Network Working Group
Internet-Draft
Updates: 8620 (if approved)
Intended status: Standards Track
Expires: 9 October 2026

B. Gondwana
Fastmail

7 April 2026

JMAP File Storage extension
draft-ietf-jmap-filenode-13

Abstract

The JMAP base protocol (RFC8620) provides the ability to upload and download arbitrary binary data. This binary data is called a "blob", and can be used in all other JMAP extensions.

This extension adds a method to expose blobs as a filesystem along with the types of metadata that are provided by other remote filesystem protocols.

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 9 October 2026.

Copyright Notice

Table of Contents

1. Introduction

JMAP ([JMAP-CORE] — JSON Meta Application Protocol) is a generic protocol for synchronizing data 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.

In the same way that JMAP Calendars ([JMAP-CALENDARS]) replaces CalDAV ([CALDAV]) and JMAP Contacts ([JMAP-CONTACTS]) replaces CardDAV ([CARDDAV]), this document replaces the use of WebDAV ([WEBDAV]) for remote filesystem access.

1.1. Conventions Used in This Document

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.

2. Addition to the Capabilities Object

The capabilities object is returned as part of the JMAP Session object; see [JMAP-CORE], Section 2.

This document defines an additional capability URI.

2.1. urn:ietf:params:jmap:filenode

The capability urn:ietf:params:jmap:filenode being present in the "accountCapabilities" property of an account represents support for the FileNode datatype. Servers that include the capability in one or more "accountCapabilities" properties MUST also include the property in the "capabilities" property.

The value of this property in the JMAP session "capabilities" property MUST be an empty object.

The value of this property in an account's "accountCapabilities" property is an object that MUST contain the following information on server capabilities and permissions for that account:

* maxFileNodeDepth: "UnsignedInt|null"

  The maximum depth of the FileNode hierarchy (i.e., one more than the maximum number of ancestors a FileNode may have), or null for no limit.

* maxSizeFileNodeName: "UnsignedInt"

  The maximum length, in (UTF-8) octets, allowed for the name of a FileNode. This MUST be at least 100, although it is recommended servers allow more.

* forbiddenNameChars: "String|null"

  A string where each character is forbidden in FileNode names. If null, the server does not restrict name characters beyond the requirement that names be non-empty Net-Unicode strings. The server MUST reject names containing any of these characters with an "invalidProperties" error.

* forbiddenNodeNames: "String[]|null"

  An array of complete names that the server will not accept as FileNode names, compared case-insensitively. For example, [".", "..", "CON", "PRN", "AUX", "NUL"] would forbid common path-traversal names and Windows reserved device names. If null, the server does not forbid any specific names.

* fileNodeQuerySortOptions: "String[]"

  A list of all the values the server supports for the "property" field of the Comparator object in a "FileNode/query" sort (see "FileNode/query" below). This MAY include properties the client does not recognise (for example, custom properties specified in a vendor extension). Clients MUST ignore any unknown properties in the list.

* mayCreateTopLevelFileNode: "Boolean"

If true, the user may create a FileNode (see "FileNode/set" below) in this account with a null parentId. (Permission for creating a child of an existing FileNode is given by the "myRights" property on that FileNode.)

* webTrashUrl: "String|null"

  The URL at which the folder with the "trash" role can be viewed on the web. If null, there is no web URL available.

* caseInsensitiveNames: "Boolean"

  If true, the server treats file names as case-insensitive for all name collision checks, including the sibling uniqueness constraint and onExists handling. The server preserves the original case as provided by the client.

* webUrlTemplate: "String|null"

  A template by which any node can be viewed on the web, in URI Template (level 1) format [URI-TEMPLATE]. The available variable is {id}, which is the FileNode id. If null, there is no web URL available.

* webWriteUrlTemplate: "String|null"

  A URI Template (level 1) [URI-TEMPLATE] for writing content to a file node. The available variable is {id}, which is the FileNode id. If null, direct HTTP writes are not supported and clients must use FileNode/set to update content. See "Direct HTTP Write" below for the HTTP semantics.

