

JMAP
Internet-Draft
Intended status: Standards Track
Expires: September 19, 2019

N. Jenkins
FastMail
C. Newman
Oracle
March 18, 2019

JSON Meta Application Protocol
draft-ietf-jmap-core-17

Abstract

This document specifies a protocol for clients to efficiently query, fetch and modify JSON-based data objects, with support for push notification of changes and fast resynchronisation, and out-of-band binary data upload/download.

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 September 19, 2019.

Copyright Notice

Copyright (c) 2019 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	3
1.1.	Notational conventions	4
1.2.	The Id data type	5
1.3.	The Int and UnsignedInt data types	6
1.4.	The Date and UTCDate data types	6
1.5.	JSON as the data encoding format	6
1.6.	Terminology	7
1.6.1.	User	7
1.6.2.	Accounts	7
1.6.3.	Data types and records	7
1.7.	The JMAP API model	8
1.8.	Vendor-specific extensions	8
2.	The JMAP Session resource	9
2.1.	Example	12
2.2.	Service autodiscovery	14
3.	Structured data exchange	14
3.1.	Making an API request	14
3.1.1.	The Invocation data type	15
3.2.	The Request object	15
3.2.1.	Example request	16
3.3.	The Response object	16
3.3.1.	Example response:	17
3.4.	Omitting arguments	17
3.5.	Errors	18
3.5.1.	Request-level errors	18
3.5.2.	Method-level errors	19
3.6.	References to previous method results	21
3.7.	Localisation of user-visible strings	25
3.8.	Security	26
3.9.	Concurrency	26
4.	The Core/echo method	26
4.1.	Example	26
5.	Standard methods and naming convention	26
5.1.	/get	27
5.2.	/changes	28
5.3.	/set	31
5.4.	/copy	36
5.5.	/query	38
5.6.	/queryChanges	43
5.7.	Examples	46
5.8.	Proxy considerations	52
6.	Binary data	53
6.1.	Uploading binary data	54
6.2.	Downloading binary data	55
6.3.	Blob/copy	55
7.	Push	56

7.1. The StateChange object	57
7.1.1. Example	57
7.2. PushSubscription	58
7.2.1. PushSubscription/get	60
7.2.2. PushSubscription/set	61
7.2.3. Example	62
7.3. Event Source	64
8. Security considerations	65
8.1. Transport confidentiality	65
8.2. Authentication scheme	66
8.3. Service autodiscovery	66
8.4. JSON parsing	66
8.5. Denial of service	67
8.6. Connection to unknown push server	67
8.7. Push encryption	68
8.8. Traffic analysis	68
9. IANA considerations	69
9.1. Assignment of jmap service name	69
9.2. Registration of well-known URI suffix for JMAP	69
9.3. Registration of the jmap URN sub-namespace	69
9.4. Creation of "JMAP Capabilities" registry	70
9.4.1. Preliminary community review	70
9.4.2. Submit request to IANA	71
9.4.3. Designated expert review	71
9.4.4. Change procedures	71
9.4.5. JMAP Capabilities registry template:	72
9.4.6. Initial registration for JMAP core	72
9.4.7. Registration for JMAP error placeholder in JMAP capabilities registry	72
9.5. Creation of "JMAP Error Codes" registry	72
9.5.1. Designated expert review	73
9.5.2. JMAP Error Codes registry template:	73
9.5.3. Initial JMAP Error Codes registry	74
10. References	81
10.1. Normative References	81
10.2. Informative References	85
Authors' Addresses	85

1. Introduction

JMAP is a protocol for synchronising data, such as mail, calendars or contacts, between a client and a server. It is optimised for mobile and web environments, and aims to provide a consistent interface to different data types.

This specification is for the generic mechanism of data synchronisation. Further specifications define the data models for different data types that may be synchronised via JMAP.

JMAP is designed to make efficient use of limited network resources. Multiple API calls may be batched in a single request to the server, reducing round trips and improving battery life on mobile devices. Push connections remove the need for polling, and an efficient delta update mechanism ensures a minimum of data is transferred.

JMAP is designed to be horizontally scalable to a very large number of users. This is facilitated by separate end points for users after login, the separation of binary and structured data, and a data model for sharing that does not allow data dependencies between accounts.

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.

The underlying format used for this specification is JSON. Consequently, the terms "object" and "array" as well as the four primitive types (strings, numbers, booleans, and null) are to be interpreted as described in section 1 of [RFC8259]. Unless otherwise noted, all the property names and values are case sensitive.

Some examples in this document contain "partial" JSON documents used for illustrative purposes. In these examples, three periods "..." are used to indicate a portion of the document that has been removed for compactness.

For compatibility with publishing requirements, line breaks have been inserted inside long JSON strings, with the following continuation lines indented. To form the valid JSON example, any line breaks inside a string must be replaced with a space, and any other white-space after the line break removed.

Unless otherwise specified, examples of API exchanges only show the `_methodCalls_` array of the Request object or the `_methodResponses_` array of the Response object. For compactness, the rest of the Request/Response object is omitted.

Type signatures are given for all JSON values in this document. The following conventions are used:

- o "*" - The type is undefined (the value could be any type, although permitted values may be constrained by the context of this value).
- o "String" - The JSON string type.

- o "Number" - The JSON number type.
- o "Boolean" - The JSON boolean type.
- o "A[B]" - A JSON object where the keys are all of type "A", and the values are all of type "B".
- o "A[]" - An array of values of type "A".
- o "A|B" - The value is either of type "A" or of type "B".

Other types may also be given, with their representation defined elsewhere in this document.

Object properties may also have a set of attributes defined along with the type signature. These have the following meanings:

- o **server-set**: Only the server can set the value for this property. The client **MUST NOT** send this property when creating a new object of this type.
- o **immutable**: The value **MUST NOT** change after the object is created.
- o **default**: (This is followed by a JSON value). The value that will be used for this property if it is omitted in an argument, or when creating a new object of this type.

1.2. The Id data type

All record ids are assigned by the server, and are immutable.

Where "Id" is given as a datatype, it means a "String" of at least 1 and maximum 255 octets in size, and **MUST** only contain characters from the "URL and Filename safe" Base 64 Alphabet, as defined in section 5 of [RFC4648], excluding the pad character ("="). This means the allowed characters are the ASCII alphanumeric characters ("A-Za-z0-9"), hyphen ("-"), and underscore ("_").

These characters are safe to use in almost any context (e.g., filesystems, URIs, IMAP atoms). For maximum safety, servers **SHOULD** also follow defensive allocation strategies to avoid creating risks where glob completion or data type detection may be present (e.g., on filesystems or in spreadsheets). In particular, it is wise to avoid:

- o Ids starting with a dash
- o Ids starting with digits

- o Ids that contain only digits
- o Ids that differ only by ASCII case (for example, A vs. a)
- o the specific sequence of three characters "NIL" (because this sequence can be confused with the IMAP protocol expression of the null value)

A good solution to these issues is to prefix every id with a single alphabetical character.

1.3. The Int and UnsignedInt data types

Where "Int" is given as a data type, it means an integer in the range $-2^{53}+1 \leq \text{value} \leq 2^{53}-1$, the safe range for integers stored in a floating-point double, represented as a JSON "Number".

Where "UnsignedInt" is given as a data type, it means an "Int" where the value MUST be in the range $0 \leq \text{value} \leq 2^{53}-1$.

1.4. The Date and UTCDate data types

Where "Date" is given as a type, it means a string in [RFC3339] `_date-time_` format. To ensure a normalised form, the `_time-secfrac_` MUST always be omitted if zero, and any letters in the string (e.g. "T" and "Z") MUST be upper-case. For example, `"2014-10-30T14:12:00+08:00"`.

Where "UTCDate" is given as a type, it means a "Date" where the `_time-offset_` component MUST be "Z" (i.e. it must be in UTC time). For example, `"2014-10-30T06:12:00Z"`.

1.5. JSON as the data encoding format

JSON is a text-based data interchange format as specified in [RFC8259]. The I-JSON format defined in [RFC7493] is a strict subset of this, adding restrictions to avoid potentially confusing scenarios (for example, it mandates that an object MUST NOT have two members with the same name).

All data sent from the client to the server or from the server to the client (except binary file upload/download) MUST be valid I-JSON according to the RFC, and is therefore case-sensitive and encoded in UTF-8 ([RFC3629]).

1.6. Terminology

1.6.1. User

A user is a person accessing data via JMAP. A user has a set of permissions determining the data that they can see.

1.6.2. Accounts

An account is a collection of data. A single account may contain an arbitrary set of data types, for example a collection of mail, contacts and calendars. Most JMAP methods take a mandatory `_accountId_` argument that specifies on which account the operations are to take place.

An account is not the same as a user, although it is common for a primary account to directly belong to the user. For example, you may have an account that contains data for a group or business, to which multiple users have access.

A single set of credentials may provide access to multiple accounts, for example if another user is sharing their work calendar with the authenticated user, or if there is a group mailbox for a support-desk inbox.

In the event of a severe internal error, a server may have to reallocate ids or do something else that violates standard JMAP data constraints for an account. In this situation, the data on the server is no longer compatible with cached data the client may have from before. The server **MUST** treat this as though the account has been deleted and then recreated with a new account id. Clients will then be forced to throw away any data with the old account id and refetch all data from scratch.

1.6.3. Data types and records

JMAP provides a uniform interface for creating, retrieving, updating and deleting various types of objects. A **data type** is a collection of named, typed properties, just like the schema for a database table. Each instance of a data type is called a **record**.

The id of a record is immutable, and assigned by the server. The id **MUST** be unique among all records of the **same type** within the **same account**. Ids may clash across accounts, or for two records of different types within the same account.

1.7. The JMAP API model

JMAP uses HTTP [RFC7230] to expose API, Push, Upload and Download resources. All HTTP requests MUST use the "https://" scheme ([RFC2818] HTTP over TLS). All HTTP requests MUST be authenticated.

An authenticated client can fetch the user's JMAP Session object with details about the data and capabilities the server can provide as shown in section 2. The client may then exchange data with the server in the following ways:

1. The client may make an API request to the server to get or set structured data. This request consists of an ordered series of method calls. These are processed by the server, which then returns an ordered series of responses. This is described in sections 3 to 5.
2. The client may download or upload binary files from/to the server. This is detailed in section 6.
3. The client may connect to a push channel on the server, to be notified when data has changed. This is explained in section 7.

1.8. Vendor-specific extensions

Individual services will have custom features they wish to expose over JMAP. This may take the form of extra data types and/or methods not in the spec, or extra arguments to JMAP methods, or extra properties on existing data types (which may also appear in arguments to methods that take property names).

The server can advertise custom extensions it supports by including the identifiers in the capabilities object. Identifiers for vendor extensions MUST be a URL belonging to a domain owned by the vendor, to avoid conflict. The URL SHOULD resolve to documentation for the changes the extension makes.

To ensure compatibility with clients that don't know about a specific custom extension, and for compatibility with future versions of JMAP, to use an extension the client MUST opt in by passing the appropriate capability identifier in the `_using_` array of the Request object, as described in section 3.2. The server MUST only follow the specifications that are opted-into and behave as though it does not implement anything else when processing a request.

2. The JMAP Session resource

You need two things to connect to a JMAP server:

1. The URL for the JMAP Session resource. This may be requested directly from the user, or discovered automatically based on a username domain (see section 2.2 below).
2. Credentials to authenticate with. How to obtain credentials is out of scope for this document.

An authenticated GET request to the JMAP Session resource MUST return the details about the data and capabilities the server can provide to the client given those credentials.

The response to a successful request is a JSON-encoded **JMAP Session** object. It has the following properties:

- o **capabilities**: "String[Object]" An object specifying the capabilities of this server. Each key is a URI for a capability supported by the server. The value for each of these keys is an object with further information about the server's capabilities in relation to that capability.

The client MUST ignore any properties it does not understand.

The capabilities object MUST include a property called "urn:ietf:params:jmap:core". The value of this property is an object which MUST contain the following information on server capabilities (suggested minimum values for limits are supplied that allow clients to make efficient use of the network):

- * **maxSizeUpload**: "UnsignedInt" The maximum file size, in octets, that the server will accept for a single file upload (for any purpose). Suggested minimum: 50,000,000.
- * **maxConcurrentUpload**: "UnsignedInt" The maximum number of concurrent requests the server will accept to the upload endpoint. Suggested minimum: 4.
- * **maxSizeRequest**: "UnsignedInt" The maximum size, in octets, that the server will accept for a single request to the API endpoint. Suggested minimum: 10,000,000.
- * **maxConcurrentRequests**: "UnsignedInt" The maximum number of concurrent requests the server will accept to the API endpoint. Suggested minimum: 4.

- * `*maxCallsInRequest*`: "UnsignedInt" The maximum number of method calls the server will accept in a single request to the API endpoint. Suggested minimum: 16.
- * `*maxObjectsInGet*`: "UnsignedInt" The maximum number of objects that the client may request in a single `"/get"` type method call. Suggested minimum: 500
- * `*maxObjectsInSet*`: "UnsignedInt" The maximum number of objects the client may send to create, update or destroy in a single `"/set"` type method call. This is the combined total, e.g. if the maximum is 10 you could not create 7 objects and destroy 6, as this would be 13 actions, which exceeds the limit. Suggested minimum: 500.
- * `*collationAlgorithms*`: "String[]" A list of identifiers for algorithms registered in the collation registry defined in [RFC4790] that the server supports for sorting when querying records.

Specifications for future capabilities will define their own properties on the capabilities object.

Servers MAY advertise vendor-specific JMAP extensions, as described in section 1.8. To avoid conflict, an identifier for a vendor-specific extension MUST be a URL with a domain owned by the vendor. Clients MUST opt in to any capability it wishes to use (see section 3.2).

- o `*accounts*`: "Id[Account]" A map of `*account id*` to Account object for each account (see section 1.5.2) the user has access to. An `*Account*` object has the following properties:
 - * `*name*`: "String" A user-friendly string to show when presenting content from this account, e.g. the email address representing the owner of the account.
 - * `*isPersonal*`: "Boolean" This is "true" if the account belongs to the authenticated user, rather than a group account or a personal account of another user that has been shared with them.
 - * `*isReadOnly*`: "Boolean" This is "true" if the entire account is read-only.
 - * `*accountCapabilities*`: "String[Object]" The set of capability URIs for the methods supported in this account. Each key is a URI for a capability that has methods you can use with this

account. The value for each of these keys is an object with further information about the account's permissions and restrictions with respect to this capability, as defined in the capability's specification.

The client MUST ignore any properties it does not understand.

The server advertises the full list of capabilities it supports in the capabilities object, as defined above. If the capability defines new methods, the server MUST include it in the `_accountCapabilities_` object if the user may use those methods with this account. It MUST NOT include it in the `_accountCapabilities_` object if the user cannot use those methods with this account.

For example, you may have access to your own account with mail, calendars and contacts data, and also a shared account that only has contacts data (a business address book for example). In this case the `_accountCapabilities_` property on the first account would include something like `"urn:ietf:params:jmap:mail"`, `"urn:ietf:params:jmap:calendars"`, `"urn:ietf:params:jmap:contacts"`, while the second account would just have the last of these.

Attempts to use the methods defined in a capability with one of the accounts that does not support that capability are rejected with an `_accountNotSupportedByMethod_` error (see section 3.5.2: method-level errors).

- o `*primaryAccounts*`: "String[Id]" A map of capability URIs (as found in `_accountCapabilities_`) to the account id to be considered the user's main or default account for data pertaining to that capability. If no account being returned belongs to the user, or in any other way there is no appropriate way to determine a default account, there MAY be no entry for a particular URI, even though that capability is supported by the server (and in the capabilities object). `"urn:ietf:params:jmap:core"` SHOULD NOT be present.
- o `*username*`: "String" The username associated with the given credentials, or the empty string if none.
- o `*apiUrl*`: "String" The URL to use for JMAP API requests.
- o `*downloadUrl*`: "String" The URL endpoint to use when downloading files, in [RFC6570] URI Template (level 1) format. The URL MUST contain variables called `"accountId"`, `"blobId"`, `"type"` and `"name"`. The use of these variables is described in section 6.2. Due to

potential encoding issues with slashes in content types, it is RECOMMENDED to put the "type" variable in the query section of the URL.

- o *uploadUrl*: "String" The URL endpoint to use when uploading files, in [RFC6570] URI Template (level 1) format. The URL MUST contain a variable called "accountId". The use of this variable is described in section 6.1.
- o *eventSourceUrl*: "String" The URL to connect to for push events, as described in section 7.3, in [RFC6570] URI Template (level 1) format. The URL MUST contain variables called "types", "closeafter" and "ping". The use of these variables is described in section 7.3.
- o *state*: "String" A (preferably short) string representing the state of this object on the server. If the value of any other property on the session object changes, this string will change. The current value is also returned on the API Response object (see section 3.3), allowing clients to quickly determine if the session information has changed (e.g. an account has been added or removed) and so they need to refetch the object.

To ensure future compatibility, other properties MAY be included on the JMAP Session object. Clients MUST ignore any properties they are not expecting.

Implementors must take care to avoid inappropriate caching of the session object at the HTTP layer. Since the client should only refetch when it detects there is a change (via the sessionState property of an API response), it is RECOMMENDED to disable HTTP caching altogether, for example by setting "Cache-Control: no-cache, no-store, must-revalidate" on the response.

2.1. Example

In the following example JMAP Session object, the user has access to their own mail and contacts via JMAP, as well as read-only access to shared mail from another user. The server is advertising a custom "https://example.com/apis/foobar" capability.

```
{
  "capabilities": {
    "urn:ietf:params:jmap:core": {
      "maxSizeUpload": 50000000,
      "maxConcurrentUpload": 8,
      "maxSizeRequest": 10000000,
      "maxConcurrentRequest": 8,
```

```
    "maxCallsInRequest": 32,
    "maxObjectsInGet": 256,
    "maxObjectsInSet": 128,
    "collationAlgorithms": [
      "i;ascii-numeric",
      "i;ascii-casemap",
      "i;unicode-casemap"
    ]
  },
  "urn:ietf:params:jmap:mail": {},
  "urn:ietf:params:jmap:contacts": {},
  "https://example.com/apis/foobar": {
    "maxFoosFinangled": 42
  }
},
"accounts": {
  "A13824": {
    "name": "john@example.com",
    "isPersonal": true,
    "isReadOnly": false,
    "accountCapabilities": {
      "urn:ietf:params:jmap:mail": {
        "maxMailboxesPerEmail": null,
        "maxMailboxDepth": 10,
        ...
      },
      "urn:ietf:params:jmap:contacts": {
        ...
      }
    }
  },
  "A97813": {
    "name": "jane@example.com",
    "isPersonal": false,
    "isReadOnly": true,
    "accountCapabilities": {
      "urn:ietf:params:jmap:mail": {
        "maxMailboxesPerEmail": 1,
        "maxMailboxDepth": 10,
        ...
      }
    }
  }
},
"primaryAccounts": {
  "urn:ietf:params:jmap:mail": "A13824",
  "urn:ietf:params:jmap:contacts": "A13824"
},
```

```
"username": "john@example.com",
"apiUrl": "https://jmap.example.com/api/",
"downloadUrl": "https://jmap.example.com
  /download/{accountId}/{blobId}/{name}?accept={type}",
"uploadUrl": "https://jmap.example.com/upload/{accountId}/",
"eventSourceUrl": "https://jmap.example.com
  /eventsources/?types={types}&closeafter={closeafter}&ping={ping}",
"state": "75128aab4b1b"
}
```

2.2. Service autodiscovery

There are two standardised autodiscovery methods in use for internet protocols:

- o *DNS SRV* ([RFC2782], [RFC6186] and [RFC6764])
- o *.well-known/servicename* ([RFC5785])

A JMAP-supporting host for the domain "example.com" SHOULD publish a SRV record "_jmap._tcp.example.com" which gives a `_hostname_` and `_port_` (usually port "443"). The JMAP Session resource is then "https://`{hostname}`[:`{port}`]/.well-known/jmap" (following any redirects).

If the client has a username in the form of an email address, it MAY use the domain portion of this to attempt autodiscovery of the JMAP server.

3. Structured data exchange

The client may make an API request to the server to get or set structured data. This request consists of an ordered series of method calls. These are processed by the server, which then returns an ordered series of responses.

3.1. Making an API request

To make an API request, the client makes an authenticated POST request to the API resource, which is defined by the `_apiUrl_` property in the JMAP Session object.

The request MUST be of type "application/json" and consist of a single JSON **Request** object, as defined in section 3.2. If successful, the response MUST also be of type "application/json" and consist of a single **Response** object, as defined in section 3.3.

3.1.1. The Invocation data type

Method calls and responses are represented by the **Invocation** data type. This is a tuple, represented as a JSON array containing three elements:

1. A "String" **name** of the method to call or of the response.
2. A "String[*]" object containing *_named_* **arguments** for that method or response.
3. A "String" **method call id**: an arbitrary string from the client to be echoed back with the responses emitted by that method call (a method may return 1 or more responses, as it may make implicit calls to other methods; all responses initiated by this method call get the same method call id in the response).

