JMAP
Internet-Draft
Intended status: Standards Track
Expires: 28 November 2026

M. De Gennaro
Stalwart Labs

27 May 2026

JMAP Object Metadata
draft-ietf-jmap-metadata-02

Abstract

This document defines an extension to the JSON Meta Application Protocol (JMAP) that lets clients and servers attach metadata to existing JMAP data types. Each opted-in data type gains two new properties, metadata and privateMetadata, whose values are objects keyed by _namespace identifier_. A namespace identifier is either a name registered with IANA or a domain name controlled by the vendor providing the namespace; the latter allows vendors and applications to extend the metadata schema without prior coordination. Because metadata is carried as a property on the related object, clients use the existing /get, /set, /changes, and /query methods to read, modify, and synchronize it.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 28 November 2026.

Copyright Notice

Table of Contents

1. Introduction

JMAP ([RFC8620], JSON Meta Application Protocol) is a generic protocol for synchronizing data, such as mail, calendars or contacts, between a client and a server. It is optimized for mobile and web environments, and aims to provide a consistent interface to different data types.

Metadata, or annotations, are auxiliary data elements that provide additional context, user-defined properties, or system-specific information about primary data objects. They enable user annotations, application-specific settings, collaborative tagging, and similar functionality. Other protocols have addressed this need with mechanisms such as the IMAP METADATA extension [RFC5464] and WebDAV dead properties [RFC4918]; this specification provides an analogous facility within JMAP.

This document defines a uniform mechanism for managing such metadata. Each opted-in JMAP data type gains a metadata property (shared metadata) and, optionally, a privateMetadata property (per-user metadata). The value of each property is an object keyed by _namespace identifier_. This document defines no initial namespaces; vendors use domain-name identifiers for proprietary extensions, and additional registered identifiers may be defined by future specifications.

Because metadata is carried as a property on the related object, the existing JMAP methods for that data type apply unchanged. Clients fetch metadata with the type's /get method, modify it through /set patches, learn of changes through /changes, and search across metadata and primary properties together through /query.

1.1. Notational Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

1.2. Addition to the Capabilities Object

The capabilities object is returned as part of the JMAP Session object; see Section 2 of [RFC8620]. This document defines one additional capability URI.

1.2.1. urn:ietf:params:jmap:metadata

This capability represents support for the metadata extension defined in this document. Servers that include this capability provide the metadata and (optionally) privateMetadata properties on the data types they list.

The value of this property in the JMAP Session "capabilities" property is an empty object.

The value of this property in an account's "accountCapabilities" property is an object containing the following field:

dataTypes: String[DataTypeMetadataInfo]
: An object whose keys are the names of JMAP data types for which the server supports metadata in this account. A type that does not appear in this object does not gain the metadata or privateMetadata properties in this account. The value associated with each type is a DataTypeMetadataInfo object as defined below.

A *DataTypeMetadataInfo* object has the following fields:

namespaces: String[]
: The set of IANA-registered metadata namespace names (as defined in Section 8.2) that the server supports on this data type. Each value MUST be a registered name (a sequence of US-ASCII letters, digits, hyphens, and underscores, with no dot). Vendor (domain-name) namespaces MUST NOT appear in this list; their support is signalled by supportsVendorNamespaces instead.

supportsVendorNamespaces: Boolean (default: false)
: Indicates whether the server accepts vendor (domain-name) namespaces on this data type. If true, any well-formed domain-name namespace (Section 2.1) may be written by clients, subject to the other constraints of this specification (access control, maxDepth, quota). If false, any /set operation that targets a domain-name namespace MUST be rejected with an "invalidProperties" SetError, and any subselector or filter condition referencing such a namespace MUST be treated as unsupported per the rules of Section 3.1 and Section 3.5.

A server SHOULD NOT advertise a data type that supports neither registered namespaces (empty namespaces) nor vendor namespaces (supportsVendorNamespaces: false), since such an entry advertises support for nothing.

supportsPrivate: Boolean (default: false)
Indicates whether this account, on this data type, supports per-user privateMetadata. This is a server feature flag, not a per-user permission. If false, the privateMetadata property MUST be absent from response objects of this type, all privateMetadata* filter conditions (Section 3.5) MUST be rejected with an "unsupportedFilter" error, and any /set operation that targets privateMetadata MUST be rejected with an "invalidProperties" SetError. Per-user write authorization on a privateMetadata write that the server otherwise supports is handled separately, through the "forbidden" SetError defined in Section 4.

maxDepth: UnsignedInt|null (default: null)
Maximum depth of nested objects within a namespace value on this data type. A depth of 1 indicates only flat properties are supported; a depth of 2 allows one level of nesting; and so forth. A value of null indicates no server-enforced limit; clients SHOULD treat null as unbounded, subject to the quota and per-value size limits the server may otherwise enforce. Depth counting is defined in Section 2.1.

A worked example of the capability object appears in Section 6.1.