2.1.1. Capability Example

```
{
  "urn:ietf:params:jmap:filenode": {
    "maxFileNodeDepth": 50,
    "maxSizeFileNodeName": 255,
    "fileNodeQuerySortOptions": [
      "name", "type", "size", "created", "modified",
      "nodeType", "tree"
    ],
    "forbiddenNameChars": "/<>:\"\\|?*",
    "forbiddenNodeNames": [".", "..", "CON", "PRN", "AUX",
      "NUL", "COM0", "COM1", "COM2", "COM3", "COM4", "COM5",
      "COM6", "COM7", "COM8", "COM9", "LPT0", "LPT1", "LPT2",
      "LPT3", "LPT4", "LPT5", "LPT6", "LPT7", "LPT8", "LPT9"
    ],
    "caseInsensitiveNames": false,
    "mayCreateTopLevelFileNode": false,
    "webTrashUrl": "https://files.example.com/trash",
    "webUrlTemplate": "https://files.example.com/view/{id}",
    "webWriteUrlTemplate": "https://files.example.com/write/{id}"
  }
}
```

3. FileNode Data Type

A FileNode is a set of metadata which behaves similarly to an inode in a filesystem. In [WEBDAV] terminology a FileNode can refer to either a collection or a resource.

The following JMAP Methods are selected by the urn:ietf:params:jmap:filenode capability.

3.1. FileNode objects

The filenode object has the following keys:

* id: "Id" (immutable; server-set)

  The Id of the FileNode.

* parentId: "Id|null"

  The Id of the parent node, or null if this is a top level node.

* nodeType: "String" (immutable; default: server-inferred)

  The type of node. Values are registered in the "JMAP FileNode Types" registry (see IANA Considerations). This document defines the following values: "file", "directory", and "symlink". This is

immutable after creation. If not provided on creation, the server infers the type: "file" if blobId is non-null, "symlink" if target is non-null, "directory" otherwise. Attempting to change nodeType after creation is an "invalidProperties" error.

* blobId: "Id|null"

  The blobId for the content of this node. MUST be non-null for file nodes (including zero byte files). MUST be null for directory and symlink nodes. Creating or updating a file node to have a null blobId is an "invalidProperties" error. Updating the blobId replaces the node's content; the server MUST update size to match the new blob. A blob referenced by a FileNode MUST NOT be expired or garbage collected by the server while the FileNode exists.

* target: "String[]|null"

  The target path for symlink nodes, as an array of path elements. MUST be non-null for symlinks. MUST be null for file and directory nodes. Each element is a node name in the path. An empty string as the first element indicates an absolute path from the root of the tree; otherwise the path is relative to the symlink's parent. An element of ".." refers to the parent directory. The target is not required to resolve to an existing node (symlinks may dangle). The target is mutable — updating it changes where the symlink points.

* size: "UnsignedInt|null" (server-set)

  The size in bytes of the associated blob data. This MUST be null for directory and symlink nodes, and non-null for file nodes. Size is optional on create and update, but if provided it MUST match the size of the provided blobId. If the client updates blobId without providing size, the server sets size from the new blob and returns the size in the "updated" response.

* name: "String"

  User-visible name for the FileNode. This MUST be a Net-Unicode string [UNICODE] of at least 1 character in length, subject to the maximum size given in the capability object. There MUST NOT be two sibling FileNodes with both the same parent and the same name. The server MUST reject names containing any character listed in forbiddenNameChars, or matching any entry in forbiddenNodeNames (case-insensitively), with an "invalidProperties" error. See Implementation Considerations for guidance on which characters and names to forbid.

* type: "String|null"

  The media type of the FileNode. This MUST be null for directory and symlink nodes, and non-null for file nodes. Valid values are found in the IANA media-types registry. Servers MUST NOT reject media types that are not recognised. Servers MUST reject media types if the value does not conform to the ABNF of [MEDIATYPE] Section 4.2.

* created: "UTCDate" (default: current server time)

  The date the node was created.

* modified: "UTCDate|null" (default: current server time)

  The date the node was last updated. The server does not automatically update this value. If the client does not include modified in an update, the server MUST leave it unchanged. If the client sets modified to null, the server MUST set it to the current server time. See Implementation Considerations for usage guidance.

* accessed: "UTCDate|null" (default: current server time)

  The date the node was last accessed. As with modified, the server does not automatically update this value; clients SHOULD provide an updated value when appropriate. If not included in an update, the server MUST leave it unchanged. If set to null, the server MUST set it to the current server time.

* changed: "UTCDate" (server-set)

  The date the server last recorded a change to any property of this node. The server MUST automatically update this value whenever any property of the node is modified, including blobId, name, parentId, shareWith, executable, and all other mutable properties. This is not settable by clients.

* executable: "Boolean" (default: false)

  If true, the node should be treated as an executable by operating systems that support this flag.

* isSubscribed: "Boolean" (default: true)

  This property is stored per user, and if true, the node should be displayed to the current user. Some servers may not allow this field to be changed for some or all nodes, e.g. only for

directories, or only for nodes on which the shareWith ACL is actually set for this user, not nodes which inherit their ACL from a parent.

* myRights: "FilesRights" (server-set)

  The set of rights (ACLs) the user has in relation to this node. A *FilesRights* object has the following properties:

  - mayRead: Boolean The user may read the properties and blob content of this node.