3.2. The Request object

A **Request** object has the following properties:

- o **using**: "String[]" The set of capabilities the client wishes to use. The client MAY include capability identifiers even if the method calls it makes do not utilise those capabilities. The server advertises the set of specifications it supports in the JMAP Session object, as keys on the *_capabilities_* property.
- o **methodCalls**: "Invocation[]" An array of method calls to process on the server. The method calls MUST be processed sequentially, in order.
- o **createdIds**: "Id[Id]" (optional) A map of (client-specified) creation id to the id the server assigned when a record was successfully created.

As described later in this specification, some records may have a property that contains the id of another record. To allow more efficient network usage, you can set this property to reference a record created earlier in the same API request. Since the real id is unknown when the request is created, the client can instead specify the creation id it assigned, prefixed with a "#" (see section 5.3 for more details).

As the server processes API requests, any time it successfully creates a new record it adds to this map the creation id (see the *_create_* argument to *"/set"* in section 5.3), with the server-assigned real id as the value. If it comes across a reference to

a creation id in a create/update, it looks it up in the map and replaces the reference with the real id, if found.

The client can pass an initial value for this map as the `_createdIds_` property of the Request. This may be an empty object. If given in the request, the response will also include a `createdIds` property. This allows proxy servers to easily split a JMAP request into multiple JMAP requests to send to different servers. For example it could send the first two method calls to server A, then the third to server B, before sending the fourth to server A again. By passing the `createdIds` of the previous response to the next request, it can ensure all of these still resolve. See section 5.8 for further discussion of proxy considerations.

Future specifications MAY add further properties to the Request object to extend the semantics. To ensure forwards compatibility, a server MUST ignore any other properties it does not understand on the JMAP request object.

3.2.1. Example request

```
{
  "using": [ "urn:ietf:params:jmap:core", "urn:ietf:params:jmap:mail" ],
  "methodCalls": [
    [ "method1", {
      "arg1": "arg1data",
      "arg2": "arg2data"
    }, "c1" ],
    [ "method2", {
      "arg1": "arg1data"
    }, "c2" ],
    [ "method3", {}, "c3" ]
  ]
}
```

3.3. The Response object

A **Response** object has the following properties:

- o **methodResponses**: "Invocation[]" An array of responses, in the same format as the `_methodCalls_` on the request object. The output of the methods MUST be added to the `_methodResponses_` array in the same order as the methods are processed.
- o **createdIds**: "Id[Id]" (optional; only returned if given in request) A map of (client-specified) creation id to the id the server assigned when a record was successfully created. This MUST

include all creation ids passed in the original createdIds parameter of the Request object, as well as any additional ones added for newly created records.

- o *sessionState*: "String" The current value of the "state" string on the JMAP Session object, as described in section 2. Clients may use this to detect if this object has changed and needs to be refetched.

Unless otherwise specified, if the method call completed successfully its response name is the same as the method name in the request.

3.3.1. Example response:

```
{
  "methodResponses": [
    [ "method1", {
      "arg1": 3,
      "arg2": "foo"
    }, "c1" ],
    [ "method2", {
      "isBlah": true
    }, "c2" ],
    [ "anotherResponseFromMethod2", {
      "data": 10,
      "yetmoredata": "Hello"
    }, "c2"],
    [ "error", {
      "type": "unknownMethod"
    }, "c3" ]
  ],
  "sessionState": "75128aab4b1b"
}
```

3.4. Omitting arguments

An argument to a method may be specified to have a default value. If omitted by the client, the server MUST treat the method call the same as if the default value had been specified. Similarly, the server MAY omit any argument in a response which has the default value.

Unless otherwise specified in a method description, "null" is the default value for any argument in a request or response where this is allowed by the type signature. Other arguments may only be omitted if an explicit default value is defined in the method description.

3.5. Errors

There are three different levels of granularity at which an error may be returned in JMAP.

When an API request is made, the request as a whole may be rejected due to rate limiting, malformed JSON, request for an unknown capability etc. In this case the entire request is rejected with an appropriate HTTP error response code, and an additional JSON body with more detail for the client.

Provided the request itself is syntactically valid (the JSON is valid, and when decoded matches the type signature of a Request object), the methods within it are executed sequentially by the server. Each method may individually fail, for example if invalid arguments are given, or an unknown method name is called.

Finally, methods that make changes to the server state often act upon a number of different records within a single call. Each record change may be separately rejected with a SetError, as described in section 5.3.

3.5.1. Request-level errors

When an HTTP error response is returned to the client, the server SHOULD return a JSON "problem details" object as the response body, as per [RFC7807].

The following problem types are defined:

- o "urn:ietf:params:jmap:error:unknownCapability" The client included a capability in the "using" property of the request that the server does not support.
- o "urn:ietf:params:jmap:error:notJSON" The content type of the request was not "application/json" or the request did not parse as I-JSON.
- o "urn:ietf:params:jmap:error:notRequest" The request parsed as JSON but did not match the type signature of the Request object.
- o "urn:ietf:params:jmap:error:limit" The request was not processed as it would have exceeded one of the *request* limits defined on the capability object, such as maxSizeRequest, maxCallsInRequest or maxConcurrentRequests. A "limit" property MUST also be present on the "problem details" object, containing the name of the limit being applied.

3.5.1.1. Example

```
{
  "type": "urn:ietf:params:jmap:error:unknownCapability",
  "status": 400,
  "detail": "The request object used capability
    'https://example.com/apis/foobar', which is not supported
    by this server."
}
```

Another example:

```
{
  "type": "urn:ietf:params:jmap:error:limit",
  "limit": "maxSizeRequest",
  "status": 400,
  "detail": "The request is larger than the server is willing to process."
}
```

3.5.2. Method-level errors

If a method encounters an error, the appropriate "error" response MUST be inserted at the current point in the `_methodResponses_` array and, unless otherwise specified, further processing MUST NOT happen within that method call.

Any further method calls in the request MUST then be processed as normal. Errors at the method level MUST NOT generate an HTTP-level error.

An "error" response looks like this:

```
[ "error", {
  "type": "unknownMethod"
}, "call-id" ]
```

The response name is "error", and it MUST have a type property. Other properties may be present with further information; these are detailed in the error type descriptions where appropriate.

With the exception of when the "serverPartialFail" error is returned, the externally-visible state of the server MUST NOT have changed if an error is returned at the method level.

The following error types are defined which may be returned for any method call where appropriate:

"serverUnavailable": Some internal server resource was temporarily unavailable. Attempting the same operation later (perhaps after a backoff with a random factor) may succeed.

"serverFail": An unexpected or unknown error occurred during the processing of the call. A `_description_` property should provide more details about the error. The method call made no changes to the server's state. Attempting the same operation again is expected to fail again. Contacting the service administrator is likely necessary to resolve this problem if it is persistent.

"serverPartialFail": Some, but not all expected changes described by the method occurred. The client **MUST** re-synchronise impacted data to determine server state. Use of this error is strongly discouraged.

"unknownMethod": The server does not recognise this method name.

"invalidArguments": One of the arguments is of the wrong type or otherwise invalid, or a required argument is missing. A `"description"` property **MAY** be present to help debug with an explanation of what the problem was. This is a non-localised string, and is not intended to be shown directly to end users.

"invalidResultReference": The method used a result reference for one of its arguments (see section 3.6), but this failed to resolve.

"forbidden": The method and arguments are valid, but executing the method would violate an ACL or other permissions policy.

"accountNotFound": The `_accountId_` does not correspond to a valid account.

"accountNotSupportedByMethod": The `_accountId_` given corresponds to a valid account, but the account does not support this method or data type.

"accountReadOnly": This method call would modify state in an account that is read-only (as returned on the corresponding Account object in the JMAP Session resource).

Further possible errors for a particular method are specified in the method descriptions.

Further general errors **MAY** be defined in future RFCs. Should a client receive an error type it does not understand, it **MUST** treat it the same as the "serverFail" type.

3.6. References to previous method results

To allow clients to make more efficient use of the network and avoid round trips, an argument to one method can be taken from the result of a previous method call in the same request.

To do this, the client prefixes the argument name with "#" (an octothorpe). The value is a `_ResultReference_` object as described below. When processing a method call, the server MUST first check the arguments object for any names beginning with "#". If found, the result reference should be resolved and the value used as the "real" argument. The method is then processed as normal. If any result reference fails to resolve, the whole method MUST be rejected with an "invalidResultReference" error. If an argument object contains the same argument name in normal and referenced form (e.g. "foo" and "#foo"), the method MUST return an "invalidArguments" error.

A `*ResultReference*` object has the following properties:

- o `*resultOf*`: "String" The method call id of the method call to get the result from (the string given as the third item in the array for a method call).
- o `*name*`: "String" The expected name of the response.
- o `*path*`: "String" A pointer into the arguments. This is an [RFC6901] JSON Pointer, except it also allows the use of "*" to map through an array (see description below).

To resolve:

1. Find the first response with a method call id identical to the `_resultOf_` property of the `_ResultReference_` in the `_methodResponses_` array from previously processed method calls in the same request. If none, evaluation fails.
2. If the response name is not identical to the `_name_` property of the `_ResultReference_`, evaluation fails.
3. Apply the `_path_` to the arguments object of the response (the second item in the response array) following the [RFC6901] JSON Pointer algorithm, except with the following addition in section 4 (Evaluation):

If the currently referenced value is a JSON array, the reference token may be exactly the single character "*", making the new referenced value the result of applying the rest of the JSON pointer tokens to every item in the array and returning the

results in the same order in a new array. If the result of applying the rest of the pointer tokens to a value was itself an array, its items should be included individually in the output rather than including the array itself (i.e. the result is flattened from an array of arrays to a single array).

As a simple example, suppose we have the following API request `_methodCalls_`:

```
[[ "Foo/changes", {
  "accountId": "A1",
  "sinceState": "abcdef"
}, "t0" ],
[ "Foo/get", {
  "accountId": "A1",
  "#ids": {
    "resultOf": "t0",
    "name": "Foo/changes",
    "path": "/created"
  }
}, "t1" ]]
```

After executing the first method call the `_methodResponses_` array is:

```
[[ "Foo/changes", {
  "accountId": "A1",
  "oldState": "abcdef",
  "newState": "123456",
  "hasMoreChanges": false,
  "created": [ "f1", "f4" ],
  "updated": [],
  "destroyed": []
}, "t0" ]]
```

To execute the `Foo/get` call, we look through the arguments and find there is one with a `"#"` prefix. To resolve this, we apply the algorithm above:

1. Find the first response with method call id `"t0"`. The `Foo/changes` response fulfills this criterion.
2. Check the response name is the same as in the result reference. It is, so this is fine.
3. Apply the `_path_` as a JSON pointer to the arguments object. This simply selects the `"created"` property, so the result of evaluating is: `"["f1", "f4"]"`

The JMAP server now continues to process the Foo/get call as though the arguments were:

```
{
  "accountId": "A1",
  "ids": [ "f1", "f4" ]
}
```

Now a more complicated example using the JMAP Mail data model: fetch the "from"/"date"/"subject" for every email in the first 10 threads in the Inbox (sorted newest first):

```
[[ "Email/query", {
  "accountId": "A1",
  "filter": { "inMailbox": "id_of_inbox" },
  "sort": [{ "property": "receivedAt", "isAscending": false }],
  "collapseThreads": true,
  "position": 0,
  "limit": 10,
  "calculateTotal": true
}, "t0" ],
[ "Email/get", {
  "accountId": "A1",
  "#ids": {
    "resultOf": "t0",
    "name": "Email/query",
    "path": "/ids"
  },
  "properties": [ "threadId" ]
}, "t1" ],
[ "Thread/get", {
  "accountId": "A1",
  "#ids": {
    "resultOf": "t1",
    "name": "Email/get",
    "path": "/list/*/threadId"
  }
}, "t2" ],
[ "Email/get", {
  "accountId": "A1",
  "#ids": {
    "resultOf": "t2",
    "name": "Thread/get",
    "path": "/list/*/emailIds"
  },
  "properties": [ "from", "receivedAt", "subject" ]
}, "t3" ]]
```

After executing the first 3 method calls the `_methodResponses_` array might be:

```
[ [ "Email/query", {
  "accountId": "A1",
  "queryState": "abcdefg",
  "canCalculateChanges": true,
  "position": 0,
  "total": 101,
  "ids": [ "msg1023", "msg223", "msg110", "msg93", "msg91",
    "msg38", "msg36", "msg33", "msg11", "msg1" ]
}, "t0" ],
[ "Email/get", {
  "accountId": "A1",
  "state": "123456",
  "list": [{
    "id": "msg1023",
    "threadId": "trd194"
  }, {
    "id": "msg223",
    "threadId": "trd114"
  },
  ...
],
  "notFound": []
}, "t1" ],
[ "Thread/get", {
  "accountId": "A1",
  "state": "123456",
  "list": [{
    "id": "trd194",
    "emailIds": [ "msg1020", "msg1021", "msg1023" ]
  }, {
    "id": "trd114",
    "emailIds": [ "msg201", "msg223" ]
  },
  ...
],
  "notFound": []
}, "t2" ] ]
```

So to execute the final Email/get call, we look through the arguments and find there is one with a "#" prefix. To resolve this, we apply the algorithm:

1. Find the first response with method call id "t2". The "Thread/get" response fulfills this criterion.

2. "Thread/get" is the name specified in the result reference, so this is fine.
3. Apply the `_path_` as a JSON pointer to the arguments object. Token-by-token:
 1. "list": get the array of thread objects
 2. "*": for each of the items in the array:
 1. "emailIds": get the array of email ids
 2. Concatenate these into a single array of all the ids in the result.

The JMAP server now continues to process the Email/get call as though the arguments were:

```
{
  "accountId": "A1",
  "ids": [ "msg1020", "msg1021", "msg1023", "msg201", "msg223", ... ],
  "properties": [ "from", "receivedAt", "subject" ]
}
```

The ResultReference performs a similar role to that of the creation id, in that it allows a chained method call to refer to information not available when the request is generated. However, they are different things and not interchangeable; the only commonality is the octothorpe used to indicate them.

3.7. Localisation of user-visible strings

If returning a custom string to be displayed to the user, for example an error message, the server SHOULD use information from the Accept-Language header of the request (as defined in [RFC7231] section 5.3.5) to help determine the choice of localisation if multiple are available. The Content-Language header of the response (see section 3.1.3.2 of [RFC7231]) SHOULD indicate the language being used for user-visible strings.

For example, suppose a request was made with the following header:

```
Accept-Language: fr-CH, fr;q=0.9, de;q=0.8, en;q=0.7, *;q=0.5
```

and a method generated an error to display to the user. The server has translations of the error message in English and German. Looking at the Accept-Language header, the user's preferred language is French. Since we don't have a translation for this, we look at the

next most preferred which is German. We have a German translation so the server returns this, and indicates the language chosen in a Content-Language header like so:

Content-Language: de

3.8. Security

As always, the server must be strict about data received from the client. Arguments need to be checked for validity; a malicious user could attempt to find an exploit through the API. In case of invalid arguments (unknown/insufficient/wrong type for data etc.) the method MUST return an "invalidArguments" error and terminate.

3.9. Concurrency

Method calls within a single request MUST be executed in order. However, method calls from different concurrent API requests may be interleaved. This means that the data on the server may change between two method calls within a single API request.

4. The Core/echo method

The `_Core/echo_` method returns exactly the same arguments as it is given. It is useful for testing you have a valid authenticated connection to a JMAP API endpoint.

4.1. Example

Request:

```
[[ "Core/echo", {  
  "hello": true,  
  "high": 5  
}, "b3ff" ]]
```

Response:

```
[[ "Core/echo", {  
  "hello": true,  
  "high": 5  
}, "b3ff" ]]
```

5. Standard methods and naming convention

JMAP provides a uniform interface for creating, retrieving, updating and deleting objects of a particular type. For a "Foo" data type, records of that type would be fetched via a "Foo/get" call and

modified via a "Foo/set" call. Delta updates may be fetched via a "Foo/changes" call. These methods all follow a standard format as described below.

Some types may not have all these methods. Specifications defining types MUST specify which methods are available for the type.

5.1. /get

Objects of type **Foo** are fetched via a call to `_Foo/get_`.

It takes the following arguments:

- o **accountId**: "Id" The id of the account to use.
- o **ids**: "Id[]|null" The ids of the Foo objects to return. If "null" then **all** records of the data type are returned, if this is supported for that data type and the number of records does not exceed the `_maxObjectsInGet_` limit.
- o **properties**: "String[]|null" If supplied, only the properties listed in the array are returned for each Foo object. If "null", all properties of the object are returned. The id property of the object is **always** returned, even if not explicitly requested. If an invalid property is requested, the call MUST be rejected with an "invalidArguments" error.

The response has the following arguments:

- o **accountId**: "Id" The id of the account used for the call.
- o **state**: "String" A (preferably short) string representing the state on the server for **all** the data of this type in the account (not just the objects returned in this call). If the data changes, this string MUST change. If the Foo data is unchanged, servers SHOULD return the same state string on subsequent requests for this data type. When a client receives a response with a different state string to a previous call, it MUST either throw away all currently cached objects for the type, or call `_Foo/changes_` to get the exact changes.
- o **list**: "Foo[]" An array of the Foo objects requested. This is the **empty array** if no objects were found, or if the `_ids_` argument passed in was also the empty array. The results MAY be in a different order to the `_ids_` in the request arguments. If an identical id is included more than once in the request, the server MUST only include it once in either the `_list_` or `_notFound_` argument of the response.

- o `*notFound*`: "Id[]" This array contains the ids passed to the method for records that do not exist. The array is empty if all requested ids were found, or if the `_ids_` argument passed in was either "null" or the empty array.

The following additional error may be returned instead of the `_Foo/get_` response:

"requestTooLarge": The number of `_ids_` requested by the client exceeds the maximum number the server is willing to process in a single method call.

5.2. `/changes`

When the state of the set of Foo records in an account changes on the server (whether due to creation, updates or deletion), the `_state_` property of the `_Foo/get_` response will change. The `_Foo/changes_` method allows a client to efficiently update the state of its Foo cache to match the new state on the server. It takes the following arguments:

- o `*accountId*`: "Id" The id of the account to use.
- o `*sinceState*`: "String" The current state of the client. This is the string that was returned as the `_state_` argument in the `_Foo/get_` response. The server will return the changes that have occurred since this state.
- o `*maxChanges*`: "UnsignedInt|null" The maximum number of ids to return in the response. The server MAY choose to return fewer than this value, but MUST NOT return more. If not given by the client, the server may choose how many to return. If supplied by the client, the value MUST be a positive integer greater than 0. If a value outside of this range is given, the server MUST reject the call with an "invalidArguments" error.

The response has the following arguments:

- o `*accountId*`: "Id" The id of the account used for the call.
- o `*oldState*`: "String" This is the `_sinceState_` argument echoed back; the state from which the server is returning changes.
- o `*newState*`: "String" This is the state the client will be in after applying the set of changes to the old state.

- o `*hasMoreChanges*`: "Boolean" If "true", the client may call `_Foo/changes_` again with the `_newState_` returned to get further updates. If "false", `_newState_` is the current server state.
- o `*created*`: "Id[]" An array of ids for records which have been created since the old state.
- o `*updated*`: "Id[]" An array of ids for records which have been updated since the old state.
- o `*destroyed*`: "Id[]" An array of ids for records which have been destroyed since the old state.

If a record has been created AND updated since the old state, the server SHOULD just return the id in the `_created_` list, but MAY return it in the `_updated_` list as well.

If a record has been updated AND destroyed since the old state, the server SHOULD just return the id in the `_destroyed_` list, but MAY return it in the `_updated_` list as well.

If a record has been created AND destroyed since the old state, the server SHOULD remove the id from the response entirely, but MAY include it in the `_destroyed_` list, and if so MAY also include it in the `_created_` list.

If a `_maxChanges_` is supplied, or set automatically by the server, the server MUST ensure the number of ids returned across `_created_`, `_updated_` and `_destroyed_` does not exceed this limit. If there are more changes than this between the client's state and the current server state, the server SHOULD generate an update to take the client to an intermediate state, from which the client can continue to call `_Foo/changes_` until it is fully up to date. If it is unable to calculate an intermediate state, it MUST return a "cannotCalculateChanges" error response instead.