2. The Metadata Properties

This extension introduces two properties on each opted-in data type:

metadata: String[Object] (default: {})
An object containing shared metadata associated with the object. Each key is a namespace identifier (see Section 2.1) and each value is structured according to the namespace’s definition. Shared metadata is visible to every user with read access to the related object, subject to Section 4. The server MUST NOT return null for this property and MUST always include it on opted-in data types, with an empty object {} when no metadata is set.

privateMetadata: String[Object] (default: {})
An object containing per-user metadata associated with the object, with the same structure as metadata. Private metadata is visible only to the user who set it; the server MUST filter responses so that each user sees only their own privateMetadata content, as described in Section 4. This property MUST be present in response objects (with value {} if the user has set no private metadata on

the object) when the data type's DataTypeMetadataInfo has supportsPrivate: true, and MUST be absent from response objects when supportsPrivate is false. The server MUST NOT return null for this property.

Both properties are mutable and behave like any other property of the related object. They appear in /get responses, accept patches through /set, contribute to the related type's state string (Section 2.2), and may be searched through the /query filter conditions defined in Section 3.5.

2.1. Namespaces

The keys of the metadata and privateMetadata objects are _namespace identifiers_. Each identifier MUST be one of:

* A _registered name_: a sequence of US-ASCII letters, digits, hyphens, and underscores, with no dot ("."). Registered names may be defined by future specifications using the procedure in Section 8.2; this document defines no registered names.

* A _domain name_ controlled by the vendor providing the namespace, in DNS form (containing at least one ".", e.g., example.com, acme.example.org). Vendors MAY use domain-name namespaces for proprietary metadata without registration; the dot in the name guarantees no collision with registered names.

The value associated with each namespace key is an object whose internal structure is defined by the specification or vendor owning that namespace. Property names within the namespace object follow normal JSON conventions; they do not need to be domain-prefixed, since cross-namespace isolation is already provided by the top-level key.

Example:

```
{
  "metadata": {
    "acme.example.com": {
      "color": "blue",
      "priority": "high",
      "project": {
        "id": "ALPHA-2024",
        "deadline": "2024-12-31"
      }
    }
  }
}
```

If the account capability specifies a maxDepth for a data type, the nesting depth of a namespace value MUST NOT exceed it. Depth is counted as the longest path of nested _objects_ from the namespace value to any descendant: a flat object whose values are scalars or arrays has depth 1, an object containing one level of nested objects has depth 2, and so forth. Arrays do not contribute to depth themselves, but objects appearing inside arrays do: for example, {"x": [{"y": 1}]} has depth 2 because the inner {"y": 1} is one level of nesting below the outer object. Scalar values (string, number, boolean, null) and empty arrays or objects have depth 1 at the position they occupy. Servers MUST reject patches that would produce a structure exceeding maxDepth with an "invalidProperties" SetError.

Servers MUST preserve namespace keys they do not recognize on update. A patch that targets one namespace MUST NOT remove or alter the content of any other namespace.

Vendors are encouraged to register namespace identifiers that are likely to be useful beyond the vendor's own products, using the procedure in Section 8.2. Registration enhances interoperability and avoids fragmentation.

2.1.1. Namespaces Removed Mid-Lifetime

If a server stops advertising a namespace that has previously had data written under it (for example, because of a server reconfiguration or a vendor withdrawing support), the existing stored data MUST continue to be returned by /get for read-only access. Any /set operation that targets the namespace, other than a patch that removes the namespace entirely ("metadata/<namespace>": null or "privateMetadata/<namespace>": null), MUST be rejected with an "invalidProperties" SetError. This ensures clients can always migrate data off a withdrawn namespace, but cannot continue to extend it.

2.2. State Behavior

Changes to metadata or privateMetadata are changes to the related object. They advance the related type's state string, appear in the related type's /changes response, and are otherwise indistinguishable from changes to any other property of that type, with the following clarification:

Because privateMetadata is filtered per viewer, a change made by user A to privateMetadata on a shared object MUST advance the related type's state string for user A and MUST NOT advance it for any other user. The same rule applies to /queryChanges: a change to user B's privateMetadata MUST NOT cause user A's Foo/queryChanges to consider

the object changed for filter purposes, since the change is not visible to A. A server that cannot implement per-viewer state filtering MUST NOT advertise the urn:ietf:params:jmap:metadata capability for any account in which supportsPrivate would be true for any data type, since cross-viewer state advancement leaks the existence of another user's private write (see Section 7).

3. Standard Method Extensions

For each data type listed in the dataTypes field of the account capability, the standard JMAP methods for that type are extended as described in the following subsections. These extensions are mandatory for the listed types: a server that advertises a data type in dataTypes MUST implement them.

3.1. /get

When a data type appears in dataTypes, omitting properties (or setting it to null) in Foo/get MUST return both metadata and privateMetadata (the latter where supportsPrivate is true) alongside the data type's other properties. The two properties may also be requested explicitly by name in the properties array, like any other JMAP property.