  - mayAddChildren: Boolean The user may create child nodes in this directory.

  - mayRename: Boolean The user may rename or move this node.

  - mayDelete: Boolean The user may destroy this node.

  - mayModifyContent: Boolean The user may update the content-related properties of this node (blobId, type, target, modified, accessed, executable).

  - mayShare: Boolean The user may change the sharing of this node (see [JMAP-SHARING]).

* shareWith: "Id[FilesRights]|null"

  A map of userId to rights for users this node is shared with. The owner of the node MUST NOT be in this set. This is null if the user requesting the object does not have myRights.mayShare, or if the node is not shared with anyone.

* role: "String|null"

  An indication that this directory has a special role. The role MUST be null for files. A node with the "root" role should have a null parentId. Clients MUST ignore unrecognised role values. Values are registered in the "JMAP FileNode Roles" registry (see IANA Considerations).

3.2. FileNode Methods

3.2.1. FileNode/get

This is a standard Foo/get method, with the following additional request argument:

* fetchParents: "Boolean" (default: false)

  If true, the server returns all ancestor nodes of the requested ids in addition to the requested nodes themselves. This allows a client to reconstruct the full path to each requested node in a single call. Ancestor nodes that have already been included (either because they were explicitly requested or because they are ancestors of multiple requested nodes) are not duplicated. The ancestors are returned as additional entries in the list array.

3.2.2. FileNode/changes

This is a standard Foo/changes method.

3.2.3. FileNode/set

This is a standard Foo/set method, with the following differences:

Since parentId creates a tree structure, an attempt to move a node to a parent for which this node is also an ancestor is an error, and an invalidProperties error will be returned.

A server MUST order creation and deletion operations within a single FileNode/set such that the sibling name uniqueness constraint is retained at the end of the transaction, but replacing an existing file can be done atomically.

There are these additional top level arguments:

* onDestroyRemoveChildren: "Boolean" (default: false)

  If false, an attempt to destroy a FileNode which is the parentId of another FileNode will be rejected with a nodeHasChildren error. NOTE: if all the child nodes are being destroyed in the same operation, then the server MUST NOT return this error. Servers MUST either sort the destroys children before parents, or only check this constraint on the final state, remembering that JMAP set operations must be atomic.

  If true, then all child nodes will also be destroyed when a node is destroyed. When deleting child nodes, the server MUST include the ids of all deleted nodes in the method response.

* onExists: "String|null" (default: null)

  If null, an attempt to create or update a FileNode which would cause a name collision will be rejected by the server with an "alreadyExists" error.

If "replace", the existing item will be destroyed. In this case, the server MUST include the id of the replaced item in the destroyed response list. NOTE: if the replaced item is a directory which has children, then the server MUST respond with a nodeHasChildren error to this action unless onDestroyRemoveChildren is true.

If "rename", the server will change the "name" field to not clash, using an algorithm of its choice. If the server changes the name, it MUST include the new "name" value in the created or updated response field for this id.

* compareCaseInsensitively: "Boolean" (default: false)

  If true, name collision checks (including onExists handling) treat names as case-insensitive for this request. This has no effect if the server already has caseInsensitiveNames set to true in its capability. This allows clients syncing to case-insensitive platforms to prevent name collisions that would be problematic on those platforms.

Errors (in addition to standard SetError codes from [JMAP-CORE]):

* "alreadyExists" — a create or update would cause a name collision with an existing sibling FileNode, and onExists is null. The SetError object MUST include an existingId property with the id of the existing FileNode.

* "invalidProperties" — an attempt was made to move a node to a descendant of itself, creating a cycle in the tree.

* "nodeHasChildren" — a destroy was attempted on a FileNode that has children, and onDestroyRemoveChildren is false.

3.2.4. FileNode/copy

This is a standard Foo/copy function with the same additional top-level arguments as FileNode/set, onDestroyRemoveChildren and onExists, with the same behaviour.

Errors: the same additional errors as FileNode/set apply.

3.2.5. FileNode/query

This is a standard Foo/query method except for the following:

There’s one more property to the query:

* depth: "UnsignedInt|null"

  The number of levels of subdirectories to recurse into. If absent, null, or zero, do not recurse.

The following filter criteria are defined:

* isTopLevel: "Boolean"

  If true, the node must have a null parentId to match the condition.

* parentId: "Id"

  A FileNode id. A node must have a parentId equal to this to match the condition.

* ancestorId: "Id"

  A FileNode id. A node must have an ancestor (parent, parent of parent, etc.) with an id equal to this to match the condition.