When generating intermediate states, the server may choose how to divide up the changes. For many types it will provide a better user experience to return the more recent changes first, as this is more likely to be what the user is most interested in. The client can then continue to page in the older changes while the user is viewing the newer data. For example, suppose a server went through the following states:

A -> B -> C -> D -> E

And a client asks for changes from state "B". The server might first get the ids of records created, updated or destroyed between states D and E, returning them with:

```
state: "B-D-E"
hasMoreChanges: true
```

The client will then ask for the change from state "B-D-E", and the server can return the changes between states C and D, returning:

```
state: "B-C-E"
hasMoreChanges: true
```

Finally the client will request the changes from "B-C-E" and the server can return the changes between states B and C, returning:

```
state: "E"
hasMoreChanges: false
```

Should the state on the server be modified in the middle of all this (to "F"), the server still does the same but now when the update to state "E" is returned, it would indicate that it still has more changes for the client to fetch.

Where multiple changes to a record are split across different intermediate states, the server MUST NOT return a record as created in a later response than one which gives it as updated or destroyed, and MUST NOT return a record as destroyed before a response that gives it as created or updated. The server may have to coalesce multiple changes to a record to satisfy this requirement.

The following additional errors may be returned instead of the `_Foo/changes_` response:

"cannotCalculateChanges": The server cannot calculate the changes from the state string given by the client. Usually due to the client's state being too old, or the server being unable to produce an update to an intermediate state when there are too many updates. The client MUST invalidate its Foo cache.

Maintaining state to allow calculation of `_Foo/changes_` can be expensive for the server, but always returning `_cannotCalculateChanges_` severely increases network traffic and resource usage for the client. To allow efficient sync, servers SHOULD be able to calculate changes from any state string that was given to a client within the last 30 days (but of course may support calculating updates from states older than this).

5.3. /set

Modifying the state of Foo objects on the server is done via the `_Foo/set_` method. This encompasses creating, updating and destroying Foo records. This allows the server to sort out ordering and dependencies that may exist if doing multiple operations at once (for example to ensure there is always a minimum number of a certain record type).

The `_Foo/set_` method takes the following arguments:

- o `*accountId*`: "Id" The id of the account to use.
- o `*ifInState*`: "String|null" This is a state string as returned by the `_Foo/get_` method (representing the state of all objects of this type in the account). If supplied, the string must match the current state, otherwise the method will be aborted and a "stateMismatch" error returned. If "null", any changes will be applied to the current state.
- o `*create*`: "Id[Foo]|null" A map of `_creation id_` (a temporary id set by the client) to Foo objects, or "null" if no objects are to be created.

The Foo object type definition may define default values for properties. Any such property may be omitted by the client.

The client MUST omit any properties that may only be set by the server (for example, the `_id_` property on most object types).

- o `*update*`: "Id[PatchObject]|null" A map of id to a Patch object to apply to the current Foo object with that id, or "null" if no objects are to be updated.

A `_PatchObject_` is of type "String[*]", and represents an unordered set of patches. The keys are a path in [RFC6901] JSON pointer format, with an implicit leading "/" (i.e. prefix each key with "/" before applying the JSON pointer evaluation algorithm).

All paths MUST also conform to the following restrictions; if there is any violation, the update MUST be rejected with an "invalidPatch" error:

- * The pointer MUST NOT reference inside an array (i.e. you MUST NOT insert/delete from an array; the array MUST be replaced in its entirety instead).

- * All parts prior to the last (i.e. the value after the final slash) MUST already exist on the object being patched.
- * There MUST NOT be two patches in the PatchObject where the pointer of one is the prefix of the pointer of the other, e.g. "alerts/1/offset" and "alerts".

The value associated with each pointer determines how to apply that patch:

- * If "null", set to the default value if specified for this property, otherwise remove the property from the patched object. If the key is not present in the parent, this a no-op.
- * Anything else: The value to set for this property (this may be a replacement or addition to the object being patched).

Any server-set properties MAY be included in the patch if their value is identical to the current server value (before applying the patches to the object). Otherwise, the update MUST be rejected with an `_invalidProperties_ SetError`.

This patch definition is designed such that an entire Foo object is also a valid PatchObject. The client MAY choose to optimise network usage by just sending the diff, or MAY just send the whole object; the server processes it the same either way.

- o `*destroy*`: "Id[]|null" A list of ids for Foo objects to permanently delete, or "null" if no objects are to be destroyed.

Each creation, modification or destruction of an object is considered an atomic unit. It is permissible for the server to commit changes to some objects but not others, however it MUST NOT only commit part of an update to a single record (e.g. update a `_name_` property but not a `_count_` property, if both are supplied in the update object).

The final state MUST be valid after the Foo/set is finished, however the server may have to transition through invalid intermediate states (not exposed to the client) while processing the individual create/update/destroy requests. For example, suppose there is a "name" property that must be unique. A single method call could rename an object A => B, and simultaneously rename another object B => A. If the final state is valid, this is allowed. Otherwise, each creation, modification or destruction of an object should be processed sequentially and accepted/rejected based on the current server state.

If a create, update or destroy is rejected, the appropriate error MUST be added to the notCreated/notUpdated/notDestroyed property of the response and the server MUST continue to the next create/update/destroy. It does not terminate the method.

If an id given cannot be found, the update or destroy MUST be rejected with a "notFound" set error.

The server MAY skip an update (rejecting it with a "willDestroy" SetError) if that object is destroyed in the same /set request.

Some records may hold references to other records (foreign keys). That reference may be set (via create or update) in the same request as the referenced record is created. To do this, the client refers to the new record using its creation id prefixed with a "#". The order of the method calls in the request by the client MUST be such that the record being referenced is created in the same or an earlier call. The server thus never has to look ahead. Instead, while processing a request the server MUST keep a simple map for the duration of the request of creation id to record id for each newly created record, so it can substitute in the correct value if necessary in later method calls. In the case of records with references to the same type, the server MUST order the creates and updates within a single method call so that creates happen before their creation ids are referenced by another create/update/destroy in the same call.

Creation ids are not scoped by type but are a single map for all types. A client SHOULD NOT reuse a creation id anywhere in the same API request. If a creation id is reused, the server MUST map the creation id to the most recently created item with that id. To allow easy proxying of API requests, an initial set of creation id to real id values may be passed with a request (see The Request object in section 3.2) and the final state of the map passed out with the response (see section 3.3).

The response has the following arguments:

- o *accountId*: "Id" The id of the account used for the call.
- o *oldState*: "String|null" The state string that would have been returned by _Foo/get_ before making the requested changes, or "null" if the server doesn't know what the previous state string was.
- o *newState*: "String" The state string that will now be returned by _Foo/get_.

- o `*created*`: `"Id[Foo]|null"` A map of the creation id to an object containing any properties of the created Foo object that were not sent by the client. This includes all server-set properties (such as the `_id_` in most object types) and any properties that were omitted by the client and so set to a default by the server.

This argument is "null" if no Foo objects were successfully created.

- o `*updated*`: `"Id[Foo|null]|null"` The `_keys_` in this map are the ids of all Fools that were successfully updated.

The `_value_` for each id is a Foo object containing any property that changed in a way `_not_` explicitly requested by the `_PatchObject_` sent to the server, or "null" if none. This lets the client know of any changes to server-set or computed properties.

This argument is "null" if no Foo objects were successfully updated.

- o `*destroyed*`: `"Id[]|null"` A list of Foo ids for records that were successfully destroyed, or "null" if none.
- o `*notCreated*`: `"Id[SetError]|null"` A map of creation id to a SetError object for each record that failed to be created, or "null" if all successful.
- o `*notUpdated*`: `"Id[SetError]|null"` A map of Foo id to a SetError object for each record that failed to be updated, or "null" if all successful.
- o `*notDestroyed*`: `"Id[SetError]|null"` A map of Foo id to a SetError object for each record that failed to be destroyed, or "null" if all successful.

A `*SetError*` object has the following properties:

- o `*type*`: "String" The type of error.
- o `*description*`: `"String|null"` A description of the error to help debug with an explanation of what the problem was. This is a non-localised string, and is not intended to be shown directly to end users.

The following SetError types are defined and may be returned for set operations on any record type where appropriate:

- o "forbidden": (create; update; destroy) The create/update/destroy would violate an ACL or other permissions policy.
- o "overQuota": (create; update) The create would exceed a server-defined limit on the number or total size of objects of this type.
- o "tooLarge": (create; update) The create/update would result in an object that exceeds a server-defined limit for the maximum size of a single object of this type.
- o "rateLimit": (create) Too many objects of this type have been created recently, and a server-defined rate limit has been reached. It may work if tried again later.
- o "notFound": (update; destroy) The id given to update/destroy cannot be found.
- o "invalidPatch": (update) The PatchObject given to update the record was not a valid patch (see the patch description).
- o "willDestroy" (update) The client requested an object be both updated and destroyed in the same /set request, and the server has decided to therefore ignore the update.
- o "invalidProperties": (create; update) The record given is invalid in some way. For example:
 - * It contains properties which are invalid according to the type specification of this record type.
 - * It contains a property that may only be set by the server (e.g. "id") and is different to the current value. Note, to allow clients to pass whole objects back, it is not an error to include a server-set property in an update so long as the value is identical to the current value on the server.
 - * There is a reference to another record (foreign key) and the given id does not correspond to a valid record.

The SetError object SHOULD also have a property called `_properties_` of type "String[]" that lists *all* the properties that were invalid.

Individual methods MAY specify more specific errors for certain conditions that would otherwise result in an invalidProperties error. If the condition of one of these is met, it MUST be returned instead of the invalidProperties error.

- o "singleton": (create; destroy) This is a singleton type, so you cannot create another one or destroy the existing one.

Other possible SetError types MAY be given in specific method descriptions. Other properties MAY also be present on the `_SetError_` object, as described in the relevant methods.

The following additional errors may be returned instead of the `_Foo/set_` response:

"requestTooLarge": The total number of objects to create, update or destroy exceeds the maximum number the server is willing to process in a single method call.

"stateMismatch": An "ifInState" argument was supplied and it does not match the current state.

5.4. /copy

The only way to move Foo records *between* two different accounts is to copy them using the `_Foo/copy_` method, then once the copy has succeeded, delete the original. The `_onSuccessDestroyOriginal_` argument allows you to try to do this in one method call, however note that the two different actions are not atomic, and so it is possible for the copy to succeed but the original not to be destroyed for some reason.

The copy is conceptually in three phases:

1. Reading the current values from the "from" account.
2. Writing the new copies to the other account.
3. Destroying the originals in the "from" account, if requested.

Data may change in between phases due to concurrent requests.

The `_Foo/copy_` method takes the following arguments:

- o `*fromAccountId*`: "Id" The id of the account to copy records from.
- o `*ifFromInState*`: "String|null" This is a state string as returned by the `_Foo/get_` method. If supplied, the string must match the current state of the account referenced by the `fromAccountId` when reading the data to be copied, otherwise the method will be aborted and a "stateMismatch" error returned. If "null", the data will be read from the current state.

- o `*accountId*`: "Id" The id of the account to copy records to. This MUST be different to the `"fromAccountId"`.
- o `*ifInState*`: "String|null" This is a state string as returned by the `_Foo/get_` method. If supplied, the string must match the current state of the account referenced by the `accountId`, otherwise the method will be aborted and a `"stateMismatch"` error returned. If `"null"`, any changes will be applied to the current state.
- o `*create*`: "Id[Foo]" A map of `_creation id_` to a Foo object. The object MUST contain an id property: the id (in the `fromAccount`) of the record to be copied. Any other properties included are used instead of the current value for that property on the original when creating the copy.
- o `*onSuccessDestroyOriginal*`: "Boolean" (default: false) If `"true"`, an attempt will be made to destroy the original records that were successfully copied: after emitting the `_Foo/copy_` response, but before processing the next method, the server MUST make a single call to `_Foo/set_` to destroy the original of each successfully copied record; the output of this is added to the responses as normal to be returned to the client.
- o `*destroyFromIfInState*`: "String|null" This argument is passed on as the `"ifInState"` argument to the implicit `_Foo/set_` call, if made at the end of this request to destroy the originals that were successfully copied.

Each record copy is considered an atomic unit which may succeed or fail individually.

The response has the following arguments:

- o `*fromAccountId*`: "Id" The id of the account records were copied from.
- o `*accountId*`: "Id" The id of the account records were copied to.
- o `*oldState*`: "String|null" The state string that would have been returned by `_Foo/get_` on the account records were copied to before making the requested changes, or `"null"` if the server doesn't know what the previous state string was.
- o `*newState*`: "String" The state string that will now be returned by `_Foo/get_` on the account records were copied to.

- o `*created*`: `"Id[Foo]|null"` A map of the creation id to an object containing any properties of the copied Foo object that are set by the server (such as the `_id_` in most object types; note, the id is likely to be different to the id of the object in the account it was copied from).

This argument is "null" if no Foo objects were successfully copied.

- o `*notCreated*`: `"Id[SetError]|null"` A map of creation id to a SetError object for each record that failed to be copied, "null" if none.

The `*SetError*` may be any of the standard set errors that may be returned for a `_create_` or `_update_`. In addition, the following SetError is defined:

`"alreadyExists"`: The server forbids duplicates and the record already exists in the target account. An `_existingId_` property of type "Id" MUST be included on the error object with the id of the existing record.

The following additional errors may be returned instead of the `_Foo/copy_` response:

`"fromAccountNotFound"`: The `_fromAccountId_` does not correspond to a valid account.

`"fromAccountNotSupportedByMethod"`: The `_fromAccountId_` given corresponds to a valid account, but the account does not support this data type.

`"stateMismatch"`: An `"ifInState"` argument was supplied and it does not match the current state, or an `"ifFromInState"` argument was supplied and it does not match the current state in the from account.

5.5. `/query`

For data sets where the total amount of data is expected to be very small, clients can just fetch the complete set of data and then do any sorting/filtering locally. However, for large data sets (e.g. multi-gigabyte mailboxes), the client needs to be able to search/sort/window the data type on the server.

A query on the set of Foos in an account is made by calling `_Foo/query_`. This takes a number of arguments to determine which records to include, how they should be sorted, and which part of the result

should be returned (the full list may be `_very_` long). The result is returned as a list of Foo ids.

A call to `_Foo/query_` takes the following arguments:

- o `*accountId*`: "Id" The id of the account to use.
- o `*filter*`: "FilterOperator|FilterCondition|null" Determines the set of Foos returned in the results. If "null", all objects in the account of this type are included in the results. A `*FilterOperator*` object has the following properties:
 - * `*operator*`: "String" This MUST be one of the following strings: "AND" / "OR" / "NOT":
 - + `*AND*`: all of the conditions must match for the filter to match.
 - + `*OR*`: at least one of the conditions must match for the filter to match.
 - + `*NOT*`: none of the conditions must match for the filter to match.
 - * `*conditions*`: "(FilterOperator|FilterCondition)[]" The conditions to evaluate against each record.

A `*FilterCondition*` is an "object" whose allowed properties and semantics depend on the data type and is defined in the `_/_query_` method specification for that type. It MUST NOT have an `_operator_` property.

- o `*sort*`: "Comparator[]|null" Lists the names of properties to compare between two Foo records, and how to compare them, to determine which comes first in the sort. If two Foo records have an identical value for the first comparator, the next comparator will be considered and so on. If all comparators are the same (this includes the case where an empty array or "null" is given as the `_sort_` argument), the sort order is server-dependent, but MUST be stable between calls to `Foo/query`. A `*Comparator*` has the following properties:
 - * `*property*`: "String" The name of the property on the Foo objects to compare.
 - * `*isAscending*`: "Boolean" (optional; default: true) If "true", sort in ascending order. If "false", reverse the comparator's results to sort in descending order.

- * `*collation*`: "String" (optional; default is server-dependent)
The identifier, as registered in the collation registry defined in [RFC4790], for the algorithm to use when comparing the order of strings. The algorithms the server supports are advertised in the capabilities object returned with the JMAP Session object.

If omitted, the default algorithm is server-dependent, but:

1. It MUST be unicode-aware.
2. It MAY be selected based on an Accept-Language header in the request (as defined in [RFC7231] section 5.3.5), or out-of-band information about the user's language/locale.
3. It SHOULD be case-insensitive where such a concept makes sense for a language/locale. Where the user's language is unknown, it is RECOMMENDED to follow the advice in section 5.2.3 of [RFC8264].

The "i;unicode-casemap" collation ([RFC5051]) and the Unicode Collation Algorithm (<<http://www.unicode.org/reports/tr10/>>) are two examples that fulfil these criterion and provide reasonable behaviour for a large number of languages.

When the property being compared is not a string, the `_collation_` property is ignored and the following comparison rules apply based on the type. In ascending order:

- + "Boolean": "false" comes before "true".
- + "Number": A lower number comes before a higher number.
- + "Date"/"UTCDate": The earlier date comes first.

The Comparator object may also have additional properties as required for specific sort operations defined in a type's /query method.

- o `*position*`: "Int" (default: 0) The 0-based index of the first id in the full list of results to return.

If a negative value is given, it is an offset from the end of the list. Specifically, the negative value MUST be added to the total number of results given the filter, and if still negative clamped to "0". This is now the 0-based index of the first id to return.

If the index is greater than or equal to the total number of objects in the results list then the `_ids_` array in the response will be empty, but this is not an error.

- o `*anchor*`: "Id|null" A Foo id. If supplied the `_position_` argument is ignored. The index of this id in the results will be used in combination with the "anchorOffset" argument to determine the index of the first result to return (see below for more details).
- o `*anchorOffset*`: "Int" (default: 0) The index of the first result to return relative to the index of the anchor, if an anchor is given. This MAY be negative. For example, "-1" means the Foo immediately preceding the anchor is the first result in the list returned (see below for more details).
- o `*limit*`: "UnsignedInt|null" The maximum number of results to return. If "null", no limit presumed. The server MAY choose to enforce a maximum "limit" argument. In this case, if a greater value is given (or if it is "null"), the limit is clamped to the maximum; the new limit is returned with the response so the client is aware. If a negative value is given, the call MUST be rejected with an "invalidArguments" error.
- o `*calculateTotal*`: "Boolean" (default: false) Does the client wish to know the total number of results in the query? This may be slow and expensive for servers to calculate, particularly with complex filters, so clients should take care to only request the total when needed.

If an `*anchor*` argument is given, then after filtering and sorting the anchor is looked for in the results. If found, the `*anchor offset*` is then added to its index. If the resulting index is now negative, it is clamped to 0. This index is now used exactly as though it were supplied as the "position" argument. If the anchor is not found, the call is rejected with an "anchorNotFound" error.

If an `_anchor_` is specified, any position argument supplied by the client MUST be ignored. If no `_anchor_` is supplied, any anchor offset argument MUST be ignored.

A client can use `_anchor_` instead of `_position_` to find the index of an id within a large set of results.

The response has the following arguments:

- o `*accountId*`: "Id" The id of the account used for the call.

- o ***queryState***: "String" A string encoding the current state of the query on the server. This string **MUST** change if the results of the query (i.e. the matching ids and their sort order) have changed. The queryState string **MAY** change if something has changed on the server which means the results may have changed but the server doesn't know for sure.

The queryState string only represents the ordered list of ids that match the particular query (including its sort/filter). There is no requirement for it to change if a property on an object matching the query changes but the query results are unaffected (indeed, it is more efficient if the queryState string does not change in this case). The queryState string only has meaning when compared to future responses to a query with the same type/sort/filter, or when used with `/queryChanges` to fetch changes.

Should a client receive back a response with a different queryState string to a previous call it **MUST** either throw away the currently cached query and fetch it again (note, this does not require fetching the records again, just the list of ids) or call `_Foo/queryChanges_` to get the difference.

- o ***canCalculateChanges***: "Boolean" This is "true" if the server supports calling `_Foo/queryChanges_` with these "filter"/"sort" parameters. Note, this does not guarantee that the `_Foo/queryChanges_` call will succeed, as it may only be possible for a limited time afterwards due to server internal implementation details.
- o ***position***: "UnsignedInt" The 0-based index of the first result in the "ids" array within the complete list of query results.
- o ***ids***: "Id[]" The list of ids for each foo in the query results, starting at the index given by the `_position_` argument of this response, and continuing until it hits the end of the results or reaches the "limit" number of ids. If `_position_` is \geq `_total_`, this **MUST** be the empty list.
- o ***total***: "UnsignedInt" (only if requested) The total number of foos in the results (given the `_filter_`). This argument **MUST** be omitted if the `_calculateTotal_` request argument is not "true".
- o ***limit***: "UnsignedInt" (if set by the server) The limit enforced by the server on the maximum number of results to return. This is only returned if the server set a limit, or used a different limit to that given in the request.

The following additional errors may be returned instead of the `_Foo/query_` response:

"anchorNotFound": An anchor argument was supplied, but it cannot be found in the results of the query.

"unsupportedSort": The `_sort_` is syntactically valid, but includes a property the server does not support sorting on, or a collation method it does not recognise.

"unsupportedFilter": The `_filter_` is syntactically valid, but the server cannot process it. If the filter was the result of a user's search input, the client SHOULD suggest the user simplify their search.

