Authenticated Transfer
Internet-Draft
Intended status: Standards Track
Expires: 6 December 2026

D. Holmgren
B. Newbold
Bluesky Social

4 June 2026

Authenticated Transfer: Repository
draft-holmgren-at-repository-02

Abstract

This document specifies a repository data structure for storage and transfer of public user data records as part of the Authenticated Transfer Protocol (ATP). It describes encoding formats for both individual data records and entire repositories. The repository data structure is content-addressable and cryptographically authenticated.

About This Document

This note is to be removed before publishing as an RFC.

Status information for this document may be found at https://datatracker.ietf.org/doc/draft-holmgren-at-repository/.

Discussion of this document takes place on the Authenticated Transfer Working Group mailing list (mailto:atp@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/atp/. Subscribe at https://www.ietf.org/mailman/listinfo/atp/.

Source for this draft and an issue tracker can be found at https://github.com/ietf-wg-atp/drafts.

Status of This Memo

Table of Contents

1. Introduction

The Authenticated Transfer Protocol (ATP) enables the creation of decentralized networks for publication of self-certifying data. An introduction to the overall protocol architecture is given in [AT-ARCH].

User accounts publish structured data records to the network by including them in their public repository. Records within a repository are identified by a unique path and current content version (hash). Records can be created, updated, and deleted at any time. Repositories contain the complete set of current records for the account, and do not include or reveal the existence of previous content.

The repository structure includes the account's persistent identifier, and the overall repository structure is cryptographically signed. The authenticity of the entire repository can be verified by resolving the account identifier to the current public key. Data records are not signed individually. Details of account identifier systems and their resolution process are out of scope for this document.

Large binary data such as images and media files are not stored directly within repositories. Instead, such data is stored externally and referenced in records by a hash link.

Mechanisms for synchronizing repositories between parties over the public network are described in [AT-SYNC].

This document describes version 3 of the repository format.

2. Repository Semantics

Records within a repository are discrete units of structured data, identified by a unique path and versioned by content hash. Records conform to a generic data model, but the content, schema, and semantics of record is varied and application-specific. Records are grouped by type under "collections".

The current state of a repository is summarized in a signed "commit". Any change to the contents of the repository updates the current commit. Commits for an individual account's repository are serialized using a monotonically-increasing "revision" identifier.

Updates to repositories may include operations on multiple records in a batch mutation that results in a single signed commit. Implementations should apply practical limits on batch sizes to support efficient processing and distribution of repository changes.

2.1. Account Identifiers

Repository commit objects (Section 3.2) contain a persistent account identifier, which indicates the publisher of the repository. This account identifier can be resolved to obtain the current cryptographic public keys for the account, and those keys can be used to verify the authenticity of the repository and the records it contains.

The keys associated with an account may be rotated over time. The most recent commit must always be verifiable using the currently resolvable signing key. When rotating signing keys, a new repository commit must be created, even if the contents and structure of the repository remain unchanged.

This document does not include details or recommendations on account identifier systems.

2.2. Revisions

Repository commits include a revision field (rev) which acts as a logical clock for updates to the repository over time. The revision string is a Timestamp Identifier (TID) as described in Appendix C.

Revisions may be used when comparing two versions of a repository to determine which is more recent. This is particularly relevant when synchronizing repositories indirectly, or from multiple sources over time.

If a commit TID value corresponds to a timestamp in the future (beyond a short period to accommodate clock drift) the commit SHOULD be ignored. This is to ensure that a newly published commit (with a TID corresponding to the current time) will reliably be accepted as current by the entire network.

3. Repository Structure

Repositories are structured as a Merkle Search Tree (Section 4) with a cryptographically signed commit object referencing the tree root.

The MST structure provides several fundamental properties for repository operations. As a content-addressed structure, it enables efficient verification of data. The MST maintains lexicographic key ordering, enabling structural sharing of intermediate tree nodes for related records. It is probabilistically self-balancing, offering consistent performance characteristics. Additionally the MST exhibits unicity, meaning that any given set of keys and values will always produce the same tree structure and root hash regardless of insertion order.

Repository contents are encoded using deterministic CBOR serialization and organized as a directed acyclic graph where data objects reference each other through content hashes. These hash-identified data objects, referred to as "blocks," include three distinct types: commit objects, MST internal nodes, and data records.