* descendantId: "Id"

  A FileNode id. A node must be an ancestor (parent, parent of parent, etc.) of the node with this id to match the condition. This is the inverse of ancestorId.

* nodeType: "String"

  A node type value. Only nodes with precisely this nodeType match this condition.

* role: "String"

  A role name. Only nodes with precisely this role match this condition.

* hasAnyRole: "Boolean"

  If true, any node with a defined role matches this condition. If false, any node which has a role does not match this condition.

* blobId: "Id"

  A FileNode must have a blobId equal to this to match the condition.

* isExecutable: "Boolean"

  If true, the FileNode must have a true executable value.

* createdBefore: "UTCDate"

  The creation date of the node must be before this date to match the condition.

* createdAfter: "UTCDate"

  The creation date of the node must be on or after this date to match the condition.

* modifiedBefore: "UTCDate"

  The modified date of the node must be before this date to match the condition.

* modifiedAfter: "UTCDate"

  The modified date of the node must be on or after this date to match the condition.

* accessedBefore: "UTCDate"

  The accessed date of the node must be before this date to match the condition.

* accessedAfter: "UTCDate"

  The accessed date of the node must be on or after this date to match the condition.

* minSize: "UnsignedInt"

  The size of the node in bytes must be equal to or greater than this number to match the condition.

* maxSize: "UnsignedInt"

  The size of the node in bytes must be less than this number to match the condition.

* name: "String"

  A FileNode must have exactly the same octets in its name property to match the condition.

* nameMatch: "String"

  Matches the _name_ property of the node using glob syntax. The following special patterns are defined: * matches any sequence of characters, ? matches any single character, and a pair of brackets matches any single character in the bracketed set (e.g., [abc] matches "a", "b", or "c"; [a-z] matches any lowercase letter; [!abc] or [^abc] matches any character not in the set). All other characters match literally. The match is case-insensitive.

* type: "String"

  A FileNode must have exactly the same octets in its type property to match the condition.

* typeMatch: "String"

  Matches the _type_ property of the node using the same glob syntax as _nameMatch_.

* body: "String"

  Match the content of the referenced blob, see the definition of _body_ in section 4.4.1 of [JMAP-MAIL]. The server may use any technology to extract meaningful text from the blob for searching, or interpret the string in any way, to choose the nodes that it believes the user wants to see.

* text: "String"

  Is equivalent to _body_ OR _nameMatch_ OR _typeMatch_.

It also supports the following additional sort properties:

* tree:

  Sort by tree; which means by name, but any directory/collection node is immediately followed by the recursive application of the same sort to its child nodes. This is similar to the output of the find command on a filesystem with the depth parameter provided above.

* nodeType:

  Sort by node type. Directories sort first, then symlinks, then files.

* type:

Sorts directories first, and sorts by media type for files

3.2.6. FileNode/queryChanges

This is a standard Foo/queryChanges method.

4. Direct HTTP Write

When the server advertises a webWriteUrlTemplate, clients can update a file node's content directly via HTTP without using FileNode/set. The URL is constructed by expanding the template with the FileNode id. This is only valid for file nodes (those with a non-null blobId); requests targeting directory nodes MUST be rejected with an HTTP 400 status.

The client MUST have mayModifyContent permission on the node. Requests without sufficient permission MUST be rejected with an HTTP 403 status. Authentication uses the same mechanism as other JMAP HTTP endpoints.

4.1. PUT

A PUT request replaces the entire content of the node. The request body is the new content. The Content-Type header sets the media type of the node.

On success, the server MUST update the node's blobId, size, and type properties, and return an HTTP 200 response with a JSON body:

```
{
  "blobId": "Bnewblob123",
  "size": 48576,
  "type": "text/plain"
}
```

4.2. PATCH

A PATCH request applies a partial modification to the node's current content. The Content-Type of the request indicates the patch format. The supported patch formats are those listed in the supportedPatchTypes capability of [JMAP-BLOBEXT] (e.g., application/x-rdiff-delta, application/x-bsdiff, text/x-diff). The server applies the delta to the node's current blob to produce the new content.

On success, the server MUST update the node's blobId and size properties. The type is not changed unless the client includes an X-FileNode-Type header, in which case the server MUST update the node's type to the header value. The response is an HTTP 200 with the same JSON body as PUT:

```
{
  "blobId": "Bpatched456",
  "size": 49152,
  "type": "text/plain"
}
```

Servers that do not support a given patch format MUST respond with HTTP 415 (Unsupported Media Type).

5. Access Control

Since nodes create a tree, ACLs created by shareWith automatically apply to children as well, so if mayRead is set on a node, all its child nodes are also readable. The myRights property on every node reflects the derived rights for the current user, taking inheritance into account.

