Workgroup: JMAP
Internet-Draft: draft-ietf-jmap-blob-06
Updates: 8620 (if approved)
Published: 12 November 2021
Intended Status: Standards Track
Expires: 16 May 2022
Authors: B. Gondwana, Ed.
Fastmail

JMAP Blob management extension

Abstract

The JMAP base protocol (RFC8620) provides the ability to upload and download arbitrary binary data via HTTP POST and GET on defined endpoint. This binary data is called a "Blob".

This extension adds additional ways to create and access Blobs, by making inline method calls within a standard JMAP request.

This extension also adds a reverse lookup mechanism to discover where blobs are referenced within other data types.

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 16 May 2022.

Copyright Notice

Copyright (c) 2021 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with

respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction

Sometimes JMAP ([RFC8620]) interactions require creating a Blob and then referencing it. In the same way that IMAP Literals ([RFC7888]) were extended to reduce roundtrips for simple data, embedding simple small blobs into the JMAP method stream can reduce roundtrips.

Likewise, when fetching an object, it can be useful to also fetch the raw content of that object without a separate roundtrip.

Since raw blobs could contain arbitrary binary data, this document allows the use of the base64 coding specified in [RFC4648].

Where JMAP is being proxied through a system which applies additional access restrictions, it can be useful to know where a blob is referenced in order to decide whether to allow it to be downloaded, so this document defines a way to look up where a particular blobId is referenced.

2. 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.

3. Addition to the Capabilities Object

The capabilities object is returned as part of the JMAP Session object; see [RFC8620], Section 2.

This document defines an additional capability URI.

3.1. urn:ietf:params:jmap:blob

This represents support for additional API methods on the Blob datatype.

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 that MAY contain the following information on server capabilities and permissions for that account:

*maxSizeBlobSet: UnsignedInt

if set, gives the maximum size of a blob in octets that the server will allow you to create (size of the final output of catenate or of encoded forms). This SHOULD be the same as the RFC8620 value maxSizeUpload.

*maxCatenateItems: UnsignedInt

if set, gives the maximum number of of CatenateSourceObjects allowed per creation in a Blob/set. Servers SHOULD allow at least 64 items.

*supportedTypeNames: [String]|null

an array of data type names that are supported for Blob/lookup. May be null or not present if the account does not support reverse lookups.

Example:

```
{
  "capabilities": {
    ...,
    "urn:ietf:params:jmap:blob": {}
  },
  "accounts": {
    "A13842": {
      ...
      "accountCapabilities": {
        "urn:ietf:params:jmap:blob": {
          "maxSizeBlobSet": 50000000,
          "maxCatenateItems": 100,
          "supportedTypeNames" : [
            "Mailbox",
            "Thread",
            "Email"
          ]
        }
      }
    }
  }
}
```

4. Blob Methods

A blob is a sequence of zero or more octets.

The JMAP base spec [RFC8620] defines the Blob/copy method, which is unchanged by this specfication, and is selected by the urn:ietf:params:jmap:core capability.

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

4.1. Blob/set

This is a standard JMAP set method.

4.1.1. create

Properties:

Exactly one of:

*data:asText: String

data which can be represented as utf-8 encoded text

*data:asBase64: String

binary data encoded as a ([RFC4648] Section 4) Base 64 string

*catenate: [CatenateSourceObject]

list of one or more octet sources in order

Also optionally:

*type: String|null (default: null)

hint for media type of the data

Result is:

*id: Id

the blobId which was created

*type: String|null

the media type as given in the creation (if any); or detected from content; or null

*size: UnsignedInt

as per RFC8620 - the size of the created blob in octets

Plus any other properties identical to those that would be returned in the JSON response of the RFC8620 upload endpoint (which may be extended in the future - this document anticipates that implementations will extend both the upload endpoint and the Blob/set responses in the same way)

CatenateSourceObject:

Exactly one of:

*data:asText: String|null

*data:asBase64: String|null

or a blobId source:

*blobId: Id

*offset: UnsignedInt|null (may be zero)

*length: UnsignedInt|null (must not be zero)

If null then offset is assumed to be zero.

If null then length is the remaining octets in the blob.

If the range can not be fully satisfied (i.e. extends past the end of the data in the blob) then the catenate itself is invalid and results in a notCreated response for this creation id.

If the data properties or catenate properties have any invalid references or invalid data contained in them, the server MUST NOT guess as to the user's intent, and MUST reject the creation and return a notCreated response for that creation id.

Likewise, invalid characters in the base64 of data:asBase64, or invalid UTF-8 in data:asText MUST result in a nonCreated response.