When a client wants only a subset of the metadata, it MAY request individual namespaces using slash-separated subselector paths in the properties argument. The following rules apply to subselectors introduced by this specification:

* Each subselector path MUST have exactly two segments, of the shape metadata/<namespace> or privateMetadata/<namespace>. Paths with zero, one, or three-or-more segments under these roots MUST be rejected with an "invalidArguments" method-level error.

* Multiple subselectors with the same root combine by union. For example, ["metadata/x", "metadata/y"] returns {"metadata": {"x": ..., "y": ...}} (and nothing else under metadata).

* An explicit root entry supersedes subselectors with the same root. For example, ["metadata", "metadata/x"] returns the complete metadata object; the metadata/x entry is redundant rather than restrictive.

* A subselector whose namespace is not supported on the data type MUST be silently omitted from the response. A namespace is supported if it is a registered name listed in namespaces, or if it is a domain name and supportsVendorNamespaces is true. This lets clients perform capability-tolerant fetches without first consulting the session capability.

* This specification defines slash-path subselectors only for metadata and privateMetadata. The use of slash-path subselectors with any other property name is outside the scope of this document.

3.2. /set

The metadata and privateMetadata properties are mutable.

In a create entry of Foo/set, each property's value MUST be a complete JSON object (possibly the empty object {}). A value of null for either property MUST be rejected with an "invalidProperties" SetError; the canonical "no metadata" form is {}.

In an update entry of Foo/set, clients MAY either send a complete object as the value of metadata or privateMetadata (replacing the previous value entirely), or use the PatchObject form (Section 5.3 of [RFC8620]) with the keys treated as JSON Pointer paths [RFC6901] relative to the object root. Patch paths follow the structure metadata/<namespace> (replacing or removing an entire namespace value), metadata/<namespace>/<key> (modifying a single property within a namespace), privateMetadata/<namespace>, and privateMetadata/<namespace>/<key>. As required by [RFC6901], any / character within a key MUST be escaped as ~1, and any ~ character MUST be escaped as ~0.

Setting a patch value to null removes the targeted key, per standard PatchObject semantics (Section 5.3 of [RFC8620]). Setting a path to a complete object value replaces whatever was previously at that path; in particular, a patch of the shape "metadata/<namespace>": { ... } replaces the entire content of that namespace, deleting any keys not present in the new value. Clients that wish to add or update individual properties without removing others MUST patch the specific keys.

The server MUST validate every write against the account capability for the data type. When more than one of the following rules applies, the server MUST evaluate them in the order listed; the first matching rule determines the SetError:

1. If the user lacks the required access on the related object (write permission to modify shared metadata, or read permission to modify their own privateMetadata), reject with "forbidden".

2. If the path begins with privateMetadata and supportsPrivate is false for this data type, reject with "invalidProperties".

3. If the targeted namespace is not supported on this data type, reject with "invalidProperties" (except for the migration case in Section 2.1.1). A namespace is supported if it is a registered name listed in namespaces, or if it is a domain name and supportsVendorNamespaces is true.

4. If a namespace value would exceed maxDepth, reject with "invalidProperties".

When a patch modifies one namespace, all other namespaces under metadata and privateMetadata MUST be preserved unchanged. This is a natural consequence of JSON Pointer patch semantics; it is mentioned here only to emphasize that no special "preserve unknown properties" handling beyond ordinary patch semantics is required.

3.2.1. Quota Enforcement on /set

Servers SHOULD enforce quota limits on the total storage consumed by metadata within an account, as described in Section 5. If a patch would cause the account to exceed its metadata quota, the server MUST reject the operation with an "overQuota" SetError.

3.3. /changes

For each data type listed in dataTypes, the Foo/changes request accepts the following additional optional argument:

ignoreMetadataOnlyChanges: Boolean (default: false)
: If true, the server MUST exclude from the updated array any id whose only changes since sinceState, from the viewer's perspective, are confined to metadata and/or privateMetadata. Ids with changes to additional properties (whether those additional changes coexist with metadata changes or not) remain present. The state string still advances normally, so a client using this argument continues to see correct synchronization for all non-metadata-only changes; metadata-only changes are simply not reported to this client. When this argument is true, updatedProperties (defined below) MUST be null in the response, since no metadata-only ids remain that would make it useful.

The response is extended to include the following additional argument:

updatedProperties: String[]|null This argument is determined _per viewer_: each user's Foo/changes response is computed against the changes visible to that user (in particular, privateMetadata written by other users is invisible). If, from the viewer's perspective, the only properties that have changed on every id in the updated array since the old state are metadata and/or privateMetadata, this argument MUST be set to a list containing only those property names (e.g., ["metadata"], ["privateMetadata"], or ["metadata", "privateMetadata"]). If any other property of any id in the array may also have changed (from the viewer's perspective), or if the server cannot determine the answer for any id in the array, this argument MUST be null. The argument applies uniformly to all ids in updated; per-id granularity is not provided.

This argument follows the same convention used by Mailbox/changes in Section 2.5 of [RFC8621].