When a shareWith change on a node causes the derived myRights to change for any descendant, the server MUST report those descendants as changed in FileNode/changes responses. The server SHOULD NOT report descendants whose derived myRights did not actually change, but MAY do so if it cannot efficiently determine whether the derived values differ.

If a server does not support changing access on non-directory nodes, it can set mayShare to false on those nodes, even if the parent directory has true.

Nodes which are not "discoverable" MUST return notFound errors if fetched with FileNode/get and MUST NOT be returned in response to FileNode/query. Nodes are discoverable in one of two ways:

1. If the node or an ancestor of the node has mayRead true.

2. If the node is an ancestor of a node which has mayRead true.

In the second case, the node itself will have mayRead false, and appears only to give the full path to the visible nodes. This means that a query with that node as the parent will only return those of its children which are otherwise discoverable through the second discoverability rule.

This leads to a sharee seeing the full path to anything that they have access to, but nothing else.

E.g. in a unix homedirectory environment where user Alice had shared the "forBob" folder, one might see, when logged in as Bob:

```
/home
  /alice
    /shared
      /forBob
        file1.jpg
        file2.txt
        ...
  /bob
    bobfile.txt
    ...
```

While Alice would see many more files and folders at the home directory level.

This does lead to potentially large changes in the visible Node set, so when Alice first shared that directory, Bob would have been told of a large number of new Nodes in response to a FileNode/changes query. The server might report these as "created" or "updated" — it is not always possible for the server to know whether a node was previously visible to the user.

6. Quotas

Servers that support JMAP Quotas ([JMAP-QUOTAS]) SHOULD provide Quota objects with "FileNode" in the types array. Both the octets resource type (total storage used by file node blobs) and the count resource type (number of FileNode objects) are applicable.

7. Integration with JMAP Blob Extensions

When a server advertises both urn:ietf:params:jmap:filenode and urn:ietf:params:jmap:blobext, the ArchiveEntry object type (defined in [JMAP-BLOBEXT]) is extended with two additional properties:

* nodeId: "Id|null" A FileNode id. When present, the server populates the archive entry from the referenced FileNode's properties. The blobId and name are taken from the FileNode. The FileNode's created and modified timestamps map to the archive entry's time metadata. The FileNode's myRights (specifically the mayModifyContent right) and the executable property inform the mode field in the archive entry (e.g., permissions and executable bit). If the ArchiveEntry also provides explicit values for any

of these properties, the explicit values take precedence. Mutually exclusive with providing blobId directly when the FileNode is a file.

* recurse: "Boolean" (default: false) Only valid when the referenced FileNode is a directory. When true, the server recursively includes all children of the directory in the archive. Each child entry's name is its full path relative to the directory. Directory names MUST include a trailing /.

If nodeId references a FileNode that the user does not have permission to read, or that does not exist, the server MUST reject the Blob/convert creation with a notFound SetError.

7.1. Examples

To archive an entire directory tree rooted at FileNode "dirabc" into a zip file:

```
json [["Blob/convert", { "accountId": "account1", "create": {
"myarchive": { "archive": { "type": "application/zip", "entries": [{
"nodeId": "dirabc", "name": "/", "recurse": true }] } } } }, "R1"]]
```

To archive specific files from different locations along with a full subdirectory:

```
json [["Blob/convert", { "accountId": "account1", "create": {
"selective": { "archive": { "type": "application/zip", "entries": [ {
"nodeId": "filenode1", "name": "README.txt" }, { "nodeId":
"dirnode2", "name": "docs/", "recurse": true } ] } } } }, "R1"]]
```

7.2. Extracting Archives into FileNodes

When both capabilities are present, the ExtractRecipe (defined in [JMAP-BLOBEXT]) is extended with the following additional properties:

* parentNodeId: "Id|null" A FileNode id of a directory. When present, the server extracts the archive contents into this directory, creating FileNode objects for each entry. Subdirectories in the archive are created as directory FileNodes. File entries become file FileNodes with their blobId set to the extracted content. The created and modified timestamps are taken from the archive entry metadata. If the archive entry includes a mode field, the server SHOULD use it to set the executable property on the created FileNode.

* onExists: "String|null" (default: null) Controls the behaviour when an extracted entry would collide with an existing FileNode name under the same parent. The values and semantics are the same as for FileNode/set (see above): null rejects with an alreadyExists SetError, "replace" destroys the existing node, and "rename" gives the new node a server-chosen name.

When parentNodeId is set, each ArchiveEntry in the response entries array includes an additional nodeId property containing the id of the FileNode that was created for that entry.