5.6. /queryChanges

The "Foo/queryChanges" method allows a client to efficiently update the state of a cached query to match the new state on the server. It takes the following arguments:

- o `*accountId*`: "Id" The id of the account to use.
- o `*filter*`: "FilterOperator|FilterCondition|null" The filter argument that was used with `_Foo/query_`.
- o `*sort*`: "Comparator[]|null" The sort argument that was used with `_Foo/query_`.
- o `*sinceQueryState*`: "String" The current state of the query in the client. This is the string that was returned as the `_queryState_` argument in the `_Foo/query_` response with the same sort/filter. The server will return the changes made to the query since this state.
- o `*maxChanges*`: "UnsignedInt|null" The maximum number of changes to return in the response. See error descriptions below for more details.
- o `*upToId*`: "Id|null" The last (highest-index) id the client currently has cached from the query results. When there are a large number of results, in a common case the client may have only downloaded and cached a small subset from the beginning of the results. If the sort and filter are both only on immutable properties, this allows the server to omit changes after this point in the results, which can significantly increase efficiency. If they are not immutable, this argument is ignored.

- o `*calculateTotal*`: "Boolean" (default: false) Does the client wish to know the total number of results now in the query? This may be slow and expensive for servers to calculate, particularly with complex filters, so clients should take care to only request the total when needed.

The response has the following arguments:

- o `*accountId*`: "Id" The id of the account used for the call.
- o `*oldQueryState*`: "String" This is the "sinceQueryState" argument echoed back; the state from which the server is returning changes.
- o `*newQueryState*`: "String" This is the state the query will be in after applying the set of changes to the old state.
- o `*total*`: "UnsignedInt" (only if requested) The total number of foos in the results (given the `_filter_`). This argument **MUST** be omitted if the `_calculateTotal_` request argument is not "true".
- o `*removed*`: "Id[]" The `_id_` for every foo that was in the query results in the old state and is not in the results in the new state.

If the server cannot calculate this exactly, the server **MAY** return extra foos in addition that may have been in the old results but are not in the new results.

If the sort and filter are both only on immutable properties and an `_upToId_` is supplied and exists in the results, any ids that were removed but have a higher index than `_upToId_` **SHOULD** be omitted.

If the `_filter_` or `_sort_` includes a mutable property, the server **MUST** include all foos in the current results for which this property may have changed. The position of these may have moved in the results so must be reinserted by the client to ensure its query cache is correct.

- o `*added*`: "AddedItem[]" The id and index in the query results (in the new state) for every foo that has been added to the results since the old state **AND** every foo in the current results that was included in the `_removed_` array (due to a filter or sort based upon a mutable property).

If the sort and filter are both only on immutable properties and an `_upToId_` is supplied and exists in the results, any ids that

were added but have a higher index than `_upToId_` SHOULD be omitted.

The array MUST be sorted in order of index, lowest index first.

An `*AddedItem*` object has the following properties:

```
* *id*: "Id"

* *index*: "UnsignedInt"
```

The result of this is that if the client has a cached sparse array of foo ids corresponding to the results in the old state:

```
fooIds = [ "id1", "id2", null, null, "id3", "id4", null, null, null ]
```

then if it `*splices out*` all ids in the removed array that it has in its cached results:

```
removed = [ "id2", "id3", ... ];
fooIds => [ "id1", null, null, "id3", "id4", null, null, null ]
```

and `*splices in*` (one-by-one in order, starting with the lowest index) all of the ids in the added array:

```
added = [{ id: "id5", index: 0, ... }];
fooIds => [ "id5", "id1", null, null, "id3", "id4", null, null, null ]
```

and `*truncates*` or `*extends*` to the new total length, then the results will now be in the new state.

Note: splicing in adds the item at the given index, incrementing the index of all items previously at that or a higher index. Splicing out is the inverse, removing the item and decrementing the index of every item after it in the array.

The following additional errors may be returned instead of the `_Foo/queryChanges_` response:

`"tooManyChanges"`: There are more changes than the client's `_maxChanges_` argument. Each item in the removed or added array is considered as one change. The client may retry with a higher max changes or invalidate its cache of the query results.

`"cannotCalculateChanges"`: The server cannot calculate the changes from the `queryState` string given by the client. Usually due to the client's state being too old. The client MUST invalidate its cache of the query results.

5.7. Examples

Suppose we have a type `_Todo_` with the following properties:

- o `*id*`: "Id" (immutable; server-set) The id of the object.
- o `*title*`: "String" A brief summary of what is to be done.
- o `*keywords*`: "String[Boolean]" (default: {}) A set of keywords that apply to the todo. The set is represented as an object, with the keys being the `_keywords_`. The value for each key in the object MUST be "true". (This format allows you to update an individual key using patch syntax rather than having to update the whole set of keywords as one, which an "String[]" representation would require.)
- o `*neuralNetworkTimeEstimation*`: "Number" (server-set) The title and keywords are fed into the server's state-of-the-art neural network to get an estimation of how long this todo will take, in seconds.
- o `*subTodoIds*`: "Id[]|null" The ids of a list of subtodos to complete as part of this todo.

Suppose also that all the standard methods are defined for this type, and the `FilterCondition` object supports a "hasKeyword" property to match todos with the given keyword.

A client might want to display the list of todos with either a "music" keyword or a "video" keyword, so it makes the following method call:

```
[ [ "Todo/query", {
  "accountId": "x",
  "filter": {
    "operator": "OR",
    "conditions": [
      { "hasKeyword": "music" },
      { "hasKeyword": "video" }
    ]
  },
  "sort": [{ "property": "title" }],
  "position": 0,
  "limit": 10
}, "0" ],
[ "Todo/get", {
  "accountId": "x",
  "#ids": {
    "resultOf": "0",
    "name": "Todo/query",
    "path": "/ids"
  }
}, "1" ] ]
```

This would query the server for the set of todos with a keyword of either "music" or "video", sorted by title, and limited to the first 10 results. It fetches the full object for each of these Todos using back-references to reference the result of the query. The response might look something like:

```

[[ "Todo/query", {
  "accountId": "x",
  "queryState": "y13213",
  "canCalculateChanges": true,
  "position": 0,
  "ids": [ "a", "b", "c", "d", "e", "f", "g", "h", "i", "j" ]
}, "0" ],
[ "Todo/get", {
  "accountId": "x",
  "state": "10324",
  "list": [{
    "id": "a",
    "title": "Practise Piano",
    "keywords": {
      "music": true,
      "beethoven": true,
      "mozart": true,
      "liszt": true,
      "rachmaninov": true
    },
    "neuralNetworkTimeEstimation": 3600
  }, {
    "id": "b",
    "title": "Watch Daft Punk music video",
    "keywords": {
      "music": true,
      "video": true,
      "trance": true
    },
    "neuralNetworkTimeEstimation": 18000
  },
  ...
]
], "1" ]]

```

Now suppose the user adds a keyword "chopin" and removes the keyword "mozart" from the "Practise Piano" task. The client may send the whole object to the server, as this is a valid PatchObject:


```

[[ "Todo/set", {
  "accountId": "x",
  "ifInState": "10324",
  "update": {
    "a": {
      "id": "a",
      "title": "Practise Piano",
      "keywords": {
        "music": true,
        "beethoven": true,
        "chopin": true,
        "liszt": true,
        "rachmaninov": true
      },
    },
    "neuralNetworkTimeEstimation": 360
  }
}, "0" ]]

```

or it may send a minimal patch:

```

[[ "Todo/set", {
  "accountId": "x",
  "ifInState": "10324",
  "update": {
    "a": {
      "keywords/chopin": true,
      "keywords/mozart": null
    }
  }
}, "0" ]]

```

The effect is exactly the same on the server in either case, and presuming the server is still in state "10324" it will probably return success:

```

[[ "Todo/set", {
  "accountId": "x",
  "oldState": "10324",
  "newState": "10329",
  "updated": {
    "a": {
      "neuralNetworkTimeEstimation": 5400
    }
  }
}, "0" ]]

```

The server changed the "neuralNetworkTimeEstimation" property on the object as part of this change; as this changed in a way `_not_` explicitly requested by the PatchObject sent to the server, it is returned with the "updated" confirmation.

Let us now add a subtodo to our new "Practice Piano" todo. In this example we can see the use of a reference to a creation id to allow us to set a foreign key reference to a record created in the same request:

```
[[ "Todo/set", {
  "accountId": "x",
  "create": {
    "k15": {
      "title": "Warm up with scales"
    }
  },
  "update": {
    "a": {
      "subTodoIds": [ "#k15" ]
    }
  }
}, "0" ]]
```

Now, suppose another user deleted the "Listen to Daft Punk" todo. The first user will receive a push notification (see section 7) with the changed state string for the "Todo" type. Since the new string does not match its current state, it knows it needs to check for updates. It may make a request like:

```
[[ "Todo/changes", {
  "accountId": "x",
  "sinceState": "10324",
  "maxChanges": 50
}, "0" ],
[ "Todo/queryChanges", {
  "accountId": "x",
  "filter": {
    "operator": "OR",
    "conditions": [
      { "hasKeyword": "music" },
      { "hasKeyword": "video" }
    ]
  },
  "sort": [{ "property": "title" }],
  "sinceQueryState": "y13213",
  "maxChanges": 50
}, "1" ]]
```

and receive in response:

```
[ [ "Todo/changes", {
  "accountId": "x",
  "oldState": "10324",
  "newState": "871903",
  "hasMoreChanges": false,
  "created": [],
  "updated": [],
  "destroyed": ["b"]
}, "0" ],
[ "Todo/queryChanges", {
  "accountId": "x",
  "oldQueryState": "y13213",
  "newQueryState": "y13218",
  "removed": ["b"],
  "added": null
}, "1" ] ]
```

Suppose the user has access to another account "y", for example a team account shared between multiple users. To move an existing Todo from account "x", the client would call:

```
[ [ "Todo/copy", {
  "fromAccountId": "x",
  "accountId": "y",
  "create": {
    "k5122": {
      "id": "a"
    }
  }
},
  "onSuccessDestroyOriginal": true
}, "0" ] ]
```

The server successfully copies the Todo to a new account (where it receives a new id) and deletes the original. Due to the implicit call to "Todo/set", there are two responses to the single method call, both with the same method call id:

```

[[ "Todo/copy", {
  "fromAccountId": "x",
  "accountId": "y",
  "created": {
    "k5122": {
      "id": "DAf97"
    }
  },
  "oldState": "c1d64ecb038c",
  "newState": "33844835152b"
}, "0" ],
[ "Todo/set", {
  "accountId": "x",
  "oldState": "871903",
  "newState": "871909",
  "destroyed": [ "a" ],
  ...
}, "0" ]]

```

5.8. Proxy considerations

JMAP has been designed to allow an API endpoint to easily proxy through to one or more JMAP servers. This may be useful for load balancing, augmenting capabilities, or presenting a single endpoint to accounts hosted on different JMAP servers (splitting the request based on each method's "accountId" argument). The proxy need only understand the general structure of a JMAP Request object, it does not need to know anything specifically about the methods and arguments it will pass through to other servers.

If splitting up the methods in a request to call them on different backend servers, the proxy must do two things to ensure back-references and creation id references resolve the same as if the entire request were processed on a single server:

1. It must pass a "createdIds" property with each subrequest. If this is not given by the client, an empty object should be used for the first subrequest. The "createdIds" property of each subresponse should be passed on in the next subrequest.
2. It must resolve back-references to previous method results that were processed on a different server. This is a relatively simple syntactic substitution, described in section 3.6.

When splitting a request based on accountId, proxy implementors do need to be aware of "/copy" methods, that copy between accounts. If the accounts are on different servers, the proxy will have to implement this functionality directly.

6. Binary data

Binary data is referenced by a `_blobId` in JMAP, and uploaded/downloaded separately to the core API. The `blobId` solely represents the raw bytes of data, not any associated metadata such as a file name or content type. Such metadata is stored alongside the `blobId` in the object referencing it. The data represented by a `blobId` is immutable.

Any `blobId` that exists within an account may be used when creating/updating another object in that account. For example, an Email type may have a `blobId` that represents the [RFC5322] representation of the message. A client could create a new Email object with an attachment and use this `blobId`, in effect attaching the old message to the new one. Similarly it could attach any existing attachment of an old message without having to download and upload it again.

When the client uses a `blobId` in a create/update, the server MAY assign a new `blobId` to refer to the same binary data within the new/updated object. If it does so, it MUST return any properties that contain a changed `blobId` in the created/updated response so the client gets the new ids.

A blob that is not referenced by a JMAP object (e.g. as a message attachment) MAY be deleted by the server to free up resources. Uploads (see below) are initially unreferenced blobs. To ensure interoperability:

- o The server SHOULD use a separate quota for unreferenced blobs to the accounts's usual quota. This quota SHOULD be separate per user in the case of shared accounts.
- o This quota SHOULD be at least the maximum total size that a single object can reference on this server. For example, if supporting JMAP Mail, this should be at least the maximum total attachments size for a message.
- o When an upload would take the user over quota, the server MUST delete unreferenced blobs in date order, oldest first, until there is room for the new blob.
- o Except where quota restrictions force early deletion, an unreferenced blob MUST NOT be deleted for at least 1 hour from the time of upload; if reuploaded, the same `blobId` MAY be returned, but this SHOULD reset the expiry time.

- o A blob MUST NOT be deleted during the method call which removed the last reference, so that a client can issue a create and a destroy that both reference the blob within the same method call.

6.1. Uploading binary data

There is a single endpoint which handles all file uploads for an account, regardless of what they are to be used for. The JMAP Session object has an `_uploadUrl_` property in [RFC6570] URI Template (level 1) format, which MUST contain a variable called "accountId". The client may use this template in combination with an `_accountId_` to get the URL of the file upload resource.

To upload a file, the client submits an authenticated POST request to the file upload resource.

A successful request MUST return a single JSON object with the following properties as the response:

- o `*accountId*`: "Id" The id of the account used for the call.
- o `*blobId*`: "Id", The id representing the binary data uploaded. The data for this id is immutable. The id `_only_` refers to the binary data, not any metadata.
- o `*type*`: "String" The media type of the file (as specified in [RFC6838], section 4.2) as set in the Content-Type header of the upload HTTP request.
- o `*size*`: "UnsignedInt" The size of the file in octets.

If identical binary content to an existing blob in the account is uploaded, the existing blobId MAY be returned.

Clients should use the blobId returned in a timely manner. Under rare circumstances the server may have deleted the blob before the client uses it; the client should keep a reference to the local file so it can upload it again in such a situation.

When an HTTP error response is returned to the client, the server SHOULD return a JSON "problem details" object as the response body, as per [RFC7807].

As access controls are often determined by the object holding the reference to a blob, unreferenced blobs MUST only be accessible to the uploader, even in shared accounts.

6.2. Downloading binary data

The JMAP Session object has a `_downloadUrl_` property, which is in [RFC6570] URI Template (level 1) format. The URL MUST contain variables called "accountId", "blobId", "type" and "name".

To download a file, the client makes an authenticated GET request to the download URL with the appropriate variables substituted in:

- o "accountId": The id of the account to which the record with the blobId belongs.
- o "blobId": The blobId representing the data of the file to download.
- o "type": The type for the server to set in the "Content-Type" header of the response; the blobId only represents the binary data and does not have a content-type innately associated with it.
- o "name": The name for the file; the server MUST return this as the filename if it sets a "Content-Disposition" header.

As the data for a particular blobId is immutable, and thus the response in the generated download URL is too, implementors are recommended to set long cache times and use the "immutable" Cache-Control extension ([RFC8246]) for a successful responses, for example "Cache-Control: private, immutable, max-age=31536000".

When an HTTP error response is returned to the client, the server SHOULD return a JSON "problem details" object as the response body, as per [RFC7807].

6.3. Blob/copy

Binary data may be copied *between* two different accounts using the `_Blob/copy_` method, rather than having to download then re-upload on the client.

The `_Blob/copy_` method takes the following arguments:

- o `*fromAccountId*`: "Id" The id of the account to copy blobs from.
- o `*accountId*`: "Id" The id of the account to copy blobs to.
- o `*blobIds*`: "Id[]" A list of ids of blobs to copy to the other account.

The response has the following arguments:

- o `*fromAccountId*`: "Id" The id of the account blobs were copied from.
- o `*accountId*`: "Id" The id of the account blobs were copied to.
- o `*copied*`: "Id[Id]|null" A map of the blobId in the `_fromAccount_` to the id for the blob in the account it was copied to, or "null" if none were successfully copied.
- o `*notCopied*`: "Id[SetError]|null" A map of blobId to a SetError object for each blob that failed to be copied, "null" if none.

The `*SetError*` may be any of the standard set errors that may be returned for a `_create_`, as defined in section 5.3. In addition, the "notFound" SetError error may be returned if the blobId to be copied cannot be found.

The following additional method-level error may be returned instead of the `_Blob/copy_` response:

"fromAccountNotFound": The `_fromAccountId_` included with the request does not correspond to a valid account.

7. Push

Push notifications allow clients to efficiently update (almost) instantly to stay in sync with data changes on the server. The general model for push is simple and sends minimal data over the push channel: just enough for the client to know whether it needs to resync. The format allows multiple changes to be coalesced into a single push update, and the frequency of pushes to be rate limited by the server. It doesn't matter if some push events are dropped before they reach the client; the next time it gets/sets any records of a changed type it will discover the data has changed and still sync all changes.

There are two different mechanisms by which a client can receive push notifications, to allow for the different environments in which a client may exist. An event source resource (see section 7.3) allows clients that can hold transport connections open to receive push notifications directly from the JMAP server. This is simple and avoids 3rd parties, but is often not feasible on constrained platforms such as mobile devices. Alternatively, clients can make use of any push service supported by their environment. A URL for the push service is registered with the JMAP server (see section 7.2), then the server then POSTs each notification to that URL. The push service is then responsible for routing these to the client.

7.1. The StateChange object

When something changes on the server, the server pushes a **StateChange** object to the client. A **StateChange** object has the following properties:

- o **@type**: "String" This MUST be the string "StateChange".
- o **changed**: "Id[TypeState]" A map of *_account id_* to an object encoding the state of data types that have changed for that account since the last StateChange object was pushed, for each of the accounts to which the user has access and for which something has changed.

A **TypeState** object is a map. The keys are the type name "Foo" (e.g. "Mailbox" or "Email"), and the value is the *_state_* property that would currently be returned by a call to *_Foo/get_*.

The client can compare the new state strings with its current values to see whether it has the current data for these types. If not, the changes can then be efficiently fetched in a single standard API request (using the *_/changes_* type methods).

7.1.1. Example

In this example, the server has amalgamated a few changes together across two different accounts the user has access to, before pushing the following StateChange object to the client:

```
{
  "@type": "StateChange",
  "changed": {
    "a3123": {
      "Email": "d35ecb040aab",
      "EmailDelivery": "428d565f2440",
      "CalendarEvent": "87accfac587a"
    },
    "a43461d": {
      "Mailbox": "0af7a512ce70",
      "CalendarEvent": "7a4297cecd76"
    }
  }
}
```

The client can compare the state strings with its current state for the Email, CalendarEvent etc. object types in the appropriate accounts to see if it needs to fetch changes.

If the client is itself making changes, it may receive a `StateChange` object while the `/set` API call is in flight. It can wait until the call completes and then compare if the new state string after the `/set` is the same as was pushed in the `StateChange` object; if so, and the old state of the `/set` response matches the client's previous state, it does not need to waste a request asking for changes it already knows.

7.2. PushSubscription

Clients may create a `_PushSubscription_` to register a URL with the JMAP server. The JMAP server will then make an HTTP POST request to this URL for each push notification it wishes to send to the client.

As a push subscription causes the JMAP server to make a number of requests to a previously unknown endpoint, it can be used as a vector for launching a denial of service attack. To prevent this, when a subscription is created the JMAP server immediately sends a `PushVerification` object to that URL (see section 7.2.2). The JMAP server **MUST NOT** make any further requests to the URL until the client receives the push and updates the subscription with the correct verification code.

A `*PushSubscription*` object has the following properties:

- o `*id*`: "Id" (immutable; server-set) The id of the push subscription.
- o `*deviceClientId*`: "String" (immutable) An id that uniquely identifies the client + device it is running on. The purpose of this is to allow clients to identify which `PushSubscription` objects they created even if they lose their local state, so they can revoke or update them. This string **MUST** be different on different devices, and be different from apps from other vendors. It **SHOULD** be easy to re-generate, not depend on persisted state. It is **RECOMMENDED** to use a secure hash of a string that contains:
 1. A unique identifier associated with the device where the JMAP client is running, normally supplied by the device's operating system.
 2. A custom vendor/app id, including a domain controlled by the vendor of the JMAP client.

To protect the privacy of the user, the `deviceClientId` id **MUST NOT** contain an unobfuscated device id.