3.4. /queryChanges

Foo/queryChanges is unchanged in its method signature, but its per-viewer behavior on shared objects with per-user privateMetadata MUST follow the rule given in Section 2.2: a change to another user's privateMetadata MUST NOT cause this user's Foo/queryChanges to consider the object changed for filter purposes.

3.5. /query

For each data type listed in dataTypes, the FilterCondition object for Foo/query is extended with the optional fields defined below. Several of those fields take a MetadataTextMatch value.

A *MetadataTextMatch* object has the following fields:

path: String
The path within the metadata object to match against, of the form <namespace> or <namespace>/<key> (with / and ~ escaped per [RFC6901] where applicable). Same syntax as the value of metadataExists.

value: String
The string to search for at path. The interpretation of this string (substring containment versus exact equality) is determined by the FilterCondition field that carries the MetadataTextMatch.

The extended FilterCondition fields are:

metadataExists: String
A path of the form <namespace> or <namespace>/<key> (with / and ~ escaped per [RFC6901] where applicable). Matches an object if and only if a value is present at the given path within the object's metadata property. A namespace-only path matches if the namespace key is present and its value is not the empty object {}.

privateMetadataExists: String
As metadataExists, but matches against the authenticated user's privateMetadata.

metadataTextContains: MetadataTextMatch
Matches an object if the metadata value at the supplied path is a string that contains the supplied value as a case-insensitive substring. The condition does not match if the path does not exist within metadata, or if the value at the path is not a string.

privateMetadataTextContains: MetadataTextMatch
As metadataTextContains, but searches the authenticated user's privateMetadata.

metadataTextEquals: MetadataTextMatch
Matches an object if the metadata value at the supplied path is a string that is exactly equal (case-sensitive, byte-for-byte) to the supplied value. The condition does not match if the path does not exist within metadata, or if the value at the path is not a string.

privateMetadataTextEquals: MetadataTextMatch
As metadataTextEquals, but searches the authenticated user's privateMetadata.

The following availability rules apply to all six conditions:

* Servers MUST support metadataExists and privateMetadataExists (where supportsPrivate is true), since they require only a presence check.

* Servers MAY reject any of the four text-match conditions with an "unsupportedFilter" error if implementation cost or storage layout makes them infeasible. Clients SHOULD be prepared to fall back to a client-side scan in this case.

* Any privateMetadata* condition (existence or text) MUST be rejected with an "unsupportedFilter" error if supportsPrivate is false for the data type.

* A condition that names a namespace not supported on the data type MUST match no objects (it does not produce an error; absent data simply matches no presence or text check). A namespace is supported if it is a registered name listed in namespaces, or if it is a domain name and supportsVendorNamespaces is true.

These conditions compose with the data type's existing filter conditions through the usual AND/OR/NOT operators (Section 5.5 of [RFC8620]). For example, a client may query for Email objects in a particular mailbox whose vendor metadata contains a search term in a single server-side query, as shown in Section 6.7.

4. Access Control

Access control for metadata follows the access control of the related object. The two properties are governed by different rules:

* The metadata property is shared. A user who has read access to the related object MAY read the object's metadata. A user who has write access to the related object MAY modify the object's metadata. For shareable JMAP data types as defined in Section 4 of [RFC9670], "read" and "write" mean the mayRead and mayWrite permissions of the related object (or the equivalent indicators defined by the type-specific specification).

* The privateMetadata property is per-user. The server MUST filter responses so that each user sees only the privateMetadata content they themselves wrote. Private metadata written by another user on the same related object MUST NOT be visible. To write privateMetadata on a related object, the user needs only read access to the related object; write access on the related object is not required.

The server MUST enforce these rules consistently across /get, /set, /query, /queryChanges, and /changes. If a user attempts to read metadata on an object they cannot read, the server MUST return the same error it returns for any other property of an inaccessible object. If a user attempts to write metadata without write permission on the related object, or to write privateMetadata without read permission, the server MUST reject the operation with a "forbidden" SetError.

If supportsPrivate is false for the data type, the server MUST reject every write that targets privateMetadata on that type with an "invalidProperties" SetError, regardless of the user's permissions on the related object.

When sharing permissions on a related object change, the visibility of its metadata changes accordingly. privateMetadata belonging to a user remains visible to that user as long as the user retains read access to the related object; if read access is revoked, the user's privateMetadata SHOULD be hidden along with the rest of the object.

4.1. Account Delegation and Administrative Access

Servers that support account delegation, impersonation, or administrative access have to decide whether a delegated session sees another user's privateMetadata on objects shared with both. Such exposure is rarely intended. Unless the deployment has explicitly authorized per-user metadata access for the delegated session, servers SHOULD NOT expose the account owner's privateMetadata to delegated sessions, and SHOULD NOT permit delegated sessions to write privateMetadata that would be attributed to the account owner. When a delegated session does write privateMetadata, the server MUST attribute the write to the authenticated identity of the session (the delegate), not to the account owner.