If parentNodeId references a FileNode that does not exist, is not a directory, or for which the user does not have mayAddChildren permission, the server MUST reject the Blob/convert creation with an appropriate SetError (notFound, invalidProperties, or forbidden).

7.2.1. Example

Extracting a zip file into a directory:

```
json [["Blob/convert", { "accountId": "account1", "create": { "e1": {
"extract": { "blobId": "Barchive123", "type": "application/zip",
"parentNodeId": "dir456" } } } }, "R1"]]
```

The response includes a nodeId for each created entry:

```
json [["Blob/convert", { "accountId": "account1", "created": { "e1":
{ "id": "Barchive123", "type": "application/zip", "size": 104857,
"entries": [ { "name": "docs/", "entryType": "directory", "nodeId":
"node001" }, { "name": "docs/readme.txt", "blobId": "Bdd001",
"entryType": "file", "modified": "2026-03-01T12:00:00Z", "mode":
"0644", "nodeId": "node002" }, { "name": "docs/logo.png", "blobId":
"Bdd002", "entryType": "file", "modified": "2026-02-15T09:30:00Z",
"mode": "0644", "nodeId": "node003" } ] } }, "notCreated": {} },
"R1"]]
```

8. Implementation Considerations

8.1. Modification Timestamps

Clients SHOULD provide an updated modified value when modifying a node. This gives clients control over the timestamp, for example when preserving original modification times during file synchronization. Clients that do not care about preserving timestamps can set modified to null to have the server use the current time.

Clients may also choose to update the modified timestamp on a parent directory when moving a child node in or out, to reflect that the directory's contents have changed.

8.2. Timestamp Precision

The UTCDate type (RFC 3339) supports fractional seconds. Servers SHOULD include sub-second precision in created, modified, accessed, and changed timestamps where their storage backend supports it. Clients MUST preserve fractional seconds when round-tripping timestamps back to the server.

8.3. Filename Character Restrictions

Servers should set forbiddenNameChars and forbiddenNodeNames to include characters and names that are problematic for their storage backend and for the broadest set of clients.

The following characters are problematic on common platforms:

* Windows: <, >, :, ", \, |, ?, *, and control characters (U+0000-U+001F). Windows also forbids trailing dots and spaces in names.

* macOS: : (HFS+ legacy; displayed as / in Finder).

* Linux: the null byte (U+0000).

The following names are problematic:

* All platforms: . and .. (path traversal).

* Windows: reserved device names CON, PRN, AUX, NUL, COM0-COM9, and LPT0-LPT9. These are reserved regardless of file extension (e.g., "CON.txt" is also forbidden).

A server targeting broad client compatibility should include at least /<>:"\|?* in forbiddenNameChars and at least ., .., and the Windows reserved names in forbiddenNodeNames.

8.4. Case Sensitivity

Windows and macOS filesystems are typically case-insensitive (treating "Foo" and "foo" as the same name), while Linux filesystems are typically case-sensitive. Clients syncing to case-insensitive platforms should use compareCaseInsensitively on FileNode/set requests to prevent creating sibling nodes whose names differ only in case, as these cannot coexist on the target platform.

8.5. Access Time Updates

Updating accessed on every file read generates significant API traffic for little value. Clients are encouraged to use a "relatime" strategy similar to modern POSIX filesystems: only update accessed if the current value is older than modified, or if more than 24 hours have elapsed since the last accessed update. Clients should batch access time updates with their next JMAP request rather than making a dedicated call.

9. Security considerations

All security considerations from [JMAP-CORE] apply to this document.

9.1. Path Traversal

Clients that reconstruct filesystem paths from FileNode hierarchies MUST validate the resulting paths and reject any that would escape the intended directory tree. Servers SHOULD include / in forbiddenNameChars and . and .. in forbiddenNodeNames to prevent path traversal attacks via node names.

9.2. Symlink Targets

Symlink targets are not validated by the server to point to existing nodes. Clients that resolve symlinks MUST guard against symlink loops (a symlink pointing to an ancestor of itself) and MUST NOT follow symlinks outside the FileNode tree. Servers SHOULD impose a limit on the length of target values and MAY reject targets containing ".." path components.

9.3. Denial of Service

Deep nesting of FileNodes or very large numbers of children under a single parent can consume significant server resources. Servers SHOULD enforce reasonable limits via maxFileNodeDepth and MAY impose additional limits on the number of children per node. Recursive operations such as destroying a subtree with onDestroyRemoveChildren can be expensive; servers SHOULD impose limits on the size of subtrees that can be destroyed in a single operation.

9.4. Access Control

The discoverability rules defined in this document mean that ancestor nodes of shared content are visible (with mayRead false) to users who have access to descendants. While these ancestor nodes do not expose file content, their names and structure may reveal information about the file hierarchy. Server administrators and users sharing content should be aware of this.

