples (informative)

Binary (Windows installer)

JSON

```
"platforms": {
  "windows": {
    "launchUrl": "myapp://open",
    "preferredDownloadUrl":
"https://cdn.example.com/AppSetup.exe",
```

```
    "artifactDid": "did:artifact:<cidv1>"
  }
},
"artifacts": {
  "did:artifact:<cidv1>": {
    "type": "binary",
    "os": "windows", "arch": "x64",
    "sizeBytes": 73400320,
    "distributionURIs": [
      "https://cdn.example.com/AppSetup.exe",
      "ipfs://<cidv1>"
    ],
    "signatureURIs": ["https://sig.example.com/win.sig"],
    "algo": "sha256",
    "digestHex": "<sha256-hex>"
  }
}
```

Website (SRI manifest)

JSON

```
"platforms": {
  "web": {
    "launchUrl": "https://play.example.com",
    "artifactDid": "did:artifact:<cidv1-of-jcs-sri-manifest>"
  }
},
"artifacts": {
  "did:artifact:<cidv1-of-jcs-sri-manifest>": {
    "type": "website",
    "originDid": "did:web:example.com",
    "sriManifest": {
      "version": 1,
      "origin": "example.com",
      "algo": "sha512",
```

```
      "hashEncoding": "base64",
      "paths": {
        "/assets/app.css": "sha512-…",
        "/assets/app.js":  "sha512-…"
      }
    }
  }
}
```

Editor’s note (remove before publishing)

- Add “See Appendix A.2” references next to **`artifactDid`** mentions in **`dataUrl.platforms`** and **`dataUrl.artifacts`**.
- When you create the external repo, move this appendix verbatim, add test vectors, and link back here.

Appendix B

Website Artifacts and SRI Manifests Draft Notes

This appendix captures design intent for website verification to be finalized in future version of the specification. It is for informational purposes only.

B.1 Summary (design intent)

- Website proofs will use **`artifactDid = did:artifact:<cidv1>`** (same as binaries).
- The website proof artifact will be an SRI manifest (JSON) whose JCS-canonicalized bytes are hashed (SHA-256 → CIDv1) to form the **`artifactDid`**.
- **`platforms.web.launchUrl`** remains the UX entry point; **`platforms.web.artifactDid`** (optional for v1) can point to the website proof artifact when available.

B.2 Planned artifact shape (non-normative)

JSON

```
"artifacts": {
  "did:artifact:<cidv1-of-sri-manifest>": {
    "type": "website",
    "sriManifest": {
      "version": 1,
      "origin": "example.com",        // apex domain
      "algo": "sha512",
      "hashEncoding": "base64",
      "paths": { "/assets/app.js": "sha512-..." }
    },
    "downloadUrls": ["ipfs://<cidv1-of-sri-manifest>"] // optional
  }
}
```

Notes:

- The **`origin`** inside the manifest binds it to the domain and is part of the hashed bytes.

- No separate **`originDid`** is required in v1 draft; the client already knows the domain via **`platforms.web.launchUrl`**.

B.3 Planned verification algorithm

1. Resolve **`artifactDid`** from **`platforms.web.artifactDid`**.
2. Obtain manifest bytes (inline **`sriManifest`** or fetch via **`downloadUrls`**).
3. **JCS** canonicalize → **SHA-256** → **CIDv1**; require equality with **`artifactDid`**.
4. Ensure **`registrableDomain(platforms.web.launchUrl) == sriManifest.origin`**.
5. When rendering the site, validate loaded assets against manifest SRI entries.

B.4 Open items (track in issues)

- Deterministic packaging & test vectors for SRI manifests (JCS inputs/outputs).
- Handling multi-origin assets (likely out of scope for v1 web verification).
- Optional “snapshot” artifact (tar/warc) and how to embed origin metadata inside the archive if added later.
- UX guidance for partial verification (some assets verified, others not).
- Attestation linkage: publisher → domain (**`ProofOfWebControl`**) vs. publisher → artifact (**`ArtifactBinding`**).

Appendix C

Recommended Trait Hashes

This appendix provides a list of recommended trait strings for use in the on-chain `traitHashes` array. For the onchain field, developers must hash these values with keccak256. Additional traits may be proposed via OMA3 governance processes.

Trait String	Description
"api:openapi"	Include this if the interface field has a value of 2 and the API format is OpenAPI.
"api:graphql"	Include this if the interface field has a value of 2 and the API format is GraphQL.
“api:jsonrpc”	Include this if the interface field has a value of 2 and the API format is JSON-RPC.
"api:mcp"	Include this if the interface field has a value of 2 and the API format is OpenAPI.
"api:a2a"	Include this if the interface field has a value of 2 and the API format is A2A.
"token:erc20"	Include this if the token in the fungibleTokenId field supports ERC-20.
“token:erc3009”	Include this if the token in the fungibleTokenId field supports ERC-3009.
“token:spl”	Include this if the token in the fungibleTokenId field supports SPL.
“token:2022	Include this if the token in the fungibleTokenId field supports Token-2022.
"token:transferable"	Include this if the token in the fungibleTokenId is transferable.
"token:burnable"	Include this if the token in the fungibleTokenId is burnable.
“pay:x402”	Include this if the endpoint supports x402 payments.

Appendix D

Example Test Vectors for Canonicalization+Hashing

This appendix provides example test vectors to guide implementers in ensuring correct canonicalization and hashing of JSON data for the `dataHash` field. Implementers SHOULD compute hashes using the algorithm specified in `dataHashAlgorithm` (e.g., sha256 or keccak256) and verify consistency across runtimes (e.g., Node.js, Python, Solidity). Clients may use these vectors to validate their implementations. Additional vectors should be published by implementers to cover edge cases.

Input JSON	Canonicalized JSON	Expected Hash (Compute per dataHashAlgorithm)
{"a": 1.0, "b": {"c": "\n"}}	{"a":1,"b":{"c":"\n"}}	[Compute Hash]
{"b":2,"a":1}	{"a":1,"b":2}	[Compute Hash]
{"x":[{"y":true}]}	{"x":[{"y":true}]}	[Compute Hash]