- o `*url*`: "String" (immutable) An absolute URL where the JMAP server will POST the data for the push message. This MUST begin with "https://".
- o `*keys*`: "Object|null" (immutable) Client-generated encryption keys. If supplied the server MUST use them as specified in [RFC8291] to encrypt all data sent to the push subscription. The object MUST have the following properties:
 - * `*p256dh*`: the P-256 ECDH Diffie-Hellman public key as described in [RFC8291], encoded in URL-safe Base64 representation as defined in [RFC4648].
 - * `*auth*`: the authentication secret as described in [RFC8291], encoded in URL-safe Base64 representation as defined in [RFC4648].
- o `*verificationCode*`: "String|null" This MUST be "null" (or omitted) when the subscription is created. The JMAP server then generates a verification code and sends it in a push message, and the client updates the PushSubscription object with the code; see section 7.2.2 for details.
- o `*expires*`: "UTCDate|null" The time this push subscription expires. If specified, the JMAP server MUST NOT make further requests to this resource after this time. It MAY automatically destroy the push subscription at or after this time.

The server MAY choose to set an expiry if none is given by the client, or modify the expiry time given by the client to a shorter duration.

- o `*types*`: "String[]|null" A list of types the client is interested in (using the same names as the keys in the `_TypeState_` object defined in the previous section). A `StateChange` notification will only be sent if the data for one of these types changes. Other types are omitted from the `TypeState` object. If "null", changes will be pushed for all types.

The POST request MUST have a content type of "application/json" and contain the UTF-8 JSON encoded object as the body. The request MUST have a "TTL" header, and MAY have "Urgency" and/or "Topic" headers, as specified in section 5 of [RFC8030]. The JMAP server is expected to understand and handle HTTP status responses in a reasonable manner. A "429" (Too Many Requests) response MUST cause the JMAP server to reduce the frequency of pushes; the JMAP push structure allows multiple changes to be coalesced into a single minimal

StateChange object. See the security considerations in section 8.6 for a discussion of the risks in connecting to unknown servers.

The JMAP server acts as an Application Server as defined in [RFC8030]. A client MAY use the rest of [RFC8030] in combination with its own Push Service to form a complete end-to-end solution, or MAY rely on alternative mechanisms to ensure the delivery of the pushed data after it leaves the JMAP server.

The push subscription is tied to the credentials used to authenticate the API request that created it. Should these credentials expire or be revoked, the push subscription MUST be destroyed by the JMAP server. Only subscriptions created by these credentials are returned when the client fetches existing subscriptions.

When these credentials have their own expiry (i.e. it is a session with a timeout), the server SHOULD NOT set or bound the expiry time for the push subscription given by the client, but MUST expire it when the session expires.

When these credentials are not time bounded (e.g. [RFC7617] Basic Authentication), the server SHOULD set an expiry time for the push subscription if none given, and limit the expiry time if set too far in the future. This maximum expiry time MUST be at least 48 hours in the future and SHOULD be at least 7 days in the future. An app running on a mobile device may only be able to refresh the push subscription lifetime when it is in the foreground, and so this gives a reasonable timeframe to allow this to happen.

In the case of separate access and refresh credentials, as in [RFC6749] OAuth 2.0, the server SHOULD tie the push subscription to the validity of the refresh token rather than the access token, and behave according to whether this is time-limited or not.

When a push subscription is destroyed, the server MUST securely erase the URL and encryption keys from memory and storage as soon as possible.

7.2.1. PushSubscription/get

Standard `_/get_` method as described in section 5.1, except it does **not** take or return an `_accountId_` argument, as push subscriptions are not tied to specific accounts. It also does **not** return a `_state_` argument. The `_ids_` argument may be "null" to fetch all at once.

The server MUST only return push subscriptions that were created using the same authentication credentials as for this PushSubscription/get request.

As the `_url_` and `_keys_` properties may contain data that is private to a particular device, the values for these properties MUST NOT be returned. If the `_properties_` argument is "null" or omitted, the server MUST default to all properties excluding these two. If one of them is explicitly requested, the method call MUST be rejected with a "forbidden" error.

7.2.2. PushSubscription/set

Standard `_set_` method as described in section 5.3, except it does *not* take or return an `_accountId_` argument, as push subscriptions are not tied to specific accounts. It also does *not* take an `_ifInState_` argument or return `_oldState_` or `_newState_` arguments.

The `_url_` and `_keys_` properties are immutable; if the client wishes to change these, it must destroy the current push subscription and create a new one.

When a PushSubscription is created, the server MUST immediately push a *PushVerification* object to the URL. It has the following properties:

- o **@type**: "String" This MUST be the string "PushVerification".
- o **pushSubscriptionId**: "String" The id of the push subscription that was created.
- o **verificationCode**: "String" The verification code to add to the push subscription. This MUST contain sufficient entropy to avoid the client being able to brute force guess the code.

The client MUST update the push subscription with the correct verification code before the server makes any further requests to the subscription's URL. Attempts to update the subscription with an invalid verification code MUST be rejected by the server with an "invalidProperties" SetError.

The client may update the `_expires_` property to extend (or, less commonly, shorten) the lifetime of a push subscription. The server MAY modify the proposed new expiry time to enforce server-defined limits. Extending the lifetime does not require the subscription to be verified again.

Clients SHOULD NOT update or destroy a push subscription that they did not create (i.e. has a `_deviceId` that they do not recognise).

7.2.3. Example

At "2018-07-06T02:14:29Z", a client with `deviceId` "a889-ffea-910" fetches the set of push subscriptions currently on the server, making an API request with:

```
[[ "PushSubscription/get", {
  "ids": null
}, "0" ]]
```

Which returns:

```
[[ "PushSubscription/get", {
  "list": [{
    "id": "e50b2c1d-9553-41a3-b0a7-a7d26b599ee1",
    "deviceId": "b37ff8001ca0",
    "verificationCode": "b210ef734fe5f439c1ca386421359f7b",
    "expires": "2018-07-31T00:13:21Z",
    "types": [ "Todo" ]
  }, {
    "id": "f2d0aab5-e976-4e8b-ad4b-b380a5b987e4",
    "deviceId": "X8980fc",
    "verificationCode": "f3d4618a9ae15c8b7f5582533786d531",
    "expires": "2018-07-12T05:55:00Z",
    "types": [ "Mailbox", "Email", "EmailDelivery" ]
  }],
  "notFound": []
}, "0" ]]
```

Since neither of the returned push subscription objects have the client's `deviceId`, it knows it does not have a current push subscription active on the server. So it creates one, sending this request:

```
[[ "PushSubscription/set", {
  "create": {
    "4f29": {
      "deviceId": "a889-ffea-910",
      "url": "https://example.com/push/?device=X8980fc&client=12c6d086",
      "types": null
    }
  }
}, "0" ]]
```

The server creates the push subscription but limits the expiry time to 7 days in the future, returning this response:

```
[[ "PushSubscription/set", {
  "created": {
    "4f29": {
      "id": "P43dcfa4-1dd4-41ef-9156-2c89b3b19c60",
      "keys": null,
      "expires": "2018-07-13T02:14:29Z"
    }
  }
}, "0" ]]
```

The server also immediately makes a POST request to "https://example.com/push/?device=X8980fc&client=12c6d086" with the data:

```
{
  "@type": "PushVerification",
  "pushSubscriptionId": "P43dcfa4-1dd4-41ef-9156-2c89b3b19c60",
  "verificationCode": "da1f097b11ca17f06424e30bf02bfa67"
}
```

The client receives this and updates the subscription with the verification code (note there is a potential race condition here; the client MUST be able to handle receiving the push while the request creating the subscription is still in progress):

```
[[ "PushSubscription/set", {
  "update": {
    "P43dcfa4-1dd4-41ef-9156-2c89b3b19c60": {
      "verificationCode": "da1f097b11ca17f06424e30bf02bfa67"
    }
  }
}, "0" ]]
```

The server confirms the update was successful and will now make requests to the registered URL when the state changes.

Two days later, the client updates the subscription to extend its lifetime, sending this request:

```
[["PushSubscription/set", {
  "update": {
    "P43dcfa4-1dd4-41ef-9156-2c89b3b19c60": {
      "expires": "2018-08-13T00:00:00Z"
    }
  }
}], "0" ]]
```

The server extends the expiry time, but only again to its maximum limit of 7 days in the future, returning this response:

```
[["PushSubscription/set", {
  "updated": {
    "P43dcfa4-1dd4-41ef-9156-2c89b3b19c60": {
      "expires": "2018-07-15T02:22:50Z"
    }
  }
}], "0" ]]
```

7.3. Event Source

Clients that can hold transport connections open can connect directly to the JMAP server to receive push notifications via a "text/event-stream" resource, as described in [EventSource]. This is a long running HTTP request down which the server can push data.

When a change occurs in the data on the server, it pushes an event called "state" to any connected clients, with the `_StateChange_` object as the data.

The server SHOULD also send a new event id that encodes the entire server state visible to the user immediately after sending a `_state_` event. When a new connection is made to the event-source endpoint, a client following the server-sent events specification will send a Last-Event-ID HTTP header field with the last id it saw, which the server can use to work out whether the client has missed some changes. If so, it SHOULD send these changes immediately on connection.

The JMAP Session object has an `_eventSourceUrl_` property, which is in [RFC6570] URI Template (level 1) format. The URL MUST contain variables called "types", "closeafter" and "ping".

To connect to the resource, the client makes an authenticated GET request to the event-source URL with the appropriate variables substituted in:

- o "types": This MUST be either:

- * A comma-separated list of type names, e.g. "Email,CalendarEvent". The server MUST only push changes for the types in this list.
- * The single character: "*". Changes to all types are pushed.
- o "closeafter": This MUST be one of the following values:
 - * "state": The server MUST end the HTTP response after pushing a state event. This can be used by clients in environments where buffering proxies prevent the pushed data from arriving immediately, or indeed at all, when operating in the usual mode.
 - * "no": The connection is persisted by the server as a standard event-source resource.
- o "ping": A positive integer value representing a length of time in seconds, e.g. "300". If non-zero, the server MUST send an event called "ping" whenever this time elapses since the previous event was sent. This MUST NOT set a new event id. If the value is "0" the server MUST NOT send ping events.

The server MAY modify a requested ping interval to be subject to a minimum and/or maximum value. For interoperability, servers MUST NOT have a minimum allowed value higher than 30 or a maximum allowed value less than 300.

The data for the ping event MUST be a JSON object containing an `_interval_` property, the value (type "UnsignedInt") being the interval in seconds the server is using to send pings (this may be different to the requested value if the server clamped it to be within a min/max value).

Clients can monitor for the ping event to help determine when the closeafter mode may be required.

A client MAY hold open multiple connections to the event-source resource, although it SHOULD try to use a single connection for efficiency.

8. Security considerations

8.1. Transport confidentiality

To ensure the confidentiality and integrity of data sent and received via JMAP, all requests MUST use TLS 1.2 ([RFC5246]) or later,

following the recommendations in [RFC7525]. Servers SHOULD support TLS 1.3 ([RFC8446]) or later.

Clients MUST validate TLS certificate chains to protect against man-in-the-middle attacks.

8.2. Authentication scheme

A number of HTTP authentication schemes have been standardised (<<https://www.iana.org/assignments/http-authschemes/http-authschemes.xhtml>>). Servers should take care to assess the security characteristics of different schemes in relation to their needs when deciding what to implement.

Use of the Basic authentication scheme is NOT RECOMMENDED. Services that choose to use it are strongly recommended to require generation of a unique "app password" via some external mechanism for each client they wish to connect. This allows connections from different devices to be differentiated by the server, and access to be individually revoked.

8.3. Service autodiscovery

Unless secured by something like DNSSEC, DNS SRV-based autodiscovery of server details is vulnerable to a DNS poisoning attack leading to the client talking to an attacker's server instead of the real JMAP server. The attacker may then man-in-the-middle requests and depending on the authentication scheme, steal credentials to generate its own requests.

Clients that do not support SRV lookups are likely to try just using the `"/.well-known/jmap"` path directly against the domain of the username over HTTPS. Servers SHOULD ensure this path resolves or redirects to the correct JMAP Session resource to allow this to work. If this is not feasible, servers MUST ensure this path cannot be controlled by an attacker, as again it may be used to steal credentials.

8.4. JSON parsing

The security considerations of [RFC8259] apply to the use of JSON as the data interchange format.

As for any serialization format, parsers need to thoroughly check the syntax of the supplied data. JSON uses opening and closing tags for several types and structures, and it is possible that the end of supplied data will be reached when scanning for a matching closing

tag; this is an error condition and implementations need to stop scanning at the end of the supplied data.

JSON also uses a string encoding with some escape sequences to encode special characters within a string. Care is needed when processing these escape sequences to ensure that an escape sequence is fully formed before the special processing is triggered, with special care taken when the escape sequences appear adjacent to other (non-escaped) special characters or the end of data (as in the previous paragraph).

If parsing JSON into a non-textual structured data format, implementations may need to allocate storage to hold JSON string elements. Since JSON does not use explicit string lengths, the risk of denial of service due to resource exhaustion is small, but implementations may still wish to place limits on the size of allocations they are willing to make in any given context, to avoid untrusted data causing excessive memory allocation.

8.5. Denial of service

A small request may result in a very large response, and require considerable work on the server if resource limits are not enforced. JMAP provides mechanisms for advertising and enforcing a wide variety of limits for mitigating this threat, including limits on number of objects fetched in a single method call, number of methods in a single request, number of concurrent requests, etc.

JMAP servers **MUST** implement sensible limits to mitigate against resource exhaustion attacks.

8.6. Connection to unknown push server

When a push subscription is registered, the application server will make POST requests to the given URL. There are a number of security considerations that **MUST** be considered when implementing this.

The server **MUST** ensure the URL is externally resolvable to avoid server-side request forgery, where the server makes a request to a resource on its internal network.

A malicious client may use the push subscription to attempt to flood a 3rd party server with requests, creating a denial of service attack and masking the attacker's true identity. There is no guarantee the URL that was given to the JMAP server is actually a valid push server. Upon creation of a push subscription the JMAP server sends a PushVerification object to the URL and **MUST NOT** send any further requests until the client verifies it has received the initial push.

The verification code MUST contain sufficient entropy to prevent the client from being able to verify the subscription via brute force.

The verification code does not guarantee the URL is a valid push server, only that the client is able to access the data submitted to it. While the verification step significantly reduces the set of potential targets, there is still a risk that the server is unrelated to the client and being targeted for a denial of service attack.

The server MUST limit the number of push subscriptions any one user may have to ensure the user cannot cause the server to send a large number of push notifications at once, which could again be used as part of a denial-of-service attack. The rate of creation MUST also be limited to minimise the ability to abuse the verification request as an attack vector.

8.7. Push encryption

When data changes, a small object is pushed with the new state strings for the types that have changed. While the data here is minimal, a passive man-in-the-middle attacker may be able to gain useful information. To ensure confidentiality and integrity, if the push is sent via a third party outside of the control of the client and JMAP server the client MUST specify encryption keys when establishing the PushSubscription and ignore any push notification received that is not encrypted with those keys.

The privacy and security considerations of [RFC8030] and [RFC8291] also all apply to the use of the PushSubscription mechanism.

As there is no crypto algorithm agility in [RFC8291] Web Push Encryption, if new algorithms are required in the future a new specification will be needed to provide this.

8.8. Traffic analysis

While the data is encrypted, a passive observer with the ability to monitor network traffic may be able to glean information from the timing of API requests and push notifications. For example, suppose an email or calendar invitation is sent from User A (hosted on Server X) to User B (hosted on Server Y). If Server X hosts data for many users, a passive observer can see that the two servers connected but does not know who the data was for. However, if a push notification is immediately sent to User B and the attacker can observe this as well, they may reasonably conclude that someone on Server X is connecting to User B.

9. IANA considerations

9.1. Assignment of jmap service name

IANA will assign the 'jmap' service name in the 'Service Name and Transport Protocol Port Number Registry' [RFC6335].

Service Name: jmap

Transport Protocol(s): tcp

Assignee: IESG

Contact: IETF Chair

Description: JSON Meta Application Protocol

Reference: [I-D.ietf-jmap-core]

Assignment Notes: this service name was previously assigned under the name `_JSON Mail Access Protocol_`. This will be de-assigned and re-assigned with the approval of the previous assignee.

9.2. Registration of well-known URI suffix for JMAP

IANA will register the following well-known URI suffix for JMAP as described in [RFC5785]:

URI Suffix: jmap

Change Controller: IETF

Specification Document: [I-D.ietf-jmap-core], section 2.2.

9.3. Registration of the jmap URN sub-namespace

IANA will register the following URN sub-namespace in the "IETF URN Sub-namespace for Registered Protocol Parameter Identifiers" registry as described in [RFC3553].

Registered Parameter Identifier: jmap

Reference: [I-D.ietf-jmap-core], next section

IANA Registry Reference: {insert IANA registry URL for registry in next section, upon approval}

9.4. Creation of "JMAP Capabilities" registry

IANA will create a registry for JMAP capabilities as described in section 2. JMAP capabilities are advertised in the `_capabilities_` property of the JMAP Session resource. They are used to extend the functionality of a JMAP server. A capability is referenced by a URI. The JMAP capability URI can be a URN starting with `"urn:ietf:params:jmap:"` plus a unique suffix which is the index value in the jmap URN sub-namespace. Registration of a JMAP capability with another form of URI has no impact on the jmap URN sub-namespace.

This registry follows the expert review process unless the "intended use" field is `_common_` or `_placeholder_` in which case registration follows the specification required process.

A JMAP capability registration can have an intended use of `_common_`, `_placeholder_`, `_limited_`, or `_obsolete_`. IANA will list common use registrations prominently and separately from those with other intended use values.

The JMAP capability registration procedure is not a formal standards process, but rather an administrative procedure intended to allow community comment and sanity checking without excessive time delay.

A `_placeholder_` registration reserves part of the jmap urn namespace for another purpose but is typically not included in the `_capabilities_` property of the JMAP Session resource.

9.4.1. Preliminary community review

Notice of a potential JMAP common use registration SHOULD be sent to the `jmap@ietf.org` mailing list for review. This mailing list is appropriate to solicit community feedback on a proposed JMAP capability. Registrations that are not intended for common use MAY be sent to the list for review as well; doing so is entirely OPTIONAL, but is encouraged.

The intent of the public posting to this list is to solicit comments and feedback on the choice of capability name, the unambiguity of the specification document, and a review of any interoperability or security considerations. The submitter may submit a revised registration proposal or abandon the registration completely and at any time.

9.4.2. Submit request to IANA

Registration requests can be sent to iana@iana.org.

9.4.3. Designated expert review

For a limited use registration, the designated expert's (DE) primary concern is preventing name collisions and encouraging the submitter to document security and privacy considerations; a published specification is not required. For a common use registration, the DE is expected to confirm that suitable documentation as described in [RFC8126], section 4.6, is available. The DE should also verify the capability does not conflict with work that is active or already published within the IETF.

Before a period of 30 days has passed, the DE will either approve or deny the registration request and publish a notice of the decision to the JMAP WG mailing list or its successor, as well as informing IANA. A denial notice must be justified by an explanation, and in the cases where it is possible, concrete suggestions on how the request can be modified so as to become acceptable should be provided.

If the DE does not respond within 30 days, the registrant may request the IESG take action to process the request in a timely manner.

9.4.4. Change procedures

Once a JMAP capability has been published by the IANA, the change controller may request a change to its definition. The same procedure that would be appropriate for the original registration request is used to process a change request.

JMAP capability registrations may not be deleted; capabilities that are no longer believed appropriate for use can be declared obsolete by a change to their "intended use" field; such capabilities will be clearly marked in the lists published by the IANA.

Significant changes to a capability's definition should be requested only when there are serious omissions or errors in the published specification. When review is required, a change request may be denied if it renders entities that were valid under the previous definition invalid under the new definition.

The owner of a JMAP capability may pass responsibility to another person or agency by informing the IANA; this can be done without discussion or review.

The IESG may reassign responsibility for a JMAP capability. The most common case of this will be to enable changes to be made to capabilities where the author of the registration has died, moved out of contact, or is otherwise unable to make changes that are important to the community.

9.4.5. JMAP Capabilities registry template:

Capability name: (see capability property in section 2)

Specification document:

Intended use: (one of common, limited, placeholder, or obsolete)

Change controller: (_IETF_ for standards-track/BCP RFCs)

Security and privacy considerations:

9.4.6. Initial registration for JMAP core

Capability Name: "urn:ietf:params:jmap:core"

Specification document: [I-D.ietf-jmap-core], section 2

Intended use: common

Change Controller: IETF

Security and privacy considerations: [I-D.ietf-jmap-core], section 8.

9.4.7. Registration for JMAP error placeholder in JMAP capabilities registry

Capability Name: "urn:ietf:params:jmap:error:"

Specification document: [I-D.ietf-jmap-core], section 9.5

Intended use: placeholder

Change Controller: IETF

Security and privacy considerations: [I-D.ietf-jmap-core], section 8.