3.1. Record Paths

Records within a repository are identified by a non-empty case-sensitive ASCII string called the "path". Records are stored sorted lexicographically by path, and the efficiency of some repository operations is impacted by sort order.

A path string is the combination of a collection type name and a record key, joined by a single forward slash character: <collection>/<record-key>. A path MUST consist of exactly two segments separated by /, with no leading or trailing slash.

Collection names use the Namespaced Identifier (NSID) syntax described in Appendix D. They have a prefix-ordered namespace structure, which means that records of the same collection are stored adjacently, and that collections under the same authority are grouped together.

Record keys uniquely identify records within a collection. Record keys are case-sensitive and MUST satisfy the following syntax:

* Allowed characters are ASCII alphanumerics (A-Z, a-z, 0-9), period (.), hyphen (-), underscore (_), colon (:), and tilde (~)

* Length between 1 and 512 characters (inclusive)

* The literal values . and .. are prohibited

The syntax of record keys may be constrained further on a per-collection basis at the application layer. A common choice is to use the Timestamp Identifier Appendix C syntax, which results in lexicographic sorting by time within a collection. This means that "new" records are all grouped together within a given collection.

Note that both the NSID and record key string syntaxes are valid path components as defined in Section 3.3 of [RFC3986]. It is important to maintain this property.

3.2. Commit Objects

Commit objects serve as the authoritative root of each repository, establishing cryptographic ownership and providing a verifiable reference to the state of a repository at a particular point in time. Each commit is digitally signed by the repository account owner and contains metadata necessary for verification.

A commit object contains the following data fields:

* *did* (string, required): The resolvable account identifier associated with the repository as described in Section 2.1

* *version* (integer, required): Repository format version, fixed value of *3* for the current specification

* *data* (cid-link, required): Hash pointer to the root of the repository’s MST structure

* *rev* (string, required): Repository revision identifier that functions as a logical clock and must increase monotonically (see Section 2.2). Syntax MUST match Appendix C.

* *prev* (cid-link, nullable): Optional pointer to the previous commit object in the repository’s history chain. While included for backward compatibility with version 2 repositories, this field is typically null in version 3 implementations

* *sig* (byte array, required): Cryptographic signature over the commit contents.

Commit objects are signed by the key declared by the repository owner’s resolvable identifier. Neither the signature nor the signed commit object contains information about the curve type or specific public key used for signing. This information must be obtained by resolving the account identifier as described in Section 2.1.

The procedure for signing commit objects:

1. Encode the unsigned commit object (with sig field entirely absent) as CBOR

2. Sign the encoded bytes as described in Appendix B

3. Include the signature bytes in the sig field

To verify the signature, remove the sig field and encode the unsigned commit object as CBOR. Then verify the signature against those encoded bytes.

3.3. Records

Records stored within a repository are always objects (or "maps") encoded as CBOR, following the data model and encoding rules described in Appendix A. Each record must include a top-level field named $type with a string value matching the collection type name (NSID) of the path that the record is stored at.

Invalid or corrupt data in individual records should not impact processing of the overall repository data structure, or the processing of other valid records in the same repository.

4. Merkle Search Tree

The Merkle Search Tree (MST) structure is deterministically reproducible from any given key-value mapping, where keys are non-empty byte strings (corresponding to a path) and values are hash link references to records. This deterministic construction ensures that identical input sets always produce the same root hash regardless of insertion order.

The tree’s structural organization depends solely on the keys present, not on the record values they reference. When a record value changes, the new content hash propagates up through the tree nodes to the root, but the tree’s shape and node organization remain unchanged.

The MST data structure was first published in [MSTPAPER].

4.1. Tree Structure

Each MST node contains a list of key-value entries and references to child subtrees. Entries and subtree links are maintained in lexicographic order, with all keys in a linked subtree falling within the range corresponding to that link’s position. The ordering proceeds from left (lexicographically first) to right (lexicographically last).

Keys are assigned to tree levels based on a layer value computed from the key itself. Nodes at each level contain all keys with the corresponding layer value, while subtree links point to nodes containing keys that fall within specific lexicographic ranges but have lower layer values. Adjacent keys may appear within the same node, but adjacent subtrees must be separated by at least one key entry to prevent structural ambiguity.