It is legal to create a blob by calling catenate with a single CatenateSourceObject. Please note that a catenate source can not contain additional sub-catenates, only data or blob sources.

It is envisaged that catenate sources might be extended in future, for example to fetch external content.

A server SHOULD accept at least 64 catenate items.

4.1.2. update

It is not possible to update a Blob, so any update will result in a notUpdated response.

4.1.3. destroy

It is not possible to destroy a Blob, so any destroy will result in a notDestroyed response.

4.2. Blob/set examples

The data:asBase64 field is set over multiple lines for ease of publication here, however all data:asBase64 would be sent as a continuous string with no whitespace on the wire.

Method Call:

```
[
  "Blob/set",
  {
    "accountId": "account1",
    "create": {
      "1": {
        "data:asBase64": "iVBORw0KGgoAAAANSUhEUgAAAAEAAAABAQMAAAAl21bKA
                          AAAA1BMVEX/AAAZ4gk3AAAAAXRSTlN/gFy0ywAAAApJRE
                          FUeJxjYgAAAAYAAzY3fKgAAAAASUVORK5CYII=",
        "type": "image/png"
      },
    },
  },
  "R1"
]
```

Response:

```
[
  "Blob/set",
  {
    "accountId" : "account1",
    "created" : {
      "1": {
        "id" : "G4c6751edf9dd6903ff54b792e432fba781271beb",
        "type" : "image/png",
        "size" : 95
      },
    },
  },
  "R1"
]
```

Complex catenate example:

Method Calls:


```
[
  [
    "Blob/set",
    {
      "create": {
        "b4": {
          "data:asText": "The quick brown fox jumped over the lazy dog."
        }
      }
    },
    "S4"
  ],
  [
    "Blob/set",
    {
      "create": {
        "cat": {
          "catenate": [
            {
              "data:asText": "How"
            },
            {
              "blobId": "#b4",
              "length": 7,
              "offset": 3
            },
            {
              "data:asText": "was t"
            },
            {
              "blobId": "#b4",
              "length": 1,
              "offset": 1
            },
            {
              "data:asBase64": "YXQ/"
            }
          ]
        }
      }
    },
    "CAT"
  ],
  [
    "Blob/get",
    {
      "properties": [
```

```
        "data:asText",
        "size"
      ],
      "ids": [
        "#cat"
      ]
    },
    "G4"
  ]
]

Responses:

[
  [
    "Blob/set",
    {
      "oldState": null,
      "created": {
        "b4": {
          "id": "Gc0854fb9fb03c41cce3802cb0d220529e6eef94e",
          "blobId": "Gc0854fb9fb03c41cce3802cb0d220529e6eef94e",
          "size": 45,
          "type": "application/octet-stream"
        }
      },
      "updated": null,
      "destroyed": null,
      "notCreated": null,
      "notUpdated": null,
      "notDestroyed": null,
      "accountId": "cassandane"
    },
    "S4"
  ],
  [
    "Blob/set",
    {
      "oldState": null,
      "created": {
        "cat": {
          "id": "Gcc60576f036321ae6e8037ffc56bdee589bd3e23",
          "blobId": "Gcc60576f036321ae6e8037ffc56bdee589bd3e23",
          "size": 19,
          "type": "application/octet-stream"
        }
      },
      "updated": null,
      "destroyed": null,
```

```
      "notCreated": null,
      "notUpdated": null,
      "notDestroyed": null,
      "accountId": "cassandane"
    },
    "CAT"
  ],
  [
    "Blob/get",
    {
      "list": [
        {
          "id": "Gcc60576f036321ae6e8037ffc56bdee589bd3e23",
          "data:asText": "How quick was that?",
          "size": 19
        }
      ],
      "notFound": [],
      "accountId": "cassandane"
    },
    "G4"
  ]
]
```

4.3. Blob/get

A standard JMAP get, with two additional optional parameters:

*offset: UnsignedInt|null

start this many octets into the blob data

*length: UnsignedInt|null

return at most this many octets of the blob data

Request Properties:

Any of

*data:asText

*data:asBase64

*data (returns data:asText if the selected octets are valid UTF-8, or data:asBase64)

*size

If not given, properties defaults to data and size.

Result Properties:

*data:asText: String|null

the raw octets of the selected range if they are valid UTF-8, otherwise null

*data:asBase64: String

the base64 encoding of the selected range

*isEncodingProblem: Boolean (default: false)

*isTruncated: Boolean (default: false)

*size: UnsignedInt

the number of octets in the entire blob, regardless of offset/length selectors

The size value is always the number of octets in the entire blob, regardless of offset and length.