Changes to the shareWith property on a node affect all descendants. Removing sharing from a parent node will make all descendants undiscoverable to the affected users, which may be surprising if those users had been actively working with the shared files.

9.5. Content Security

Servers that perform content scanning or malware detection SHOULD scan blobs referenced by FileNode objects. The type property is client-asserted and MUST NOT be trusted for security decisions; servers SHOULD independently verify content types where this matters.

10. IANA considerations

10.1. JMAP Capability registration for "filenode"

IANA is requested to register the "filenode" JMAP Capability as follows:

Capability Name: urn:ietf:params:jmap:filenode

Specification document: this document

Intended use: common

Change Controller: IETF

Security and privacy considerations: this document, Security Considerations

10.2. JMAP Error Codes registration for "nodeHasChildren"

IANA is requested to register the "nodeHasChildren" JMAP Error Code as follows:

JMAP Error Code: nodeHasChildren

Intended use: common

Change Controller: IETF

Description: The node being destroyed is still referenced by other nodes which have not been destroyed.

Reference: this document

10.3. JMAP Data Types registration for "FileNode"

IANA is requested to register the "FileNode" JMAP Data Type as follows:

Type Name: FileNode

Can Reference Blobs: Yes

Can Use For State Change: Yes

Capability: urn:ietf:params:jmap:filenode

Reference: this document

10.4. JMAP FileNode Types Registry

IANA is requested to create a new "JMAP FileNode Types" registry with the following initial values. New registrations are subject to Specification Required ([IANA-GUIDELINES]).

Type	Description	Reference
file	A regular file with blob content	this document
directory	A collection that may contain child nodes	this document
symlink	A symbolic link to another path in the tree	this document

Table 1

10.5. JMAP FileNode Roles Registry

IANA is requested to create a new "JMAP FileNode Roles" registry with the following initial values. New registrations are subject to Expert Review ([IANA-GUIDELINES]).

Role	Description	Reference
root	The base of a filesystem	this document
home	A user’s home directory	this document
temp	Temporary space; may be cleaned up automatically	this document
trash	Deleted data; may be removed automatically or manually	this document
documents	Document storage	this document
downloads	Downloaded files	this document
music	Audio files	this document
pictures	Photos and images	this document
videos	Video files	this document

Table 2

11. TODO

* create real-world clients to test this

12. Changes

EDITOR: please remove this section before publication.

The source of this document exists on github at:
https://github.com/brong/draft-gondwana-jmap-filenode/

draft-ietf-jmap-filenode-13

* Reordered methods to get/changes/set/copy/query/queryChanges per RFC 8620 convention.

* Promoted Direct HTTP Write to top-level section.

* Moved Access Control before Quotas and integrations.

* Added caseInsensitiveNames capability and compareCaseInsensitively parameter on FileNode/set.

draft-ietf-jmap-filenode-12

* Added symlink support: nodeType property (file, directory, symlink) with IANA registry, and target property (array of path elements).

* Replaced isFile/isDirectory query filters and sort with nodeType.

* Expanded myRights: replaced mayWrite with mayAddChildren, mayRename, mayDelete, mayModifyContent.

* Added forbiddenNameChars and forbiddenNodeNames capabilities; removed hardcoded name restrictions from spec.

* Added changed (server-set) timestamp.

* Added Implementation Considerations section with guidance on timestamps, access times, and filename restrictions.

draft-ietf-jmap-filenode-11

* Updated for blobext recipe renames (ExtractRecipe).

draft-ietf-jmap-filenode-10

* Added ExtractRecipe parentNodeId extension for extracting archives directly into FileNode trees.

draft-ietf-jmap-filenode-09

* Renamed hasType filter to isFile and isDirectory (separate filters).

* Renamed hasRole filter to role for consistency.

* Defined glob matching syntax for nameMatch/typeMatch.

* Added fetchParents argument to FileNode/get.

* Added descendantId filter to FileNode/query.

draft-ietf-jmap-filenode-08

* Removed immutability restriction on blobId, size, and type — nodes can now have their content updated via FileNode/set.

* Added webWriteUrlTemplate capability for direct HTTP PUT/PATCH writes to file nodes.

* Added documents, downloads, music, pictures, and videos roles.