4.2. Layer Calculation

The layer for a given key is calculated using SHA-256 with a 2-bit grouping scheme that provides an average fanout of 4:

1. Compute the SHA-256 hash of the key (byte string) with binary output

2. Count the number of leading binary zeros in the hash

3. Divide by 2, rounding down to the nearest integer

Examples of layer calculation:

* key1: SHA-256 begins 100000010111... → layer 0

* key7: SHA-256 begins 000111100011... → layer 1

* key515: SHA-256 begins 000000000111... → layer 4

When processing the MST structure, implementations must verify the layer assignment and ordering of keys. While this verification is most essential for untrusted inputs, implementations should perform these checks consistently regardless of data source. Additional validation of node size limits and other structural parameters is required to prevent resource exhaustion attacks, as detailed in Security Considerations (Section 6).

4.3. MST Construction Example

The following is a Merkle Search Tree containing 9 records with keys A-I. Each key would include a pointer to some record hash, though that hash is irrelevant to the construction of the tree. Each asterisk (*) represents a hash pointer to the subtree under it.

For the sake of illustration assume the following layer calculations:

* layer(D) = 2

* layer(A|E|I) = 1

* layer(B|C|F|G|H) = 0

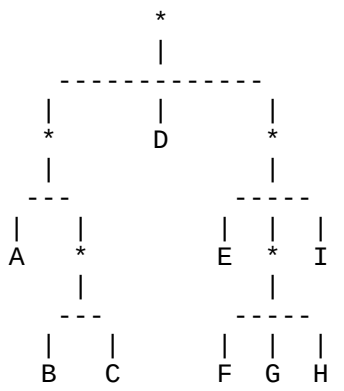

Figure 1: Example MST Structure

4.4. Empty Nodes

An empty repository containing no records is represented as a single MST node with no entries. This is the only case where a node without entries is permitted.

Nodes that contain no key entries but do contain subtree links are allowed at intermediate positions, provided those subtrees eventually contain key entries. However, such nodes MUST NOT appear at the root position — the root MUST either contain key entries or be the special case of a completely empty repository. Similarly, nodes without key entries MUST NOT appear at leaf positions except for the empty repository case.

This structure ensures that nodes lacking key-value entries are pruned from the top and bottom of the tree while preserving intermediate nodes that maintain proper height relationships and prevent subtree links from skipping layers.

4.5. MST Node Schema

Given their prevalence through the repository structure, MST nodes require a compact binary representation for storage efficiency. Keys within each node use prefix compression, where each entry specifies the number of bytes it shares with the preceding key in the array. The first entry in each node contains the complete key with a prefix length of zero. This compression applies only within individual nodes and does not extend across node boundaries. The compression scheme is mandatory to ensure deterministic MST structure across all implementations.

MST nodes contain the following fields:

* l (hash link, nullable): Reference to a subtree node at a lower layer containing keys that sort lexicographically before all keys in the current node

* e (array, required): Ordered array of entry objects, each containing:

  - p (integer, required): Number of bytes shared with the previous entry in this node

  - k (byte string, required): Key suffix remaining after removing the shared prefix bytes

  - v (hash link, required): Reference to the record data for this entry

  - t (hash link, nullable): Reference to a subtree node at a lower layer containing keys that sort after this entry's key but before the next entry's key in the current node

Hash references appearing within an MST node — the l and t subtree links, and the v record link — MUST use the constrained content-hash format defined in Appendix A.2.

4.6. MST Node example

The following example shows an MST node at layer 1 containing two subtree pointers and two key-value entries. The node contents in order are:

* Left subtree: hash link 0x01711220643b9326...

* Entry: key7 → record hash link 0x017112202d9aa87e...

* Right subtree: hash link 0x0171122047e2886f...

* Entry: key10 → record hash link 0x0171122010b6da2c...

This node would be encoded as follows:

```
{
  l: 0x01711220643b9326...
  e: [
    {
      p: 0,
      k: "key7",
      v: 0x017112202d9aa87e...
      t: 0x0171122047e2886f...
    },
    {
      p: 3,
      k: "10",
      v: 0x0171122010b6da2c...
      t: null
    }
  ]
}
```

5. Repository Serialization Format

Repositories are serialized for transmission and storage as a concatenated sequence of block data, where blocks represent the CBOR-encoded records, MST nodes, and commit objects that comprise the repository structure. The serialization is prefixed with a header that identifies the root block, typically the repository's commit object.