The data fields contain a representation of the octets within the selected range that are present in the blob. If the octets selected are not valid UTF-8 (including truncating in the middle of a multi-octet sequence) and data:asText is requested, then the key isEncodingProblem is set to true and the data:asText value is null. In the case where data was requested and the data is not valid UTF-8, then data:asBase64 is returned.

If the selected range requests data outside the blob (i.e. the offset+length is larger than the blob) then the result is either just the octets from the offset to the end of the blob, or an empty string if the offset is past the end of the blob. Either way, the isTruncated property in the result is set to true to tell the client that the requested range could not be fully satisfied.

Example (a blob containing the string "The quick brown fox jumped over the lazy dog!")

Method Call:

```
[
  "Blob/get",
  {
    "accountId" : "account1",
    "ids" : [
      "G6ec94756e3e046be78fcb33953b85b944e70673e",
      "not-a-blob"
    ],
    "properties" : [
      "data:asText",
      "data:asBase64",
      "size"
    ],
    "offset" : 4,
    "length" : 9
  },
  "R1"
]
```

Response:

```
[
  "Blob/get",
  {
    "accountId": "account1",
    "list": [
      {
        "id": "G6ec94756e3e046be78fcb33953b85b944e70673e",
        "data:asText": "quick bro",
        "data:asBase64": "cXVpY2sgYnJvCg==",
        "size": 46
      }
    ],
    "notFound": [
      "not-a-blob"
    ]
  },
  "R1"
]
```

4.4. Blob/lookup

Given a list of blobIds, this method does a reverse lookup in each of the provided type names to find the list of Ids within that data type which reference the provided blob.

The definition of reference is somewhat loosely defined, but roughly means "you could discover this blobId by looking inside this object", for example if a Mailbox contains an Email which references the blobId, then it references that blobId. Likewise for a Thread.

Parameters

*accountId: Id

The id of the account used for the call.

*typeNames: [String]

A list of names from the "JMAP Data Types" registry. Only names for which "Can Reference Blobs" is true may be specified, and the capability which defines each type must also be used by the overall JMAP request in which this method is called.

If a type name is not known by the server, or the associated capability has not been requested, then the server returns an "unknownDataType" error.

*ids: [Id]

A list of blobId values to be looked for.

Response

*list: [BlobInfo]

A list of BlobInfo objects.

BlobInfo Object

*id: Id

The Blob Identifier.

*matchedIds: String[Id List]

A map from type name to list of Ids of that data type (e.g. the name "Email" maps to a list of emailIds)

If a blob is not visible to a user at all, then the server SHOULD return that blobId in the notFound array, however it may also return an empty list for each type name, as it may not be able to know if other data types do reference that blob.

e.g.

```
[
  "Blob/lookup",
  {
    "typeNames": [
      "Mailbox",
      "Thread",
      "Email"
    ],
    "ids": [
      "Gd2f81008cf07d2425418f7f02a3ca63a8bc82003",
      "not-a-blob"
    ]
  },
  "R1"
]
```

Response:

```
[
  "Blob/lookup",
  {
    "list": [
      {
        "id": "Gd2f81008cf07d2425418f7f02a3ca63a8bc82003",
        "matchedIds": {
          "Mailbox": [
            "M54e97373",
            "Mcbe6b662"
          ],
          "Thread": [
            "T1530616e"
          ],
          "Email": [
            "E16e70a73eb4",
            "E84b0930cf16"
          ]
        }
      }
    ],
    "notFound": [
      "not-a-blob"
    ]
  },
  "R1"
]
```

5. Security considerations

JSON parsers are not all consistent in handling non-UTF-8 data. JMAP requires that all JSON data be UTF-8 encoded, so servers MUST only return a null value if data:asText is requested for a range of octets which is not valid UTF-8, and set isEncodingProblem: true.

Servers MUST apply any access controls, such that if the authenticated user would be unable to discover the blobId by making queries, then this fact can't be discovered via a Blob/lookup. For example, if an Email exists in a Mailbox which the authenticated user does not have access to see, then that emailId MUST not be returned in a lookup for a blob which is referenced by that email.

If a server might sometimes return all names empty rather than putting a blobId in the notFound response to a Blob/get, then the server SHOULD always return the same type of response, regardless of whether a blob exists but the user can't access it, or doesn't exist at all. This avoids leaking information about the existence of the blob.

The server MUST NOT trust that the data given to a Blob/set is a well formed instance of the specified media type, and if the server attempts to parse the given blob, only hardened parsers designed to deal with arbitrary untrusted data should be used. The server SHOULD NOT reject data on the grounds that it is not a valid specimen of the stated type.