* Added Quotas section referencing RFC 9425.

draft-ietf-jmap-filenode-07

* added webTrashUrl and webUrlTemplate to the capabilities object

* added capability example JSON

* added explicit Errors sections to FileNode/set and FileNode/copy

* fixed typos and spelling errors

* added BCP 14 boilerplate

* fleshed out Security Considerations

* replaced role TODO with IANA registry

* documented webUrlTemplate variables

* clarified modified/accessed semantics: client-managed with null to reset to server time

* clarified blobId constraints and blob lifetime

* documented myRights inheritance and FileNode/changes behaviour when shareWith changes

* wrapped long lines throughout

* added Integration with JMAP Blob Extensions section (ArchiveEntry extension with nodeId and recurse)

draft-ietf-jmap-filenode-06

* Documented FileNode/copy and detailed that it has the same new top-level keys

draft-ietf-jmap-filenode-05

* Renamed onDuplicate to onExists for name alignment

* added "role" for directories

* added a "TODO" for putting more restrictions on node names

* changed hasParentId to isTopLevel with reversed boolean meaning

draft-ietf-jmap-filenode-04

* Documented that blobId, size, and type are immutable after creation.

* Documented that creating a file and deleting the old copy of the file within a transaction is legal.

* Added onDuplicate top-level option for FileNode/set, giving both "rename" and "replace" options.

* Updated the definition of shareWith to say that the keys of the hash are Ids, not arbitrary strings

draft-ietf-jmap-filenode-03

* Added ’text’ and ’body’ searches (added JMAP-MAIL reference as additional information for body search)

* Updated JMAP-CONTACTS and JMAP-SHARING references to published RFC numbers rather than draft names

* Noted that the server MUST included the nodeids of deleted child nodes.

* Added isSubscribed

* Renamed mayAdmin to mayShare to align with other specs

* Described the inheritance of ACLs

draft-ietf-jmap-filenode-02

* Convert to Kramdown-RFC format (no intentional changes)

draft-ietf-jmap-filenode-01

* Refreshing draft only

draft-ietf-jmap-filenode-00

* upload as a working group document

draft-gondwana-jmap-filenode-01

* require a blobId for the zero-byte file

* make size also null for collections

* add more to the TODO section

* bikeshed; FileNode

* correct UTCDate, UnsignedInt, and normalised UTF-8.

* add some fields to the capabilities object

draft-gondwana-jmap-filenode-00

* initial proposal

13. Acknowledgements

Neil Jenkins and the JMAP working group at the IETF.

{backmatter}

14. References

14.1. Normative References

[IANA-GUIDELINES]
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>.

[JMAP-BLOBEXT]
Gondwana, B., "JMAP Blob Extensions", Work in Progress, Internet-Draft, draft-gondwana-jmap-blobext-05, 30 March 2026, <https://datatracker.ietf.org/doc/html/draft-gondwana-jmap-blobext-05>.

[JMAP-CORE]
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>.

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

[MEDIATYPE]
Freed, N., Klensin, J., and T. Hansen, "Media Type Specifications and Registration Procedures", BCP 13, RFC 6838, DOI 10.17487/RFC6838, January 2013, <https://www.rfc-editor.org/rfc/rfc6838>.

[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>.

[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>.

[URI-TEMPLATE]
Gregorio, J., Fielding, R., Hadley, M., Nottingham, M., and D. Orchard, "URI Template", RFC 6570, DOI 10.17487/RFC6570, March 2012, <https://www.rfc-editor.org/rfc/rfc6570>.

14.2. Informative References

[CALDAV] Daboo, C., Desruisseaux, B., and L. Dusseault, "Calendaring Extensions to WebDAV (CalDAV)", RFC 4791, DOI 10.17487/RFC4791, March 2007, <https://www.rfc-editor.org/rfc/rfc4791>.

[CARDDAV] Daboo, C., "CardDAV: vCard Extensions to Web Distributed Authoring and Versioning (WebDAV)", RFC 6352, DOI 10.17487/RFC6352, August 2011, <https://www.rfc-editor.org/rfc/rfc6352>.

[JMAP-CALENDARS]
Jenkins, N. and M. Douglass, "JSON Meta Application Protocol (JMAP) for Calendars", Work in Progress, Internet-Draft, draft-ietf-jmap-calendars-26, 4 November 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-jmap-calendars-26>.

[JMAP-CONTACTS]
Jenkins, N., Ed., "JSON Meta Application Protocol (JMAP) for Contacts", RFC 9610, DOI 10.17487/RFC9610, December 2024, <https://www.rfc-editor.org/rfc/rfc9610>.

[JMAP-MAIL]
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>.

[JMAP-QUOTAS]
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>.

[UNICODE] Klensin, J. and M. Padlipsky, "Unicode Format for Network Interchange", RFC 5198, DOI 10.17487/RFC5198, March 2008, <https://www.rfc-editor.org/rfc/rfc5198>.

[WEBDAV] 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>.

Author’s Address

Bron Gondwana
Fastmail
Email: brong@fastmailteam.com