9.5. Creation of "JMAP Error Codes" registry

IANA will create a registry for JMAP error codes. JMAP error codes appear in the "type" member of a JSON problem details object (as described in section 3.5.1), in the "type" member in a JMAP error

object (as described in section 3.5.2), or the "type" member of a JMAP method-specific error object (such as SetError in section 5.3). When used in a problem details object, the prefix 'urn:ietf:params:jmap:error:' is always included, and when used in JMAP objects, the prefix is always omitted.

This registry follows the expert review process. Preliminary community review for this registry follows the same procedures as the JMAP capabilities registry but is optional. The change procedures for this registry are the same as the change procedures for the JMAP capabilities registry.

9.5.1. Designated expert review

The designated expert should review the following aspects of the registration:

1. Verify the error code does not conflict with existing names.
2. Verify the error code follows the syntax limitations (does not require URI encoding).
3. Encourage the error code to follow the naming convention of previously registered errors.
4. Encourage description of client behaviors that are recommended in response to the error code. These may distinguish the error code from other error codes.
5. Encourage description of when the server should issue the error as opposed to some other error code.
6. Encourage the submitter to note any security considerations associated with the error, if any. For example, an error code that might disclose existence of data the authenticated user does not have permission to know about.

Steps 3-6 are meant to promote a higher-quality registry. However, the expert is encouraged to approve any registration that would not actively harm JMAP interoperability to make this a relatively lightweight process.

9.5.2. JMAP Error Codes registry template:

JMAP Error Code:

Intended use: (one of _common_, _limited_, _obsolete_)

Change Controller: (_IETF_ for standards-track/BCP RFCs)

Reference: (optional, only if defined in an RFC.)

Description:

9.5.3. Initial JMAP Error Codes registry

JMAP Error Code	Inten ded Use	Change Control ler	Reference	Descriptio n
accountNotFound	commo n	IETF	[I-D.ietf-jmap-core] section 3.5.2	The accountId does not correspond to a valid account.
accountNotSupportedByMethod	commo n	IETF	[I-D.ietf-jmap-core] section 3.5.2	The accountId given corr esponds to a valid account, but the account does not support this method or data type.
accountReadOnly	commo n	IETF	[I-D.ietf-jmap-core] section 3.5.2	This method call would modify state in an account that is read-only (as returned on the cor responding Account object in the JMAP Session

anchorNotFound	common	IETF	[I-D.ietf-jmap-core] section 5.5	resource). An anchor argument was supplied, but it cannot be found in the results of the query.
alreadyExists	common	IETF	[I-D.ietf-jmap-core] section 5.4	The server forbids duplicates and the record already exists in the target account. An existingId property of type Id MUST be included on the error object with the id of the existing record.
cannotCalculateChanges	common	IETF	[I-D.ietf-jmap-core] sections 5.2 and 5.6	The server cannot calculate the changes from the state string given by the client.
forbidden	common	IETF	[I-D.ietf-jmap-core] sections 3.5.2, 5.3,	The action would violate an ACL or

				and 7.2.1	other permissions policy.
fromAccountNotFound	common	IETF		[I-D.ietf-jmap-core] sections 5.4 and 6.3	The fromAccountId does not correspond to a valid account.
fromAccountNotSupportedByMethod	common	IETF		[I-D.ietf-jmap-core] section 5.4	The fromAccountId given corresponds to a valid account, but the account does not support this data type.
invalidArguments	common	IETF		[I-D.ietf-jmap-core] section 3.5.2	One of the arguments is of the wrong type or otherwise invalid, or a required argument is missing.
invalidPatch	common	IETF		[I-D.ietf-jmap-core] section 5.3	The PatchObject given to update the record was not a valid patch.
invalidProperties	common	IETF		[I-D.ietf-jmap-core] section 5.3	The record given is invalid.
notFound	common	IETF		[I-D.ietf-jmap-core] section 5.3	The id given cannot be found.

notJSON	common	IETF	[I-D.ietf-jmap-core] section 3.5.1	The content type of the request was not application/json or the request did not parse as I-JSON.
notRequest	common	IETF	[I-D.ietf-jmap-core] section 3.5.1	The request parsed as JSON but did not match the type signature of the Request object.
overQuota	common	IETF	[I-D.ietf-jmap-core] section 5.3	The create would exceed a server-defined limit on the number or total size of objects of this type.
rateLimit	common	IETF	[I-D.ietf-jmap-core] section 5.3	Too many objects of this type have been created recently, and a server-defined rate limit has been reached. It may

requestTooLarge	common	IETF	[I-D.ietf-jmap-core] sections 5.1 and 5.3	work if tried again later. The total number of actions exceeds the maximum number the server is willing to process in a single method call.
invalidResultReference	common	IETF	[I-D.ietf-jmap-core] section 3.5.2	The method used a result reference for one of its arguments, but this failed to resolve.
serverFail	common	IETF	[I-D.ietf-jmap-core] section 3.5.2	An unexpected or unknown error occurred during the processing of the call. The method call made no changes to the server's state.
serverPartialFail	limited	IETF	[I-D.ietf-jmap-core] section 3.5.2	Some, but not all expected changes described by the

				method occurred. The client MUST re-synchronise impacted data to determine server state. Use of this error is strongly discouraged.
serverUnavailable	common	IETF	[I-D.ietf-jmap-core] section 3.5.2	Some internal server resource was temporarily unavailable. Attempting the same operation later (perhaps after a backoff with a random factor) may succeed.
singleton	common	IETF	[I-D.ietf-jmap-core] section 5.3	This is a singleton type, so you cannot create another one or destroy the existing one.
stateMismatch	common	IETF	[I-D.ietf-jmap-core] section 5.3	An <code>ifInState</code> argument

tooLarge	common	IETF	[I-D.ietf-jmap-core] section 5.3	was supplied and it does not match the current state. The action would result in an object that exceeds a server-defined limit for the maximum size of a single object of this type.
tooManyChanges	common	IETF	[I-D.ietf-jmap-core] section 5.6	There are more changes than the client's maxChanges argument.
unknownCapability	common	IETF	[I-D.ietf-jmap-core] section 3.5.1	The client included a capability in the "using" property of the request that the server does not support.
unknownMethod	common	IETF	[I-D.ietf-jmap-core] section 3.5.2	The server does not recognise this method name.
unsupportedFilter	common	IETF	[I-D.ietf-jmap-core]	The filter

	n		p-core] section 5.5	is syntactically valid, but the server cannot process it.
unsupportedSort	common	IETF	[I-D.ietf-jmap-core] section 5.5	The sort is syntactically valid, but includes a property the server does not support sorting on, or a collation method it does not recognise.
willDestroy	common	IETF	[I-D.ietf-jmap-core] section 5.3	The client requested an object be both updated and destroyed in the same /set request, and the server has decided to therefore ignore the update.

10. References

10.1. Normative References

[EventSource]

Hickson, I., "Server-Sent Events", 2015,
<<https://www.w3.org/TR/eventsource/>>.

- [I-D.ietf-jmap-core]
Jenkins, N. and C. Newman, "JSON Meta Application Protocol", draft-ietf-jmap-core-16 (work in progress), March 2019.
- [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>>.
- [RFC2782] Gulbrandsen, A., Vixie, P., and L. Esibov, "A DNS RR for specifying the location of services (DNS SRV)", RFC 2782, DOI 10.17487/RFC2782, February 2000, <<https://www.rfc-editor.org/info/rfc2782>>.
- [RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, DOI 10.17487/RFC2818, May 2000, <<https://www.rfc-editor.org/info/rfc2818>>.
- [RFC3339] Klyne, G. and C. Newman, "Date and Time on the Internet: Timestamps", RFC 3339, DOI 10.17487/RFC3339, July 2002, <<https://www.rfc-editor.org/info/rfc3339>>.
- [RFC3553] Mealling, M., Masinter, L., Hardie, T., and G. Klyne, "An IETF URN Sub-namespace for Registered Protocol Parameters", BCP 73, RFC 3553, DOI 10.17487/RFC3553, June 2003, <<https://www.rfc-editor.org/info/rfc3553>>.
- [RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, DOI 10.17487/RFC3629, November 2003, <<https://www.rfc-editor.org/info/rfc3629>>.
- [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>>.
- [RFC4790] Newman, C., Duerst, M., and A. Gulbrandsen, "Internet Application Protocol Collation Registry", RFC 4790, DOI 10.17487/RFC4790, March 2007, <<https://www.rfc-editor.org/info/rfc4790>>.
- [RFC5051] Crispin, M., "i;unicode-casemap - Simple Unicode Collation Algorithm", RFC 5051, DOI 10.17487/RFC5051, October 2007, <<https://www.rfc-editor.org/info/rfc5051>>.
- [RFC5322] Resnick, P., Ed., "Internet Message Format", RFC 5322, DOI 10.17487/RFC5322, October 2008, <<https://www.rfc-editor.org/info/rfc5322>>.

- [RFC5785] Nottingham, M. and E. Hammer-Lahav, "Defining Well-Known Uniform Resource Identifiers (URIs)", RFC 5785, DOI 10.17487/RFC5785, April 2010, <<https://www.rfc-editor.org/info/rfc5785>>.
- [RFC6186] Daboo, C., "Use of SRV Records for Locating Email Submission/Access Services", RFC 6186, DOI 10.17487/RFC6186, March 2011, <<https://www.rfc-editor.org/info/rfc6186>>.
- [RFC6335] Cotton, M., Eggert, L., Touch, J., Westerlund, M., and S. Cheshire, "Internet Assigned Numbers Authority (IANA) Procedures for the Management of the Service Name and Transport Protocol Port Number Registry", BCP 165, RFC 6335, DOI 10.17487/RFC6335, August 2011, <<https://www.rfc-editor.org/info/rfc6335>>.
- [RFC6570] 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/info/rfc6570>>.
- [RFC6749] Hardt, D., Ed., "The OAuth 2.0 Authorization Framework", RFC 6749, DOI 10.17487/RFC6749, October 2012, <<https://www.rfc-editor.org/info/rfc6749>>.
- [RFC6764] Daboo, C., "Locating Services for Calendaring Extensions to WebDAV (CalDAV) and vCard Extensions to WebDAV (CardDAV)", RFC 6764, DOI 10.17487/RFC6764, February 2013, <<https://www.rfc-editor.org/info/rfc6764>>.
- [RFC6838] 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/info/rfc6838>>.
- [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/info/rfc6901>>.
- [RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <<https://www.rfc-editor.org/info/rfc7230>>.

- [RFC7231] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, DOI 10.17487/RFC7231, June 2014, <<https://www.rfc-editor.org/info/rfc7231>>.
- [RFC7493] Bray, T., Ed., "The I-JSON Message Format", RFC 7493, DOI 10.17487/RFC7493, March 2015, <<https://www.rfc-editor.org/info/rfc7493>>.
- [RFC7525] Sheffer, Y., Holz, R., and P. Saint-Andre, "Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", BCP 195, RFC 7525, DOI 10.17487/RFC7525, May 2015, <<https://www.rfc-editor.org/info/rfc7525>>.
- [RFC7617] Reschke, J., "The 'Basic' HTTP Authentication Scheme", RFC 7617, DOI 10.17487/RFC7617, September 2015, <<https://www.rfc-editor.org/info/rfc7617>>.
- [RFC7807] Nottingham, M. and E. Wilde, "Problem Details for HTTP APIs", RFC 7807, DOI 10.17487/RFC7807, March 2016, <<https://www.rfc-editor.org/info/rfc7807>>.
- [RFC8030] Thomson, M., Damaggio, E., and B. Raymor, Ed., "Generic Event Delivery Using HTTP Push", RFC 8030, DOI 10.17487/RFC8030, December 2016, <<https://www.rfc-editor.org/info/rfc8030>>.
- [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/info/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/info/rfc8174>>.
- [RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <<https://www.rfc-editor.org/info/rfc8259>>.
- [RFC8264] Saint-Andre, P. and M. Blanchet, "PRECIS Framework: Preparation, Enforcement, and Comparison of Internationalized Strings in Application Protocols", RFC 8264, DOI 10.17487/RFC8264, October 2017, <<https://www.rfc-editor.org/info/rfc8264>>.

[RFC8291] Thomson, M., "Message Encryption for Web Push", RFC 8291, DOI 10.17487/RFC8291, November 2017, <<https://www.rfc-editor.org/info/rfc8291>>.

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <<https://www.rfc-editor.org/info/rfc8446>>.

10.2. Informative References

[RFC8246] McManus, P., "HTTP Immutable Responses", RFC 8246, DOI 10.17487/RFC8246, September 2017, <<https://www.rfc-editor.org/info/rfc8246>>.

Authors' Addresses

Neil Jenkins
FastMail
PO Box 234, Collins St West
Melbourne VIC 8007
Australia

Email: neilj@fastmailteam.com
URI: <https://www.fastmail.com>

Chris Newman
Oracle
440 E. Huntington Dr., Suite 400
Arcadia CA 91006
United States of America

Email: chris.newman@oracle.com

JMAP
Internet-Draft
Updates: 5788 (if approved)
Intended status: Standards Track
Expires: September 9, 2019

N. Jenkins
FastMail
C. Newman
Oracle
March 8, 2019

JMAP (JSON Meta Application Protocol) for Mail
draft-ietf-jmap-mail-16

Abstract

This document specifies a data model for synchronising email data with a server using JMAP (the JSON Meta Application Protocol). Clients can use this to efficiently search, access, organise and send messages, and get pushed notifications for fast resynchronisation when new messages are delivered or a change is made in another client.

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 September 9, 2019.

Copyright Notice

Copyright (c) 2019 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	4
1.1. Notational conventions	4
1.2. Terminology	5
1.3. Additions to the capabilities object	5
1.3.1. urn:ietf:params:jmap:mail	5
1.3.2. urn:ietf:params:jmap:submission	6
1.3.3. urn:ietf:params:jmap:vacationresponse	7
1.4. Data type support in different accounts	7
1.5. Push	8
1.5.1. Example	8
1.6. Ids	8
2. Mailboxes	8
2.1. Mailbox/get	12
2.2. Mailbox/changes	12
2.3. Mailbox/query	13
2.4. Mailbox/queryChanges	14
2.5. Mailbox/set	14
2.6. Example	14
3. Threads	18
3.1. Thread/get	19
3.1.1. Example	19
3.2. Thread/changes	19
4. Emails	20
4.1. Properties of the Email object	20
4.1.1. Metadata	21
4.1.2. Header fields parsed forms	23
4.1.3. Header fields properties	28
4.1.4. Body parts	30
4.2. Email/get	36
4.2.1. Example	38
4.3. Email/changes	39
4.4. Email/query	40
4.4.1. Filtering	40
4.4.2. Sorting	42
4.4.3. Thread collapsing	44
4.5. Email/queryChanges	44
4.6. Email/set	44
4.7. Email/copy	47
4.8. Email/import	47
4.9. Email/parse	49
4.10. Examples	51
5. Search snippets	58
5.1. SearchSnippet/get	59

5.2. Example	60
6. Identities	61
6.1. Identity/get	62
6.2. Identity/changes	62
6.3. Identity/set	62
6.4. Example	63
7. Email submission	63
7.1. EmailSubmission/get	68
7.2. EmailSubmission/changes	68
7.3. EmailSubmission/query	68
7.4. EmailSubmission/queryChanges	69
7.5. EmailSubmission/set	69
7.5.1. Example	71
8. Vacation response	74
8.1. VacationResponse/get	75
8.2. VacationResponse/set	75
9. Security considerations	75
9.1. EmailBodyPart value	75
9.2. HTML email display	76
9.3. Multiple part display	78
9.4. Email submission	78
9.5. Partial account access	79
9.6. Permission to send from an address	79
10. IANA considerations	80
10.1. JMAP capability registration for "mail"	80
10.2. JMAP capability registration for "submission"	80
10.3. JMAP capability registration for "vacationresponse"	81
10.4. IMAP and JMAP keywords registry	81
10.4.1. Registration of JMAP keyword '\$draft'	81
10.4.2. Registration of JMAP keyword '\$seen'	82
10.4.3. Registration of JMAP keyword '\$flagged'	83
10.4.4. Registration of JMAP keyword '\$answered'	84
10.4.5. Registration of '\$recent' keyword	85
10.5. Registration of "inbox" role in	85
10.6. JMAP Error Codes registry	86
10.6.1. mailboxHasChild	86
10.6.2. mailboxHasEmail	86
10.6.3. blobNotFound	86
10.6.4. tooManyKeywords	87
10.6.5. tooManyMailboxes	87
10.6.6. invalidEmail	87
10.6.7. tooManyRecipients	87
10.6.8. noRecipients	88
10.6.9. invalidRecipients	88
10.6.10. forbiddenMailFrom	88
10.6.11. forbiddenFrom	89
10.6.12. forbiddenToSend	89
11. References	89

11.1. Normative References	89
11.2. Informative References	93
11.3. URIs	93
Authors' Addresses	94

1. Introduction

JMAP ([I-D.ietf-jmap-core] - JSON Meta Application Protocol) is a generic protocol for synchronising data, such as mail, calendars or contacts, between a client and a server. It is optimised for mobile and web environments, and aims to provide a consistent interface to different data types.

This specification defines a data model for accessing a mail store over JMAP, allowing you to query, read, organise and submit mail for sending.

The data model is designed to allow a server to provide consistent access to the same data via IMAP ([RFC3501]) as well as JMAP. As in IMAP, a message must belong to a mailbox, however in JMAP its id does not change if you move it between mailboxes, and the server may allow it to belong to multiple mailboxes simultaneously (often exposed in a user agent as labels rather than folders).

As in IMAP, emails may also be assigned zero or more keywords: short arbitrary strings. These are primarily intended to store metadata to inform client display, such as unread status or whether a message has been replied to. An IANA registry allows common semantics to be shared between clients and extended easily in the future.

A message and its replies are linked on the server by a common thread id. Clients may fetch the list of messages with a particular thread id to more easily present a threaded or conversational interface.

Permissions for message access happen on a per-mailbox basis. Servers may give the user restricted permissions for certain mailboxes, for example if another user's inbox has been shared read-only with them.

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.

Type signatures, examples and property descriptions in this document follow the conventions established in section 1.1 of [I-D.ietf-jmap-core]. Data types defined in the core specification are also used in this document.

Servers MUST support all properties specified for the new data types defined in this document.

1.2. Terminology

The same terminology is used in this document as in the core JMAP specification.

1.3. Additions to the capabilities object

The capabilities object is returned as part of the JMAP Session object; see [I-D.ietf-jmap-core], section 2.

This document defines three additional capability URIs.

1.3.1. urn:ietf:params:jmap:mail

This represents support for the Mailbox, Thread, Email, and SearchSnippet data types and associated API methods. The value of this property in the JMAP session `_capabilities_` property is the empty object.

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

- o `*maxMailboxesPerEmail*`: "UnsignedInt|null" The maximum number of mailboxes that can be assigned to a single Email object (see section 4). This MUST be an integer ≥ 1 , or "null" for no limit (or rather, the limit is always the number of mailboxes in the account).
- o `*maxMailboxDepth*`: "UnsignedInt|null" The maximum depth of the mailbox hierarchy (i.e. one more than the maximum number of ancestors a mailbox may have), or "null" for no limit.
- o `*maxSizeMailboxName*`: "UnsignedInt" The maximum length, in (UTF-8) octets, allowed for the name of a mailbox. This MUST be at least 100, although it is recommended servers allow more.
- o `*maxSizeAttachmentsPerEmail*`: "UnsignedInt" The maximum total size of attachments, in octets, allowed for a single Email object. A server MAY still reject import or creation of emails with a lower

attachment size total (for example, if the body includes several megabytes of text, causing the size of the encoded MIME structure to be over some server-defined limit). Note, this limit is for the sum of unencoded attachment sizes. Users are generally not knowledgeable about encoding overhead etc., nor should they need to be, so marketing and help materials normally tell them the "max size attachments". This is the unencoded size they see on their hard drive, and so this capability matches that and allows the client to consistently enforce what the user understands as the limit. The server may separately have a limit for the total size of the RFC5322 message, which will have attachments Base64 encoded and message headers and bodies too. For example, suppose the server advertises "maxSizeAttachmentsPerEmail: 50000000" (50 MB). The enforced server limit may be for an RFC5322 size of 70000000 octets (70 MB). Even with Base64 encoding and a 2 MB HTML body, 50 MB attachments would fit under this limit.

- o `*emailQuerySortOptions*`: "String[]" A list of all the values the server supports for the "property" field of the Comparator object in an Email/query sort (see section 5.5). 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.
- o `*mayCreateTopLevelMailbox*`: "Boolean" If "true", the user may create a mailbox (see section 2) in this account with a "null" parentId. (Permission for creating a child of an existing mailbox is given by the myRights property on that mailbox.)

1.3.2. urn:ietf:params:jmap:submission

This represents support for the Identity and MessageSubmission data types and associated API methods. The value of this property in the JMAP session `_capabilities_` property is the empty object.