Serialized repositories may contain partial repository state, such as when transmitting cryptographic proofs for specific records. In these situations, they may not include unrelated MST nodes or records outside the proof path.

The block-and-header layout described here is compatible with Content-Addressable archive (CAR) formats such as [DASL-CAR].

5.1. Header Format

The header is constructed by CBOR-encoding an object with the following fields:

* version (integer, required): Fixed value of 1

* roots (array, required): Single-element array containing the hash link of the commit block

The CBOR-encoded header is prefixed with its byte length encoded as an unsigned LEB128 integer as described in Section 5.2.2 of [WEBASSEMBLY].

5.2. Block Format

Following the header, each repository block is serialized by concatenating:

1. The combined byte length of the following two components, encoded as an unsigned LEB128 integer

2. The block's content hash, prefixed with 0x01711220 as specified in Appendix A.2

3. The CBOR-encoded block data

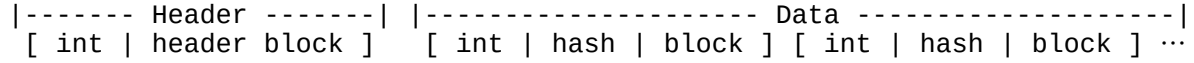

Figure 2: Repo Serialization Layout

5.3. Block Ordering

Producers SHOULD emit blocks in pre-order traversal of the included repository portion: header, commit object, root MST node, then a recursive depth-first interleaving of subtree nodes and the records they reference.

Preorder traversal enables streaming verification of repositories, allowing parsers to walk the MST structure and output key-to-record mappings while maintaining minimal MST state in memory. This approach supports efficient stream processing of large repositories without requiring complete buffering of the serialized data.

Parsers MUST tolerate other block orderings, duplicate occurrences of the same block, and additional unrelated blocks. Specifically:

* Duplicate blocks SHOULD be deduplicated rather than treated as an error.

* Dangling references — for example, hash links pointing to records or blobs that are not present in the serialized data — MAY be present and unresolvable; this is not an error in itself.

* Unrelated blocks not referenced by the repository structure SHOULD be ignored. Excessive quantities of such blocks MAY be treated as a form of resource abuse; see Section 6.

6. Security Considerations

Repositories constitute untrusted input as account holders have complete control over repository contents and repository hosts control binary encoding. Implementations must handle potential denial of service vectors from both malicious actors and accidental conditions such as corrupted data or implementation bugs.

6.1. CBOR Processing limits

Generic precautions must be followed when processing CBOR data, including enforcement of maximum serialized object size, maximum recursion depth for nested structures, and memory budget limits for deserialized data. While some CBOR implementations include these protections by default, implementations should verify and configure appropriate limits regardless of library defaults.

6.2. MST Structure Attacks

The efficiency of MST data structures depends on a uniform distribution of key hashes. Since account holders control record keys, they can perform key mining to generate sets of keys with specific layer assignments and sorting characteristics, resulting in inefficient tree structures. Such attacks can cause excessive storage overhead and network amplification during repository transmission.

To mitigate these attacks, implementations should:

* Limit the number of entries per MST node to a statistically reasonable maximum

* Impose limits on overall repository height

* Monitor and restrict other structural parameters that could be exploited through sophisticated key mining

6.3. Repository Import Validation

When importing repositories, implementations should verify the completeness and integrity of the repository structure. Serialized repositories may contain additional unrelated blocks beyond those required for the repository structure. Care should be taken during storage to avoid resource waste on unreferenced blocks and to prevent potential storage exhaustion attacks.

7. IANA Considerations

This document has no IANA actions.

8. References

8.1. Normative References

[CBOR] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", STD 94, RFC 8949, DOI 10.17487/RFC8949, December 2020, <https://www.rfc-editor.org/rfc/rfc8949>.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <https://www.rfc-editor.org/rfc/rfc3986>.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <https://www.rfc-editor.org/rfc/rfc4648>.

[RFC7049] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", RFC 7049, DOI 10.17487/RFC7049, October 2013, <https://www.rfc-editor.org/rfc/rfc7049>.

[SEC2] Standards for Efficient Cryptography Group, "SEC 2: Recommended Elliptic Curve Domain Parameters", January 2010, <https://www.secg.org/sec2-v2.pdf>.