Blob/set catenate can be used to recreate dangerous content on the far side of security scanners (anti-virus or exfiltration scanners for example) which may be watching the upload endpoint. Server implementations SHOULD provide a hook to allow security scanners to check the resulting blobId from a catenate in the same way that they do for the upload endpoint.

6. IANA considerations

6.1. JMAP Capability registration for "blob"

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

Capability Name: urn:ietf:params:jmap:blob

Specification document: this document

Intended use: common

Change Controller: IETF

Security and privacy considerations: this document, Section XXX

6.2. JMAP Error Codes Registration for "unknownDataType"

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

JMAP Error Code: unknownDataType

Intended use: common

Change Controller: IETF

Reference: this document

Description: The server does not recognise this data type, or the capability to enable it was not present.

6.3. Creation of "JMAP Data Types" Registry

IANA is requested to create a new registry "JMAP Data Types" with the initial content:

Type Name	Can Reference Blobs	Can use for State Change	Capability	Reference
Core	No	No	urn:ietf:params:jmap:core	[RFC8620]
PushSubscription	No	No	urn:ietf:params:jmap:core	[RFC8620]
Mailbox	Yes	Yes	urn:ietf:params:jmap:mail	[RFC8621]
Thread	Yes	Yes	urn:ietf:params:jmap:mail	[RFC8621]
Email	Yes	Yes	urn:ietf:params:jmap:mail	[RFC8621]
EmailDelivery	No	Yes	urn:ietf:params:jmap:mail	[RFC8621]
SearchSnippet	No	No	urn:ietf:params:jmap:mail	[RFC8621]
Identity	No	Yes	urn:ietf:params:jmap:submission	[RFC8621]
EmailSubmission	No	Yes	urn:ietf:params:jmap:submission	[RFC8621]
VacationResponse	No	Yes	urn:ietf:params:jmap:vacationresponse	[RFC8621]
MDN	No	No	urn:ietf:params:jmap:mdn	[RFC9007]

Table 1

7. Changes

EDITOR: please remove this section before publication.

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

draft-ietf-jmap-blob-06:

*removed asHex - we only need base64 and text

*added reference to where base64 is defined

*made 'destroy' not be allowed

*expanded JSON examples for readability

*removed 'expires' from examples

draft-ietf-jmap-blob-05:

*discovered I hadn't actually included typeNames and matchedIds anywhere except the updates section, oops!

*added a catenate example

*tightened up some text

draft-ieft-jmap-blob-04:

*added security considerations for scanning catenate results

draft-ieft-jmap-blob-03:

*added capabilities object

*renamed types to typeNames and matchedIds

*added details of how to handle non-UTF8 data and truncation in Blob/get

*added isTruncated and isEncodingProblem to Blob/get to tell the client if the request wasn't entirely satisfied.

draft-ieft-jmap-blob-02:

*fixed incorrect RFC number in reference and HTTP PUT -> POST, thanks Ken.

*added acknowledgements section

*removed all 'datatype' text and changed to 'data type' or 'type name' as appropriate (issue #1 proposal)

*expanded security considerations section and moved optional Blob/lookup empty case into Blob/lookup section

draft-ieft-jmap-blob-01:

*renamed 'datatypes' to 'types' to align with PushSubscription from RFC8620.

*added example for Blob/get

*specified offset and length precisely

draft-ieft-jmap-blob-00:

*initial adoption as an IETF document, otherwise identical to draft-gondwana-jmap-blob-02

draft-gondwana-jmap-blob-02

*renamed 'objects' to 'datatypes'

*specified Blob/lookup

*added IANA registry for datatypes

draft-gondwana-jmap-blob-01

*added an example

draft-gondwana-jmap-blob-00

*initial proposal

8. Acknowledgements

Joris Baum, Neil Jenkiuns, Alexey Melnikov, Ken Murchison, Robert Stepanek and the JMAP working group at the IETF.

9. Normative References

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

[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/info/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/info/rfc8174>.

10. Informative References

[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/info/rfc8620>.

[RFC7888] Melnikov, A., Ed., "IMAP4 Non-synchronizing Literals", RFC 7888, DOI 10.17487/RFC7888, May 2016, <https://www.rfc-editor.org/info/rfc7888>.

[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/info/rfc8621>.

Author's Address

Bron Gondwana (editor)
Fastmail
Level 2, 114 William St
Melbourne VIC 3000
Australia

Email: brong@fastmailteam.com
URI: https://www.fastmail.com