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

- o `*maxDelayedSend*`: "UnsignedInt" The number in seconds of the maximum delay the server supports in sending (see the EmailSubmission object description). This is "0" if the server does not support delayed send.
- o `*submissionExtensions*`: "String[String[]]" The set of SMTP submission extensions supported by the server, which the client may use when creating an EmailSubmission object (see section 7). Each key in the object is the `_ehlo-name_`, and the value is a list

of `_ehlo-args_`. A JMAP implementation that talks to a Submission [RFC6409] server SHOULD have a configuration setting that allows an administrator to modify the set of submission EHLO capabilities it may expose on this property. This allows a JMAP server to easily add access to a new submission extension without code changes. By default, the JMAP server should hide EHLO capabilities that are to do with the transport mechanism and thus are only relevant to the JMAP server (for example PIPELINING, CHUNKING, or STARTTLS). Examples of Submission extensions to include:

- * FUTURERELEASE ([RFC4865])
- * SIZE ([RFC1870])
- * DSN ([RFC3461])
- * DELIVERYBY ([RFC2852])
- * MT-PRIORITY ([RFC6710])

A JMAP server MAY advertise an extension and implement the semantics of that extension locally on the JMAP server even if a submission server used by JMAP doesn't implement it. The full IANA registry of submission extensions can be found at <https://www.iana.org/assignments/mail-parameters/mail-parameters.xhtml#mail-parameters-2>.

1.3.3. `urn:ietf:params:jmap:vacationresponse`

This represents support for the VacationResponse data type and associated API methods. The value of this property is an empty object in both the JMAP session `_capabilities_` property and an account's `_accountCapabilities_` property.

1.4. Data type support in different accounts

The server MUST include the appropriate capability strings as keys in the `_accountCapabilities_` property of any account with which the user may use the data types represented by that URI. Supported data types may differ between accounts the user has access to. For example, in the user's personal account they may have access to all three sets of data, but in a shared account they may only have data for `"urn:ietf:params:jmap:mail"`. This means they can access Mailbox/Thread/Email data in the shared account but are not allowed to send as that account (and so do not have access to Identity/MessageSubmission objects) or view/set its vacation response.

1.5. Push

Servers MUST support the JMAP push mechanisms, as specified in [I-D.ietf-jmap-core] section 7, to receive notifications when the state changes for any of the types defined in this specification.

In addition, servers that implement the "urn:ietf:params:jmap:mail" capability MUST support pushing state changes for a type called "EmailDelivery". There are no methods to act on this type; it only exists as part of the push mechanism. The state string for this MUST change whenever a new Email is added to the store, but SHOULD NOT change upon any other change to the Email objects, for example if one is marked as read or deleted.

Clients in battery constrained environments may wish to delay fetching changes initiated by the user, but fetch new messages immediately so they can notify the user. To do this, they can register for pushes for the EmailDelivery type rather than the Email type (defined in section 4).

1.5.1. Example

The client has registered for push notifications (see [I-D.ietf-jmap-core]) just for the "EmailDelivery" type. The user marks an email as read on another device, causing the state string for the "Email" type to change, however as nothing new was added to the store the "EmailDelivery" state does not change and nothing is pushed to the client. A new message arrives in the user's inbox, again causing the "Email" state to change. This time the "EmailDelivery" state also changes, and a StateChange object is pushed to the client with the new state string. The client may then resync to fetch the new message immediately.

1.6. Ids

If a JMAP Mail server also provides an IMAP interface to the data and supports [RFC8474] IMAP Extension for Object Identifiers, the ids SHOULD be the same for mailbox, thread, and email objects in JMAP.

2. Mailboxes

A mailbox represents a named set of emails. This is the primary mechanism for organising emails within an account. It is analogous to a folder or a label in other systems. A mailbox may perform a certain role in the system; see below for more details.

For compatibility with IMAP, an email MUST belong to one or more mailboxes. The email id does not change if the email changes mailboxes.

A **Mailbox** object has the following properties:

- o **id**: "Id" (immutable; server-set) The id of the mailbox.
- o **name**: "String" User-visible name for the mailbox, e.g. "Inbox". This MUST be a Net-Unicode string ([RFC5198]) of at least 1 character in length, subject to the maximum size given in the capability object. There MUST NOT be two sibling mailboxes with both the same parent and the same name. Servers MAY reject names that violate server policy (e.g., names containing slash (/) or control characters).
- o **parentId**: "Id|null" (default: null) The mailbox id for the parent of this mailbox, or "null" if this mailbox is at the top level. Mailboxes form acyclic graphs (forests) directed by the child-to-parent relationship. There MUST NOT be a loop.
- o **role**: "String|null" (default: null) Identifies mailboxes that have a particular common purpose (e.g. the "inbox"), regardless of the *_name_* (which may be localised). This value is shared with IMAP (exposed in IMAP via the [RFC6154] SPECIAL-USE extension). However, unlike in IMAP, a mailbox MUST only have a single role, and there MUST NOT be two mailboxes in the same account with the same role. Servers providing IMAP access to the same data are encouraged to enforce these extra restrictions in IMAP as well. Otherwise, it is implementation dependent how to modify the IMAP attributes to ensure compliance when exposing the data over JMAP. The value MUST be one of the mailbox attribute names listed in the IANA IMAP Mailbox Name Attributes Registry [1], as established in [RFC8457], converted to lower-case. New roles may be established here in the future. An account is not required to have mailboxes with any particular roles.
- o **sortOrder**: "UnsignedInt" (default: 0) Defines the sort order of mailboxes when presented in the client's UI, so it is consistent between devices. The number MUST be an integer in the range $0 \leq \text{sortOrder} < 2^{31}$. A mailbox with a lower order should be displayed before a mailbox with a higher order (that has the same parent) in any mailbox listing in the client's UI. Mailboxes with equal order SHOULD be sorted in alphabetical order by name. The sorting should take into account locale-specific character order convention.

- o `*totalEmails*`: "UnsignedInt" (server-set) The number of emails in this mailbox.
- o `*unreadEmails*`: "UnsignedInt" (server-set) The number of emails in this mailbox that have neither the "\$seen" keyword nor the "\$draft" keyword.
- o `*totalThreads*`: "UnsignedInt" (server-set) The number of threads where at least one email in the thread is in this mailbox.
- o `*unreadThreads*`: "UnsignedInt" (server-set) An indication of the number of "unread" threads in the mailbox. For compatibility with existing implementations, the way "unread threads" is determined is not mandated in this document. The simplest solution to implement is simply the number of threads where at least one email in the thread is both in this mailbox and has neither the "\$seen" nor "\$draft" keywords. However, a quality implementation will return the number of unread items the user would see if they opened that mailbox. A thread is shown as unread if it contains any unread messages that will be displayed when the thread is opened. Therefore "unreadThreads" should be the number of threads where at least one email in the thread has neither the "\$seen" nor the "\$draft" keyword AND at least one email in the thread is in this mailbox. Note, the unread email does not need to be the one in this mailbox. In addition, the Trash mailbox (that is a mailbox whose "role" is "trash") is treated specially:
 1. Emails that are *only* in the Trash (and no other mailbox) are ignored when calculating the "unreadThreads" count of other mailboxes.
 2. Emails that are *not* in the Trash are ignored when calculating the "unreadThreads" count for the Trash mailbox.

The result of this is that emails in the Trash are treated as though they are in a separate thread for the purposes of unread counts. It is expected that clients will hide emails in the Trash when viewing a thread in another mailbox and vice versa. This allows you to delete a single email to the Trash out of a thread. So for example, suppose you have an account where the entire contents is a single thread with 2 emails: an unread email in the Trash and a read email in the Inbox. The "unreadThreads" count would be "1" for the Trash and "0" for the Inbox.

- o `*myRights*`: "MailboxRights" (server-set) The set of rights (ACLs) the user has in relation to this mailbox. These are backwards compatible with IMAP ACLs, as defined in [RFC4314]. A `_MailboxRights_` object has the following properties:

- * ***mayReadItems***: "Boolean" If true, the user may use this mailbox as part of a filter in a `_Email/query_` call and the mailbox may be included in the `_mailboxIds_` set of `_Email_` objects. Email objects may be fetched if they are in **at least one** mailbox with this permission. If a sub-mailbox is shared but not the parent mailbox, this may be "false". Corresponds to IMAP ACLs "lr" (if mapping from IMAP, both are required for this to be "true").
- * ***mayAddItems***: "Boolean" The user may add mail to this mailbox (by either creating a new email or moving an existing one). Corresponds to IMAP ACL "i".
- * ***mayRemoveItems***: "Boolean" The user may remove mail from this mailbox (by either changing the mailboxes of an email or deleting it). Corresponds to IMAP ACLs "te" (if mapping from IMAP, both are required for this to be "true").
- * ***maySetSeen***: "Boolean" The user may add or remove the "\$seen" keyword to/from an email. If an email belongs to multiple mailboxes, the user may only modify "\$seen" if they have this permission for **all** of the mailboxes. Corresponds to IMAP ACL "s".
- * ***maySetKeywords***: "Boolean" The user may add or remove any keyword *_other than_* "\$seen" to/from an email. If an email belongs to multiple mailboxes, the user may only modify keywords if they have this permission for **all** of the mailboxes. Corresponds to IMAP ACL "w".
- * ***mayCreateChild***: "Boolean" The user may create a mailbox with this mailbox as its parent. Corresponds to IMAP ACL "k".
- * ***mayRename***: "Boolean" The user may rename the mailbox or make it a child of another mailbox. Corresponds to IMAP ACL "x" (although this covers both rename and delete permissions).
- * ***mayDelete***: "Boolean" The user may delete the mailbox itself. Corresponds to IMAP ACL "x" (although this covers both rename and delete permissions).
- * ***maySubmit***: "Boolean" Messages may be submitted directly to this mailbox. Corresponds to IMAP ACL "p".
- o ***isSubscribed***: "Boolean" Has the user indicated they wish to see this mailbox in their client? This SHOULD default to "false" for mailboxes in shared accounts the user has access to, and "true" for any new mailboxes created by the user themselves. This MUST be

stored separately per-user where multiple users have access to a shared mailbox. A user may have permission to access a large number of shared accounts, or a shared account with a very large set of mailboxes, but only be interested in the contents of a few of these. Clients may choose only to display mailboxes to the user that have the "isSubscribed" property set to "true", and offer a separate UI to allow the user to see and subscribe/unsubscribe from the full set of mailboxes. However, clients MAY choose to ignore this property, either entirely for ease of implementation, or just for an account where "isPersonal" is "true" (indicating it is the user's own, rather than a shared account). This property corresponds to IMAP ([RFC3501]) mailbox subscriptions.

For IMAP compatibility, an email in both the Trash and another mailbox SHOULD be treated by the client as existing in both places (i.e. when emptying the trash, the client should just remove the Trash mailbox and leave it in the other mailbox).

The following JMAP methods are supported:

2.1. Mailbox/get

Standard "/get" method as described in [I-D.ietf-jmap-core] section 5.1. The `_ids_` argument may be "null" to fetch all at once.

2.2. Mailbox/changes

Standard "/changes" method as described in [I-D.ietf-jmap-core] section 5.2, but with one extra argument to the response:

- o `*updatedProperties*`: "String[]|null" If only the mailbox counts (unread/total emails/threads) have changed since the old state, this will be the list of properties that may have changed, i.e. `["totalEmails", "unreadEmails", "totalThreads", "unreadThreads"]`. If the server is unable to tell if only counts have changed, it MUST just be "null".

Since counts frequently change but other properties are generally only changed rarely, the server can help the client optimise data transfer by keeping track of changes to email/thread counts separately to other state changes. The `_updatedProperties_` array may be used directly via a back-reference in a subsequent Mailbox/get call in the same single request so only these properties are returned if nothing else has changed.

2.3. Mailbox/query

Standard `"/query"` method as described in [I-D.ietf-jmap-core] section 5.5, but with the following additional request argument:

- o `*sortAsTree*`: "Boolean" (default: false) If "true", when sorting the query results and comparing two mailboxes a and b:
 - * If a is an ancestor of b, it always comes first regardless of the `_sort_` comparators. Similarly, if a is descendant of b, then b always comes first.
 - * Otherwise, if a and b do not share a `_parentId_`, find the nearest ancestors of each that do have the same `_parentId_` and compare the sort properties on those mailboxes instead.

The result of this is that the mailboxes are sorted as a tree according to the `parentId` properties, with each set of children with a common parent sorted according to the standard sort comparators.

- o `*filterAsTree*`: "Boolean" (default: false) If "true", a mailbox is only included in the query if all its ancestors are also included in the query according to the filter.

A `*FilterCondition*` object has the following properties, any of which may be omitted:

- o `*parentId*`: "Id|null" The Mailbox `_parentId_` property must match the given value exactly.
- o `*name*`: "String" The Mailbox `_name_` property contains the given string.
- o `*role*`: "String|null" The Mailbox `_role_` property must match the given value exactly.
- o `*hasAnyRole*`: "Boolean" If "true", a Mailbox matches if it has any non-"null" value for its `_role_` property.
- o `*isSubscribed*`: "Boolean" The "isSubscribed" property of the mailbox must be identical to the value given to match the condition.

A Mailbox object matches the `FilterCondition` if and only if all of the given conditions match. If zero properties are specified, it is automatically "true" for all objects.

The following Mailbox properties MUST be supported for sorting:

- o "sortOrder"
- o "name"

2.4. Mailbox/queryChanges

Standard `"/queryChanges"` method as described in [I-D.ietf-jmap-core] section 5.6.

2.5. Mailbox/set

Standard `"/set"` method as described in [I-D.ietf-jmap-core] section 5.3, but with the following additional request argument:

- o `*onDestroyRemoveMessages*`: "Boolean" (default: false) If "false", any attempt to destroy a mailbox that still has messages in it will be rejected with a "mailboxHasEmail" SetError. If "true", any messages that were in the mailbox will be removed from it, and if in no other mailboxes will be destroyed when the mailbox is destroyed.

The following extra `_SetError_` types are defined:

For `*destroy*`:

- o "mailboxHasChild": The mailbox still has at least one child mailbox. The client MUST remove these before it can delete the parent mailbox.
- o "mailboxHasEmail": The mailbox has at least one message assigned to it and the `_onDestroyRemoveMessages_` argument was "false".

2.6. Example

Fetching all mailboxes in an account:

```
[[ "Mailbox/get", {  
  "accountId": "u33084183",  
  "ids": null  
}, "0" ]]
```

And response:

```
[["Mailbox/get", {
  "accountId": "u33084183",
  "state": "78540",
  "list": [{
    "id": "MB23cfa8094c0f41e6",
    "name": "Inbox",
    "parentId": null,
    "role": "inbox",
    "sortOrder": 10,
    "totalEmails": 16307,
    "unreadEmails": 13905,
    "totalThreads": 5833,
    "unreadThreads": 5128,
    "myRights": {
      "mayAddItems": true,
      "mayRename": false,
      "maySubmit": true,
      "mayDelete": false,
      "maySetKeywords": true,
      "mayRemoveItems": true,
      "mayCreateChild": true,
      "maySetSeen": true,
      "mayReadItems": true
    },
    "isSubscribed": true
  }, {
    "id": "MB674cc24095db49ce",
    "name": "Important mail",
    ...
  }, ... ],
  "notFound": []
}, "0" ]]
```

Now suppose a message is marked read and we get a push update that the Mailbox state has changed. You might fetch the updates like this:

```
[ [ "Mailbox/changes", {
  "accountId": "u33084183",
  "sinceState": "78540"
}, "0" ],
[ "Mailbox/get", {
  "accountId": "u33084183",
  "#ids": {
    "resultOf": "0",
    "name": "Mailbox/changes",
    "path": "/created"
  }
}, "1" ],
[ "Mailbox/get", {
  "accountId": "u33084183",
  "#ids": {
    "resultOf": "0",
    "name": "Mailbox/changes",
    "path": "/updated"
  },
  "#properties": {
    "resultOf": "0",
    "name": "Mailbox/changes",
    "path": "/updatedProperties"
  }
}, "2" ] ]
```

This fetches the list of ids for created/updated/destroyed mailboxes, then using back-references fetches the data for just the created/updated mailboxes in the same request. The response may look something like this:

```
[ [ "Mailbox/changes", {
  "accountId": "u33084183",
  "oldState": "78541",
  "newState": "78542",
  "hasMoreChanges": false,
  "updatedProperties": [
    "totalEmails", "unreadEmails",
    "totalThreads", "unreadThreads"
  ],
  "created": [],
  "updated": ["MB23cfa8094c0f41e6"],
  "destroyed": []
}, "0" ],
[ "Mailbox/get", {
  "accountId": "u33084183",
  "state": "78542",
  "list": [],
  "notFound": []
}, "1" ],
[ "Mailbox/get", {
  "accountId": "u33084183",
  "state": "78542",
  "list": [{
    "id": "MB23cfa8094c0f41e6",
    "totalEmails": 16307,
    "unreadEmails": 13903,
    "totalThreads": 5833,
    "unreadThreads": 5127
  }],
  "notFound": []
}, "2" ] ]
```

Here's an example where we try to rename one mailbox and destroy another:

```
[ [ "Mailbox/set", {
  "accountId": "u33084183",
  "ifInState": "78542",
  "update": {
    "MB674cc24095db49ce": {
      "name": "Maybe important mail"
    }
  },
  "destroy": [ "MB23cfa8094c0f41e6" ]
}, "0" ] ]
```

Suppose the rename succeeds, but we don't have permission to destroy the mailbox we tried to destroy, we might get back:

```
[["Mailbox/set", {
  "accountId": "u33084183",
  "oldState": "78542",
  "newState": "78549",
  "updated": {
    "MB674cc24095db49ce": null
  },
  "notDestroyed": {
    "MB23cfa8094c0f41e6": {
      "type": "forbidden"
    }
  }
}], "0" ]]
```

3. Threads

Replies are grouped together with the original message to form a thread. In JMAP, a thread is simply a flat list of emails, ordered by date. Every email **MUST** belong to a thread, even if it is the only email in the thread.

The exact algorithm for determining whether two emails belong to the same thread is not mandated in this spec to allow for compatibility with different existing systems. For new implementations, it is suggested that two messages belong in the same thread if both of the following conditions apply:

1. An identical RFC5322 message id appears in both messages in any of the Message-Id, In-Reply-To and References headers.
2. After stripping automatically added prefixes such as "Fwd:", "Re:", "[List-Tag]" etc. and ignoring whitespace, the subjects are the same. This avoids the situation where a person replies to an old message as a convenient way of finding the right recipient to send to, but changes the subject and starts a new conversation.

If emails are delivered out of order for some reason, a user may receive two emails in the same thread but without headers that associate them with each other. The arrival of a third email in the thread may provide the missing references to join them all together into a single thread. Since the `_threadId_` of an email is immutable, if the server wishes to merge the threads, it **MUST** handle this by deleting and reinserting (with a new email id) the emails that change `threadId`.

A `*Thread*` object has the following properties:

- o `*id*`: "Id" (immutable; server-set) The id of the thread.
- o `*emailIds*`: "Id[]" (server-set) The ids of the emails in the thread, sorted by the `_receivedAt_` date of the email, oldest first. If two emails have an identical date, the sort is server-dependent but MUST be stable (sorting by id is recommended).

The following JMAP methods are supported:

3.1. Thread/get

Standard `"/get"` method as described in [I-D.ietf-jmap-core] section 5.1.

3.1.1. Example

Request:

```
[["Thread/get", {
  "accountId": "acme",
  "ids": ["f123u4", "f41u44"]
}, "#1" ]]
```

with response:

```
[["Thread/get", {
  "accountId": "acme",
  "state": "f6a7e214",
  "list": [
    {
      "id": "f123u4",
      "emailIds": [ "eaa623", "f782cbb" ]
    },
    {
      "id": "f41u44",
      "emailIds": [ "82cf7bb" ]
    }
  ],
  "notFound": []
}, "#1" ]]
```

3.2. Thread/changes

Standard `"/changes"` method as described in [I-D.ietf-jmap-core] section 5.2.

4. Emails

The **Email** object is a representation of an [RFC5322] message, which allows clients to avoid the complexities of MIME parsing, transfer encoding and character encoding.

4.1. Properties of the Email object

Broadly, a message consists of two parts: a list of header fields, then a body. The JMAP Email object provides a way to access the full structure, or to use simplified properties and avoid some complexity if this is sufficient for the client application.

While raw headers can be fetched and set, the vast majority of clients should use an appropriate parsed form for each of the headers it wants to process, as this allows it to avoid the complexities of various encodings that are required in a valid RFC5322 message.