5. Quota

Servers SHOULD enforce quota limits on the storage consumed by metadata within an account. Unbounded metadata growth could be exploited to exhaust resources, particularly because privateMetadata is per-user and so a single shared object can carry one private payload per user with access to it.

Servers that support the JMAP Quotas extension [RFC9425] SHOULD integrate metadata storage into the quota framework defined there. Servers that do not implement [RFC9425] SHOULD still enforce implementation-defined limits and reject overruns with an "overQuota" SetError.

The size metric used for metadata quota is implementation-defined; clients SHOULD NOT assume that two servers compute identical sizes for the same data. Servers that wish to expose a predictable metric to clients SHOULD document it.

6. Examples

6.1. Capability

A session for an account whose server supports metadata on three data types, with varying private-metadata and depth settings:

```
{
  "capabilities": {
    "urn:ietf:params:jmap:metadata": {}
  },
  "accounts": {
    "A1": {
      "accountCapabilities": {
        "urn:ietf:params:jmap:metadata": {
          "dataTypes": {
            "Email": {
              "namespaces": [],
              "supportsVendorNamespaces": true,
              "supportsPrivate": true,
              "maxDepth": 4
            },
            "Mailbox": {
              "namespaces": [],
              "supportsVendorNamespaces": true,
              "supportsPrivate": true,
              "maxDepth": null
            },
            "CalendarEvent": {
              "namespaces": [],
              "supportsVendorNamespaces": true,
              "supportsPrivate": false,
              "maxDepth": 2
            },
            "FileNode": {
              "namespaces": ["photography"],
              "supportsVendorNamespaces": false,
              "supportsPrivate": false,
              "maxDepth": 3
            }
          }
        }
      }
    }
  }
}
```

6.2. Fetching a Mailbox with its metadata

Request the shared metadata and the user’s private metadata for a single Mailbox:

```
[
  ["Mailbox/get", {
    "accountId": "A1",
    "ids": ["MB1"],
    "properties": [
      "id", "name",
      "metadata/acme.example.com",
      "privateMetadata/acme.example.com"
    ]
  }, "c1"]
]
```

Response:

```
[
  ["Mailbox/get", {
    "accountId": "A1",
    "state": "mb-100",
    "list": [
      {
        "id": "MB1",
        "name": "Team Inbox",
        "metadata": {
          "acme.example.com": {
            "color": "blue",
            "owner": "team-alpha"
          }
        },
        "privateMetadata": {
          "acme.example.com": {
            "workflowState": "pending-review"
          }
        }
      }
    ],
    "notFound": []
  }, "c1"]
]
```

6.3. Patching a single property within a namespace

Update one property inside a namespace without disturbing the others:

```
[
  ["Mailbox/set", {
    "accountId": "A1",
    "update": {
      "MB1": {
        "metadata/acme.example.com/color": "green"
      }
    }
  }, "c1"]
]
```

Removing a single property uses a patch to null:

```
[
  ["Mailbox/set", {
    "accountId": "A1",
    "update": {
      "MB1": {
        "metadata/acme.example.com/color": null
      }
    }
  }, "c1"]
]
```

6.4. Creating an Email with private metadata

Create a draft Email and attach a private workflow annotation atomically. The metadata is just another property on the create:

```
[
  ["Email/set", {
    "accountId": "A1",
    "create": {
      "draft1": {
        "mailboxIds": { "MB1": true },
        "subject": "Project Update",
        "from": [{ "email": "alice@example.com" }],
        "to":   [{ "email": "bob@example.com" }],
        "bodyStructure": {
          "type": "text/plain",
          "partId": "1"
        },
        "bodyValues": {
          "1": { "value": "Here is the project update..." }
        },
        "privateMetadata": {
          "acme.example.com": {
            "workflowState": "pending-review",
            "assignedTo": "carol@example.com"
          }
        }
      }
    }
  }, "c1"]
]
```

The response is a single Email/set response; there is no separate metadata response, because metadata is part of the Email object.

6.5. Attaching photography metadata to a FileNode

This example uses a _fictional_ IANA-registered namespace photography, assumed for the purpose of this example to be specified for photographic information about image files and applicable to the FileNode data type. The capability for the account in Section 6.1 advertises FileNode as supporting photography and no vendor namespaces.

Request:

```
[
  ["FileNode/set", {
    "accountId": "A1",
    "update": {
      "F456": {
        "metadata/photography": {
          "geoLocation": {
            "latitude": 46.362,
            "longitude": 14.090
          },
          "cameraMake": "Canon",
          "cameraModel": "EOS R5",
          "aperture": "f/2.8",
          "shutterSpeed": "1/250",
          "iso": 400,
          "focalLength": "50mm",
          "dateTaken": "2023-10-01T01:14:00Z",
          "imageSize": {
            "width": 6000,
            "height": 4000
          }
        }
      }
    }
  }, "c1"]
]
```

Subsequent retrieval of the FileNode, requesting only the relevant metadata namespace:

```
[
  ["FileNode/get", {
    "accountId": "A1",
    "ids": ["F456"],
    "properties": [
      "id", "name", "type", "size",
      "metadata/photography"
    ]
  }, "c2"]
]
```