[WEBASSEMBLY] Rossberg, A., "WebAssembly Core Specification", March 2026, <https://www.w3.org/TR/wasm-core-2>.

8.2. Informative References

[AT-ARCH] Newbold, B. and D. Holmgren, "Authenticated Transfer: Architecture Overview", June 2026.

[AT-SYNC] Holmgren, D. and B. Newbold, "Authenticated Transfer Protocol: Synchronization", June 2026.

[DASL-CAR] "DASL: Content Addressable aRchives (CAR)", n.d., <https://dasl.ing/car.html>.

[DRISL] "DRISL — Deterministic Representation for Interoperable Structures & Links", n.d., <https://dasl.ing/drisl.html>.

[MSTPAPER] Auvolat, A. and F. Taani, "Merkle Search Trees: Efficient State-Based CRDTs in Open Networks", October 2019, <https://inria.hal.science/hal-02303490/document>.

Appendix A. Data Model

All components of the repository data structure conform to a limited data model and defined encoding rules. CBOR encoding (following the rules in Appendix A.2) is used for consistent hashing of data. A JSON encoding is also defined for record data, with lossless mapping between the CBOR and JSON encodings.

The data model includes the following types:

* *null values*: represented as 'null' in JSON, and the null special value (major 7) in CBOR
* *boolean values*: represented as 'true' / 'false' in JSON, and special values (major 7) in CBOR
* *integer values*: with signed 64-bit precision. Represented as numbers in JSON, and Integers (majors 0,1) in CBOR
* *string values*: represented as strings in JSON, and UTF-8 Strings (major 3) in CBOR
* *byte string values*: represented with a special object type in JSON (see Appendix A.3) and as a Byte String (major 2) in CBOR
* *content hash links*: as described in Appendix A.1, represented as a special object type in JSON, and as a tag 42 byte string in CBOR
* *arrays*: represented as arrays in JSON, and Arrays (major 4) in CBOR
* *objects*: represented as objects in JSON, and Maps (major 5) in CBOR. Object keys must always be strings.

As a best practice to ensure compatibility with programming languages which represent all numbers in floating point by default, integer values should be limited to 53 bits of precision when possible.

A.1. Content Identifier (CID) Hashes

References to data objects by hash occur throughout the repository data structure. They also occur between records at the application layer. A consistent way of computing and encoding these hash links, named Content Identifier (CID), is described here. In addition to "CID Links" between objects, it is possible to represent CIDs as regular hash strings (without the "link" data model semantics). It is also possible to represent the hash of arbitrary binary data as a CID.

Data objects to be referenced are first encoded as CBOR. The encoded bytes are hashed using SHA-256, resulting in a 32-byte binary hash value. The hash bytes are prefixed with the 4-byte prefix value 0x01711220, resulting in a 36-byte binary CID.

This fixed prefix value is used for historical reasons, and indicates that the referenced data is CBOR encoded. If using a CID to reference arbitrary binary data, use the fixed 4-byte value 0x01551220 instead.

When representing a CID link in CBOR, the binary CID value has an additional null byte (0x00) prepended, then the 37 bytes are stored as a byte string using the IANA-registered CBOR Tag 42.

When representing a CID value as a string, the 36-byte binary CID value is encoded using [RFC4648] lower-case base32, and then the ASCII character ’b’ (lower-case B) is prefixed. This results in a 59 character lower-case ASCII string.

When referencing a CID link in JSON, first compute the string representation as described above. The link is then represented as a JSON object with a single key ($link) and the value being the string value. For example:

```
{
  "$link": "bafyreidfayvfuwqa7qlnopdjiqrxzs6blmoeu4rujcjtnci5beludirz2a"
}
```

A.2. CBOR Encoding

Repository content requires consistent binary representation across all implementations to ensure identical content hashes and verifiable integrity. All records, MST nodes, and commits must be encoded using Deterministically Encoded CBOR as specified in Section 4.2 of [CBOR], with map key ordering following the original specification in Section 3.9 of [RFC7049] for historical compatibility.

The encoding rules that apply in this document are:

* Integers are encoded in their shortest form

* All arrays, maps, and strings are encoded with explicit lengths; CBOR's indefinite-length encoding is not used

* Floating-point values are not used; this includes NaN and infinity values

* Map keys are sorted using the legacy length-first ordering of Section 3.9 of [RFC7049]

* Maps MUST NOT contain duplicate keys

The encoding rules described here are compatible with similar deterministic-CBOR profiles such as [DRISL].