Response:

```
[
  ["FileNode/get", {
    "accountId": "A1",
    "state": "f-200",
    "list": [
      {
        "id": "F456",
        "name": "lake-island.jpg",
        "type": "image/jpeg",
        "size": 2458624,
        "metadata": {
          "photography": {
            "geoLocation": {
              "latitude": 46.362,
              "longitude": 14.090
            },
            "cameraMake": "Canon",
            "cameraModel": "EOS R5",
            "aperture": "f/2.8",
            "shutterSpeed": "1/250",
            "iso": 400,
            "focalLength": "50mm",
            "dateTaken": "2023-10-01T01:14:00Z",
            "imageSize": {
              "width": 6000,
              "height": 4000
            }
          }
        }
      }
    ],
    "notFound": []
  }, "c2"]
]
```

6.6. Updating a CalendarEvent and its metadata atomically

This example updates a primary property of a CalendarEvent and two vendor metadata properties in the same /set call. All changes apply together; if any fails for the same id, none take effect, since they are part of a single update entry.

Request:

```
[
  ["CalendarEvent/set", {
    "accountId": "A1",
    "update": {
      "CE789": {
        "title": "Quarterly Review Meeting",
        "start": "2024-12-15T14:00:00",
        "metadata/acme.example.com/lastModifiedReason":
            "Rescheduled per manager request",
        "metadata/acme.example.com/approvalStatus": "pending"
      }
    }
  }, "c1"]
]
```

A successful response is a single CalendarEvent/set response; metadata changes do not produce a separate response because they are properties of the event itself:

```
[
  ["CalendarEvent/set", {
    "accountId": "A1",
    "oldState": "c-200",
    "newState": "c-201",
    "updated": {
      "CE789": null
    }
  }, "c1"]
]
```

6.7. Querying by metadata and a primary property together

Find all Email objects in a particular Mailbox whose private vendor memo contains the text "follow up":

```
[
  ["Email/query", {
    "accountId": "A1",
    "filter": {
      "operator": "AND",
      "conditions": [
        { "inMailbox": "MB-inbox" },
        { "privateMetadataExists": "acme.example.com/memo" },
        {
          "privateMetadataTextContains": {
            "path":  "acme.example.com/memo",
            "value": "follow up"
          }
        }
      ]
    }
  }, "c1"]
]
```

6.8. Detecting a metadata-only change through /changes

After a server-side metadata update, the next Email/changes call returns:

```
[
  ["Email/changes", {
    "accountId": "A1",
    "oldState": "e-100",
    "newState": "e-103",
    "hasMoreChanges": false,
    "created": [],
    "updated": ["EM1", "EM2"],
    "destroyed": [],
    "updatedProperties": ["metadata"]
  }, "c1"]
]
```

The client knows it only needs to refetch the metadata property of EM1 and EM2, not the full Email objects.

7. Security considerations

7.1. Metadata Confidentiality

Metadata may contain sensitive information: personal notes, workflow states, application-specific data, and other user-generated content. Servers MUST enforce the access-control rules in Section 4. Failures here are direct privacy breaches.

privateMetadata is per-user. The server MUST ensure that each user sees only their own content, even on shared related objects. This is the most common implementation pitfall in this specification: a query or /get implementation that returns privateMetadata keyed by user other than the requester is a serious bug. Servers MUST verify that result filtering applies uniformly to /get, /query, /changes, /queryChanges, and to any cached or denormalized representation.

Per-viewer state (Section 2.2) interacts with confidentiality. A server that advances the related type's state for user B in response to user A's private write is not leaking the metadata content, but it is leaking a signal that _something_ private to A changed on the related object. The specification therefore requires per-viewer state filtering and forbids advertising the capability for accounts that cannot provide it.

7.2. User Identity and Authentication

The private-metadata model relies on accurate user identification. Servers MUST:

* Reliably identify the authenticated user for every metadata operation.

* Maintain accurate association between privateMetadata content and the user that wrote it.

* Prevent authentication bypass or identity spoofing that could allow access to other users' private metadata.

* Carefully handle account delegation and administrative access: see Section 4.1.

7.3. Injection Attacks Through Namespace Values

Namespace values accept arbitrary client-supplied content. Servers and clients MUST treat namespace values as untrusted user input. Typical risks include script injection (values rendered in a web context without sanitization), JSON injection (values that break downstream parsing), path traversal (cleverly named keys), and SQL injection (values stored in SQL columns without parameterization).

Servers MUST:

* Validate that namespace identifiers conform to the registered-name or domain-name syntax.

* Sanitize values before use in any context where interpretation could occur.

* Apply appropriate output encoding when displaying metadata.

* Enforce limits on the size of individual values and on maxDepth.

* Reject or sanitize values containing control characters or other potentially harmful content.

Clients that render metadata to users MUST treat the content as untrusted, with appropriate sandboxing, Content Security Policy, and context-appropriate escaping.

7.4. Resource Exhaustion

Without controls, metadata can be abused for denial of service:

* Storage exhaustion: bulk creation of large metadata payloads, particularly under privateMetadata, where every user with access to a shared object can independently consume space.

* Processing exhaustion: deeply nested or pathologically large namespace values.

* Query complexity: text-match filter conditions (metadataTextContains/metadataTextEquals and their private variants) that require expensive scanning, particularly when combined with per-user filtering across many users. Servers MAY reject such conditions with "unsupportedFilter" (Section 3.5) to bound the cost.

Servers SHOULD apply the quota mechanism in Section 5, enforce maxDepth, time out long-running queries, and apply rate limiting to metadata writes. Servers SHOULD monitor for unusual usage patterns (e.g., a single user creating private metadata on unusually many objects).

7.5. Server Vulnerabilities

Implementations MUST be robust against malformed input. In particular, servers MUST:

* Validate all input against the type signatures in this document.

* Handle JSON parsing errors gracefully without exposing internal state.

* Protect against integer overflow, buffer overflow, and similar low-level vulnerabilities when processing metadata.

* Implement efficient per-user filtering of privateMetadata so that it does not become a denial-of-service vector under load.

* Log security-relevant events (auth failures, permission denials, attempts to read another user's private metadata) for monitoring.

7.6. Client Vulnerabilities

Clients that handle metadata MUST:

* Validate server responses; in particular, do not assume the structure of a namespace without checking.

* Distinguish clearly in the UI between metadata (visible to others) and privateMetadata (visible only to the user), so users do not inadvertently disclose private notes.

* Refuse to execute or interpret metadata content as code without explicit user consent and appropriate sandboxing.

* Apply Content Security Policy and similar browser-level mitigations when displaying metadata in web UIs.

8. IANA considerations

8.1. JMAP Capability Registration for "metadata"

IANA will register the "metadata" JMAP Capability as follows:

Capability Name: urn:ietf:params:jmap:metadata
Specification document: this document
Intended use: common
Change Controller: IETF
Security and privacy considerations: this document, Section 7

8.2. Creation of the "JMAP Metadata Namespaces" Registry

IANA has created the "JMAP Metadata Namespaces" registry to record metadata namespace identifiers used as top-level keys in the metadata and privateMetadata properties defined by this specification.

This registry uses the Expert Review process ([RFC8126], Section 4.5). Registrants must propose a name without a dot ("."), since names containing a dot are reserved for vendor domain-name namespaces and need no registration. The designated expert MUST

ensure that the proposed name does not collide with an existing registration and that the specification provides sufficient detail for interoperability (in particular, the structure of the namespace's value).

Registrations may be of intended use "common", "reserved", or "obsolete". A "reserved" registration reserves a name without assigning semantics; an "obsolete" registration marks a name that should no longer be used by up-to-date implementations.

8.2.1. Preliminary Community Review

Notice of a potential new registration SHOULD be sent to the JMAP mailing list jmap@ietf.org (mailto:jmap@ietf.org) for review. The intent is to solicit comments on the namespace identifier, the clarity of the specification, and any interoperability or security considerations. The submitter MAY revise or withdraw the proposal at any time.

8.2.2. Change Procedures

The change controller for a registration MAY request changes to its definition using the same procedure as the original registration. Significant changes that would invalidate existing data SHOULD be made only to correct serious errors. Registrations MUST NOT be deleted; namespaces that are no longer appropriate for use can be marked obsolete by a change to their intended-use field.

The owner of a registration MAY transfer responsibility to another person or agency by informing IANA.

8.2.3. "JMAP Metadata Namespaces" Registry Template

Namespace Identifier:
: The name registered for use as a key in the metadata or privateMetadata object. MUST consist of US-ASCII letters, digits, hyphens, and underscores, and MUST NOT contain a dot.

Value Type:
: A description of the structure of values stored under this namespace key (for example, "object whose values are strings").

Applicable Properties:
: One of: "metadata" (this namespace may appear only under metadata), "privateMetadata" (only under privateMetadata), or "both" (under either).

Applicable Data Types:

A list of JMAP data type names on which this namespace is meaningful, or "any" if the namespace is not restricted.

Reference or Description:
A brief description, or an RFC number and section reference, describing the namespace's contents. May be omitted for "reserved" entries.

Intended Usage:
"common", "reserved", or "obsolete".

Change Controller:
Who may request changes to this entry (IETF for RFCs from the IETF stream).

8.2.4. Submit Request to IANA

Registration requests can be sent to iana@iana.org (mailto:iana@iana.org).

8.2.5. Designated Expert Review

The designated expert (DE) is primarily concerned with preventing name collisions and ensuring that the specification provides enough detail for interoperability. For common-use registrations, the DE is expected to confirm that suitable documentation (per Section 4.6 of [RFC8126]) is available. A published specification is not required for reserved or obsolete registrations.