A.3. JSON Encoding

The JSON representation of records or other repository data objects does not need to have a deterministic binary encoding.

Byte strings are represented in JSON using a special object type. The binary data is first string encoded in base64, as described in [RFC4648] Section 4. This variant is not URL-safe, and = padding is optional. The special JSON object has a single string key $bytes, and the value is the base64 encoded data. For example:

```
{
  "$bytes": "nFERjvLLiw9qm45JrqH9QTzyC2Lu1Xb4ne6+sBrCzI0"
}
```

Content hash links (CID links) are represented as special objects as described in Appendix A.1.

Appendix B. Cryptography

AT implementations must support both of the following elliptic curves and signature algorithms:

* NIST P-256 (also known as secp256r1 or p256) [SEC2]

* secp256k1 (also known as k256) [SEC2]

B.1. Signature Malleability

ECDSA signatures exhibit malleability, allowing transformation into distinct but equally valid signatures without access to the private key or original data. While the security impact is limited, signature malleability could enable broadcast of multiple valid versions of the same repository commit with different hashes, potentially causing consumer confusion.

To prevent such scenarios, AT requires all ECDSA signatures to be canonicalized in low-S form. Specifically, the s component of the signature must satisfy s n/2, where n is the order of the curve’s base point.

B.2. Signature Generation

To compute a signature over CBOR-encoded bytes in the context of AT:

1. Compute the SHA-256 hash of the encoded bytes. Do not encode the resulting hash bytes.

2. Sign the hash bytes using the current signing key associated with the account

3. Format the signature bytes as a concatenation of the 32-byte r and 32-byte s values

Appendix C. Timestamp Identifier (TID)

Timestamped Identifiers (TIDs) are compact string encodings of 64-bit integers, which can be used as logical clocks or locally-unique sorted identifiers. They are not expected to be globally unique.

They have the following structure:

* 64-bit integer with big-endian byte ordering

* Base32-sortable encoding using characters 234567abcdefghijklmnopqrstuvwxyz

* Fixed 13-character length with no padding (integer zero encodes as 2222222222222)

The layout of the 64-bit integer is:

* The top bit is always 0

* The next 53 bits represent microseconds since the UNIX epoch. 53 bits is chosen as the maximum safe integer precision in a 64-bit floating point number, as used by Javascript.

* The final 10 bits are an arbitrary "clock identifier."

When generating a sequence of TIDs in the same context (eg, for an individual account), care should be taken to ensure that the TID value always increments. If the system clock rolls backwards, or multiple TIDs are generated in the same microsecond, the microsecond component should be incremented past the previous generated value.

Appendix D. Namespaced Identifier (NSID) Syntax

Collections are identified by a Namespaced Identifier (NSID): an ASCII string in reverse domain-name order followed by an additional name segment. The portion preceding the final segment is the *domain authority*; the final segment is the *name*.

NSIDs MUST conform to the following syntax:

* Overall:
  - MUST contain only ASCII characters
  - MUST separate the domain authority and the name by an ASCII period (.)
  - MUST contain at least three segments
  - MUST be at most 317 characters in total length
* Domain authority:
  - Composed of segments separated by ASCII periods (.)
  - At most 253 characters in total (including periods), and at least two segments
  - Each segment MUST contain at least 1 and at most 63 characters
  - The allowed characters are ASCII letters (A-Z, a-z), digits (0-9), and hyphens (-)
  - Segments MUST NOT start or end with a hyphen
  - The first segment (the top-level domain) MUST NOT start with a digit

  - The domain authority is not case-sensitive and SHOULD be normalized to lowercase

* Name:

  - MUST contain at least 1 and at most 63 characters
  - The allowed characters are ASCII letters and digits only (A-Z, a-z, 0-9)
  - Hyphens are not allowed
  - MUST NOT start with a digit
  - Case-sensitive; implementations MUST NOT normalize case

Acknowledgments

TODO acknowledge.

Authors' Addresses

Daniel Holmgren
Bluesky Social
Email: daniel@blueskyweb.xyz

Bryan Newbold
Bluesky Social
Email: bryan@blueskyweb.xyz