The DE will either approve or deny the registration and publish a notice of the decision to the JMAP working group mailing list (or its successor) and to IANA. A denial MUST be justified and SHOULD include concrete suggestions for how the request can be modified to become acceptable.

8.2.6. Initial Contents

This document defines no initial entries for the "JMAP Metadata Namespaces" registry. Future specifications may register entries through the procedure defined in this section.

9. References

9.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC6901] Bryan, P., Ed., Zyp, K., and M. Nottingham, Ed., "JavaScript Object Notation (JSON) Pointer", RFC 6901, DOI 10.17487/RFC6901, April 2013, <https://www.rfc-editor.org/rfc/rfc6901>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/rfc/rfc8126>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

[RFC8620] Jenkins, N. and C. Newman, "The JSON Meta Application Protocol (JMAP)", RFC 8620, DOI 10.17487/RFC8620, July 2019, <https://www.rfc-editor.org/rfc/rfc8620>.

[RFC9670] Jenkins, N., Ed., "JSON Meta Application Protocol (JMAP) Sharing", RFC 9670, DOI 10.17487/RFC9670, November 2024, <https://www.rfc-editor.org/rfc/rfc9670>.

9.2. Informative References

[RFC4918] Dusseault, L., Ed., "HTTP Extensions for Web Distributed Authoring and Versioning (WebDAV)", RFC 4918, DOI 10.17487/RFC4918, June 2007, <https://www.rfc-editor.org/rfc/rfc4918>.

[RFC5464] Daboo, C., "The IMAP METADATA Extension", RFC 5464, DOI 10.17487/RFC5464, February 2009, <https://www.rfc-editor.org/rfc/rfc5464>.

[RFC8621] Jenkins, N. and C. Newman, "The JSON Meta Application Protocol (JMAP) for Mail", RFC 8621, DOI 10.17487/RFC8621, August 2019, <https://www.rfc-editor.org/rfc/rfc8621>.

[RFC9425] Cordier, R., Ed., "JSON Meta Application Protocol (JMAP) for Quotas", RFC 9425, DOI 10.17487/RFC9425, June 2023, <https://www.rfc-editor.org/rfc/rfc9425>.

Appendix A. Changes

[[This section to be removed by RFC Editor]]

draft-ietf-jmap-metadata-02

* Redesigned the extension around two properties (metadata and privateMetadata) on each opted-in JMAP data type, rather than a separate Metadata data type with its own methods. Removed the Metadata data type, all Metadata/* methods, the related object types, the extended /get//set arguments, and the uniqueness and cascading-deletion machinery.

* Added a per-data-type capability advertising IANA-registered namespace support (namespaces), vendor (domain-name) namespace support (supportsVendorNamespaces), private-metadata support (supportsPrivate), and a nesting limit (maxDepth).

* Added updatedProperties to Foo/changes for opted-in types, modelled on Mailbox/changes in RFC 8621. Specified per-viewer behavior for /changes and /queryChanges on shared objects carrying per-user private metadata.

* Added FilterCondition fields to Foo/query: metadataExists, privateMetadataExists, metadataTextContains, metadataTextEquals, and the corresponding private variants. Servers MAY reject the text-match conditions with "unsupportedFilter" when implementation cost is prohibitive.

* Added an ignoreMetadataOnlyChanges argument to Foo/changes, letting metadata-uninterested clients skip wakeups for metadata-only changes on data types they otherwise care about.

* Specified PatchObject semantics for metadata and privateMetadata, including handling of unknown namespaces, error ordering on /set, and the migration case where a namespace is removed from the server’s advertised set after data has been written.

* Replaced the previous metadata-types and metadata-properties registries with a single "JMAP Metadata Namespaces" registry. This document defines no initial entries; bindings to other protocols (e.g., IMAP METADATA, WebDAV dead properties) are expected to be defined in separate specifications.

draft-ietf-jmap-metadata-01

* Renamed parentType and parentId properties to relatedType and relatedId throughout the document to better reflect their purpose and avoid implying a strict hierarchy.

* Added filterRelatedType and filterMetadataType optional parameters to Metadata/changes method to allow filtering changes by related object type and metadata type.

* Changed the error type from "invalidProperties" to "alreadyExists" when a client attempts to create a Metadata object that violates the uniqueness constraint.

* Added requirement that relatedType must be specified in the same FilterCondition object whenever relatedIds is used in Metadata/query.

* Added id as a MUST support property for sorting in Metadata/query; other properties changed to SHOULD support.

* Replaced the metadataTypes argument with a fetchMetadata Boolean argument (default: false) in the extended /get method.

* Added onSuccessCreateMetadata and onSuccessUpdateMetadata arguments to the extended /set method to enable transactional metadata operations that are conditional on the success of the related object operation.

draft-ietf-jmap-metadata-00

* Initial version

Author's Address

Mauro De Gennaro
Stalwart Labs LLC
1309 Coffeen Avenue, Suite 1200
Sheridan, WY 82801
United States of America
Email: mauro@stalw.art
URI: https://stalw.art