The body of a message is normally a MIME-encoded set of documents in a tree structure. This may be arbitrarily nested, but the majority of email clients present a flat model of an email body (normally plain text or HTML), with a set of attachments. Flattening the MIME structure to form this model can be difficult, and causes inconsistency between clients. Therefore in addition to the `_bodyStructure_` property, which gives the full tree, the Email object contains 3 alternate properties with flat lists of body parts:

- o `_textBody_/_htmlBody_`: These provide a list of parts that should be rendered sequentially as the "body" of the message. This is a list rather than a single part as messages may have headers and/or footers appended/prepended as separate parts as they are transmitted, and some clients send text and images intended to be displayed inline in the body (or even videos and sound clips) as multiple parts rather than a single HTML part with referenced images.

Because MIME allows for multiple representations of the same data (using "multipart/alternative"), there is a `textBody` property (which prefers a plain text representation) and an `htmlBody` property (which prefers an HTML representation) to accommodate the two most common client requirements. The same part may appear in both lists where there is no alternative between the two.

- o `_attachments_`: This provides a list of parts that should be presented as "attachments" to the message. Some images may be solely there for embedding within an HTML body part; clients may wish to not present these as attachments in the user interface if they are displaying the HTML with the embedded images directly.

Some parts may also be in `htmlBody/textBody`; again, clients may wish to not present these as attachments in the user interface if rendered as part of the body.

The `_bodyValues_` property allows for clients to fetch the value of text parts directly without having to do a second request for the blob, and have the server handle decoding the charset into unicode. This data is in a separate property rather than on the `EmailBodyPart` object to avoid duplication of large amounts of data, as the same part may be included twice if the client fetches more than one of `bodyStructure`, `textBody` and `htmlBody`.

In the following subsections the common notational convention for wildcards has been adopted for content types, so `"foo/*"` means any content type that starts with `"foo/"`.

Due to the number of properties involved, the set of `_Email_` properties is specified over the following four sub-sections. This is purely for readability; all properties are top-level peers.

4.1.1.1. Metadata

These properties represent metadata about the [RFC5322] message, and are not derived from parsing the message itself.

- o `*id*`: "Id" (immutable; server-set) The id of the Email object. Note, this is the JMAP object id, NOT the [RFC5322] Message-ID header field value.
- o `*blobId*`: "Id" (immutable; server-set) The id representing the raw octets of the [RFC5322] message. This may be used to download the raw original message, or to attach it directly to another Email etc.
- o `*threadId*`: "Id" (immutable; server-set) The id of the Thread to which this Email belongs.
- o `*mailboxIds*`: "Id[Boolean]" The set of Mailbox ids this email belongs to. An email in the mail store MUST belong to one or more mailboxes at all times (until it is deleted). The set is represented as an object, with each key being a `_Mailbox id_`. The value for each key in the object MUST be `"true"`.
- o `*keywords*`: "String[Boolean]" (default: {}) A set of keywords that apply to the email. The set is represented as an object, with the keys being the `_keywords_`. The value for each key in the object MUST be `"true"`. Keywords are shared with IMAP. The six system keywords from IMAP are treated specially. The following four

keywords have their first character changed from "\" in IMAP to "\$" in JMAP and have particular semantic meaning:

- * "\$draft": The email is a draft the user is composing.
- * "\$seen": The email has been read.
- * "\$flagged": The email has been flagged for urgent/special attention.
- * "\$answered": The email has been replied to.

The IMAP "\Recent" keyword is not exposed via JMAP. The IMAP "\Deleted" keyword is also not present: IMAP uses a delete+expunge model, which JMAP does not. Any message with the "\Deleted" keyword MUST NOT be visible via JMAP (including as part of any mailbox counts). Users may add arbitrary keywords to an email. For compatibility with IMAP, a keyword is a case-insensitive string of 1-255 characters in the ASCII subset %x21-%x7e (excludes control chars and space), and MUST NOT include any of these characters: "() {] % * " \" Because JSON is case-sensitive, servers MUST return keywords in lower-case. The IANA Keyword Registry [2] as established in [RFC5788] assigns semantic meaning to some other keywords in common use. New keywords may be established here in the future. In particular, note:

- * "\$forwarded": The email has been forwarded.
- * "\$phishing": The email is highly likely to be phishing. Clients SHOULD warn users to take care when viewing this email and disable links and attachments.
- * "\$junk": The email is definitely spam. Clients SHOULD set this flag when users report spam to help train automated spam-detection systems.
- * "\$notjunk": The email is definitely not spam. Clients SHOULD set this flag when users indicate an email is legitimate, to help train automated spam-detection systems.
- o *size*: "UnsignedInt" (immutable; server-set) The size, in octets, of the raw data for the [RFC5322] message (as referenced by the _blobId_, i.e. the number of octets in the file the user would download).
- o *receivedAt*: "UTCDate" (immutable; default: time of creation on server) The date the email was received by the message store. This is the _internal date_ in IMAP ([RFC3501]).

4.1.2. Header fields parsed forms

Header field properties are derived from the [RFC5322] and [RFC6532] message header fields. All header fields may be fetched in a raw form. Some headers may also be fetched in a parsed form. The structured form that may be fetched depends on the header. The following forms are defined:

4.1.2.1. Raw

Type: "String"

The raw octets of the header field value from the first octet following the header field name terminating colon, up to but excluding the header field terminating CRLF. Any standards-compliant message MUST be either ASCII (RFC5322) or UTF-8 (RFC6532), however other encodings exist in the wild. A server SHOULD replace any octet or octet run with the high bit set that violates UTF-8 syntax with the unicode replacement character (U+FFFD). Any NUL octet MUST be dropped.

This form will typically have a leading space, as most generated messages insert a space after the colon that terminates the header field name.

4.1.2.2. Text

Type: "String"

The header field value with:

1. White space unfolded (as defined in [RFC5322] section 2.2.3).
2. The terminating CRLF at the end of the value removed.
3. Any SP characters at the beginning of the value removed.
4. Any syntactically correct [RFC2047] encoded sections with a known character set decoded. Any [RFC2047] encoded NUL octets or control characters are dropped from the decoded value. Any text that looks like [RFC2047] syntax but violates [RFC2047] placement or whitespace rules MUST NOT be decoded.
5. The resulting unicode converted to NFC form.

If any decodings fail, the parser SHOULD insert a unicode replacement character (U+FFFD) and attempt to continue as much as possible.

To prevent obviously nonsense behaviour, which can lead to interoperability issues, this form may only be fetched or set for the following header fields:

- o Subject
- o Comments
- o Keywords
- o List-Id
- o Any header not defined in [RFC5322] or [RFC2369]

4.1.2.3. Addresses

Type: "EmailAddress[]"

The header is parsed as an "address-list" value, as specified in [RFC5322] section 3.4, into the "EmailAddress[]" type. There is an EmailAddress item for each "mailbox" parsed from the "address-list". Group and comment information is discarded.

The *EmailAddress* object has the following properties:

- o *name*: "String|null" The `_display-name_` of the [RFC5322] `_mailbox_`. If this is a `_quoted-string_`:
 1. The surrounding DQUOTE characters are removed.
 2. Any `_quoted-pair_` is decoded.
 3. White-space is unfolded, and then any leading and trailing white-space is removed.If there is no `_display-name_` but there is a `_comment_` immediately following the `_addr-spec_`, the value of this SHOULD be used instead. Otherwise, this property is "null".
- o *email*: "String" The `_addr-spec_` of the [RFC5322] `_mailbox_`.

Any syntactically correct [RFC2047] encoded sections with a known encoding MUST be decoded, following the same rules as for the `_Text_` form.

Parsing SHOULD be best-effort in the face of invalid structure to accommodate invalid messages and semi-complete drafts. EmailAddress

objects MAY have an `_email_` property that does not conform to the `_addr-spec_` form (for example, may not contain an @ symbol).

For example, the following "address-list" string:

```
" James Smythe" <james@example.com>, Friends:
  jane@example.com, =?UTF-8?Q?John_Sm=C3=AETH?=
  <john@example.com>;
```

would be parsed as:

```
[
  { "name": "James Smythe", "email": "james@example.com" },
  { "name": null, "email": "jane@example.com" },
  { "name": "John Smith", "email": "john@example.com" }
]
```

To prevent obviously nonsense behaviour, which can lead to interoperability issues, this form may only be fetched or set for the following header fields:

- o From
- o Sender
- o Reply-To
- o To
- o Cc
- o Bcc
- o Resent-From
- o Resent-Sender
- o Resent-Reply-To
- o Resent-To
- o Resent-Cc
- o Resent-Bcc
- o Any header not defined in [RFC5322] or [RFC2369]

4.1.2.4. GroupedAddresses

Type: "EmailAddressGroup[]"

This is similar to the Addresses form but preserves group information. The header is parsed as an "address-list" value, as specified in [RFC5322] section 3.4, into the "GroupedAddresses[]" type. Consecutive mailboxes that are not part of a group are still collected under an EmailAddressGroup object to provide a uniform type.

The *EmailAddressGroup* object has the following properties:

- o *name*: "String|null" The `_display-name_` of the [RFC5322] `_group_`, or "null" if the addresses are not part of a group. If this is a `_quoted-string_` it is processed the same as the `_name_` in the `_EmailAddress_` type.
- o *addresses*: "EmailAddress[]" The `_mailbox_es` that belong to this group, represented as EmailAddress objects.

Any syntactically correct [RFC2047] encoded sections with a known encoding MUST be decoded, following the same rules as for the `_Text_` form.

Parsing SHOULD be best-effort in the face of invalid structure to accommodate invalid messages and semi-complete drafts.

For example, the following "address-list" string:

```
" James Smythe" <james@example.com>, Friends:
jane@example.com, =?UTF-8?Q?John_Sm=C3=AEth?=
<john@example.com>;
```

would be parsed as:

```
[
  { "name": null, "addresses": [
    { "name": "James Smythe", "email": "james@example.com" }
  ]},
  { "name": "Friends", "addresses": [
    { "name": null, "email": "jane@example.com" },
    { "name": "John Smith", "email": "john@example.com" }
  ]}
]
```

To prevent obviously nonsense behaviour, which can lead to interoperability issues, this form may only be fetched or set for the same header fields as the `_Addresses_` form.

4.1.2.5. MessageIds

Type: "String[]|null"

The header is parsed as a list of "msg-id" values, as specified in [RFC5322] section 3.6.4, into the "String[]" type. CFWS and surrounding angle brackets ("<>") are removed. If parsing fails, the value is "null".

To prevent obviously nonsense behaviour, which can lead to interoperability issues, this form may only be fetched or set for the following header fields:

- o Message-ID
- o In-Reply-To
- o References
- o Resent-Message-ID
- o Any header not defined in [RFC5322] or [RFC2369]

4.1.2.6. Date

Type: "Date|null"

The header is parsed as a "date-time" value, as specified in [RFC5322] section 3.3, into the "Date" type. If parsing fails, the value is "null".

To prevent obviously nonsense behaviour, which can lead to interoperability issues, this form may only be fetched or set for the following header fields:

- o Date
- o Resent-Date
- o Any header not defined in [RFC5322] or [RFC2369]

4.1.2.7. URLs

Type: "String[]|null"

The header is parsed as a list of URLs, as described in [RFC2369], into the "String[]" type. Values do not include the surrounding angle brackets or any comments in the header with the URLs. If parsing fails, the value is "null".

To prevent obviously nonsense behaviour, which can lead to interoperability issues, this form may only be fetched or set for the following header fields:

- o List-Help
- o List-Unsubscribe
- o List-Subscribe
- o List-Post
- o List-Owner
- o List-Archive
- o Any header not defined in [RFC5322] or [RFC2369]

4.1.3. Header fields properties

The following low-level *Email* property is specified for complete access to the header data of the message:

- o *headers*: "EmailHeader[]" (immutable) This is a list of all [RFC5322] header fields, in the same order they appear in the message. An *EmailHeader* object has the following properties:
 - * *name*: "String" The header `_field name_` as defined in [RFC5322], with the same capitalization that it has in the message.
 - * *value*: "String" The header `_field value_` as defined in [RFC5322], in `_Raw_` form.

In addition, the client may request/send properties representing individual header fields of the form:

header:{header-field-name}

Where "{header-field-name}" means any series of one or more printable ASCII characters (i.e. characters that have values between 33 and 126, inclusive), except colon. The property may also have the following suffixes:

- o *:as{header-form}* This means the value is in a parsed form, where "{header-form}" is one of the parsed-form names specified above. If not given, the value is in `_Raw_` form.
- o *:all* This means the value is an array, with the items corresponding to each instance of the header field, in the order they appear in the message. If this suffix is not used, the result is the value of the `*last*` instance of the header field (i.e. identical to the `*last*` item in the array if `:all` is used), or `"null"` if none.

If both suffixes are used, they **MUST** be specified in the order above. Header field names are matched case-insensitively. The value is typed according to the requested form, or an array of that type if `:all` is used. If no header fields exist in the message with the requested name, the value is `"null"` if fetching a single instance, or the empty array if requesting `:all`.

As a simple example, if the client requests a property called `"header:subject"`, this means find the `_last_` header field in the message named `"subject"` (matched case-insensitively) and return the value in `_Raw_` form, or `"null"` if no header of this name is found.

For a more complex example, consider the client requesting a property called `"header:Resent-To:asAddresses:all"`. This means:

1. Find `_all_` header fields named `Resent-To` (matched case-insensitively).
2. For each instance parse the header field value in the `_Addresses_` form.
3. The result is of type `"EmailAddress[][]"` - each item in the array corresponds to the parsed value (which is itself an array) of the `Resent-To` header field instance.

The following convenience properties are also specified for the `*Email*` object:

- o `*messageId*`: `"String[]|null"` (immutable) The value is identical to the value of `_header:Message-ID:asMessageIds_`. For messages conforming to RFC5322 this will be an array with a single entry.

- o `*inReplyTo*`: "String[]|null" (immutable) The value is identical to the value of `_header:In-Reply-To:asMessageIds_`.
- o `*references*`: "String[]|null" (immutable) The value is identical to the value of `_header:References:asMessageIds_`.
- o `*sender*`: "EmailAddress[]|null" (immutable) The value is identical to the value of `_header:Sender:asAddresses_`.
- o `*from*`: "EmailAddress[]|null" (immutable) The value is identical to the value of `_header:From:asAddresses_`.
- o `*to*`: "EmailAddress[]|null" (immutable) The value is identical to the value of `_header:To:asAddresses_`.
- o `*cc*`: "EmailAddress[]|null" (immutable) The value is identical to the value of `_header:Cc:asAddresses_`.
- o `*bcc*`: "EmailAddress[]|null" (immutable) The value is identical to the value of `_header:Bcc:asAddresses_`.
- o `*replyTo*`: "EmailAddress[]|null" (immutable) The value is identical to the value of `_header:Reply-To:asAddresses_`.
- o `*subject*`: "String|null" (immutable) The value is identical to the value of `_header:Subject:asText_`.
- o `*sentAt*`: "Date|null" (immutable; default on creation: current server time) The value is identical to the value of `_header>Date:asDate_`.

4.1.4. Body parts

These properties are derived from the [RFC5322] message body and its [RFC2045] MIME entities.

A `*EmailBodyPart*` object has the following properties:

- o `*partId*`: "String|null" Identifies this part uniquely within the Email. This is scoped to the `_emailId_` and has no meaning outside of the JMAP Email object representation. This is "null" if, and only if, the part is of type "multipart/*".
- o `*blobId*`: "Id|null" The id representing the raw octets of the contents of the part, after decoding any known `_Content-Transfer-Encoding_` (as defined in [RFC2045]), or "null" if, and only if, the part is of type "multipart/*". Note, two parts may be transfer-encoded differently but have the same blob id if their

decoded octets are identical and the server is using a secure hash of the data for the blob id. If the transfer encoding is unknown, it is treated as though it had no transfer-encoding.

- o ***size***: "UnsignedInt" The size, in octets, of the raw data after content transfer decoding (as referenced by the `_blobId_`, i.e. the number of octets in the file the user would download).
- o ***headers***: "EmailHeader[]" This is a list of all header fields in the part, in the order they appear in the message. The values are in `_Raw_` form.
- o ***name***: "String|null" This is the [RFC2231] decoded `_filename_` parameter of the `_Content-Disposition_` header field, or (for compatibility with existing systems) if not present then the [RFC2047] decoded `_name_` parameter of the `_Content-Type_` header field.
- o ***type***: "String" The value of the `_Content-Type_` header field of the part, if present, otherwise the implicit type as per the MIME standard ("text/plain", or "message/rfc822" if inside a "multipart/digest"). CFWS is removed and any parameters are stripped.
- o ***charset***: "String|null" The value of the charset parameter of the `_Content-Type_` header field, if present, or "null" if the header field is present but not of type "text/*". If there is no `_Content-Type_` header field, or it exists and is of type "text/*" but has no charset parameter, this is the implicit charset as per the MIME standard: "us-ascii".
- o ***disposition***: "String|null" The value of the `_Content-Disposition_` header field of the part, if present, otherwise "null". CFWS is removed and any parameters are stripped.
- o ***cid***: "String|null" The value of the `_Content-Id_` header field of the part, if present, otherwise "null". CFWS and surrounding angle brackets ("<>") are removed. This may be used to reference the content from within an [HTML] body part using the "cid:" protocol, as defined in [RFC2392].
- o ***language***: "String[]|null" The list of language tags, as defined in [RFC3282], in the `_Content-Language_` header field of the part, if present.
- o ***location***: "String|null" The URI, as defined in [RFC2557], in the `_Content-Location_` header field of the part, if present.

- o `*subParts*`: "EmailBodyPart[]"|null" If type is "multipart/*", this contains the body parts of each child.

In addition, the client may request/send EmailBodyPart properties representing individual header fields, following the same syntax and semantics as for the Email object, e.g. "header:Content-Type".

The following `*Email*` properties are specified for access to the body data of the message:

- o `*bodyStructure*`: "EmailBodyPart" (immutable) This is the full MIME structure of the message body, represented as an array of the message's top-level MIME parts, without recursing into "message/rfc822" or "message/global" parts. Note that EmailBodyParts may have subParts if they are of type "multipart/*".
- o `*bodyValues*`: "String[EmailBodyValue]" (immutable) This is a map of `_partId_` to an `*EmailBodyValue*` object for none, some or all "text/*" parts. Which parts are included and whether the value is truncated is determined by various arguments to `_Email/get_` and `_Email/parse_`. An `*EmailBodyValue*` object has the following properties:
 - * `*value*`: "String" The value of the body part after decoding `_Content-Transfer-Encoding_` and decoding the `_Content-Type_` charset, if both known to the server, and with any CRLF replaced with a single LF. The server MAY use heuristics to determine the charset to use for decoding if the charset is unknown, or if no charset is given, or if it believes the charset given is incorrect. Decoding is best-effort and SHOULD insert the unicode replacement character (U+FFFD) and continue when a malformed section is encountered. Note that due to the charset decoding and line ending normalisation, the length of this string will probably not be exactly the same as the `_size_` property on the corresponding EmailBodyPart.
 - * `*isEncodingProblem*`: "Boolean" (default: false) This is "true" if malformed sections were found while decoding the charset, or the charset was unknown, or the content-transfer-encoding was unknown.
 - * `*isTruncated*`: "Boolean" (default: false) This is "true" if the `_value_` has been truncated.

See the security considerations section for issues related to truncation and heuristic determination of content-type and charset.

- o `*textBody*`: "EmailBodyPart[]" (immutable) A list of "text/plain", "text/html", "image/*", "audio/*" and/or "video/*" parts to display (sequentially) as the message body, with a preference for "text/plain" when alternative versions are available.
- o `*htmlBody*`: "EmailBodyPart[]" (immutable) A list of "text/plain", "text/html", "image/*", "audio/*" and/or "video/*" parts to display (sequentially) as the message body, with a preference for "text/html" when alternative versions are available.
- o `*attachments*`: "EmailBodyPart[]" (immutable) A list of all parts in `_bodyStructure_`, traversing depth-first, which satisfy either of the following conditions:
 - * not of type "multipart/*" and not included in `_textBody_` or `_htmlBody_`
 - * of type "image/*", "audio/*" or "video/*" and not in both `_textBody_` and `_htmlBody_`

None of these parts include subParts, including "message/*" types. Attached messages may be fetched using the Email/parse method and the blobId. Note, an [HTML] body part may reference image parts in attachments using "cid:" links to reference the `_Content-Id_`, as defined in [RFC2392], or by referencing the `_Content-Location_`.

- o `*hasAttachment*`: "Boolean" (immutable; server-set) This is "true" if there are one or more parts in the message that a client UI should offer as downloadable. A server SHOULD set hasAttachment to "true" if the `_attachments_` list contains at least one item that does not have "Content-Disposition: inline". The server MAY ignore parts in this list that are processed automatically in some way, or are referenced as embedded images in one of the "text/html" parts of the message. The server MAY set hasAttachment based on implementation-defined or site configurable heuristics.
- o `*preview*`: "String" (immutable; server-set) A plain text fragment of the message body. This is intended to be shown as a preview line on a mailbox listing, and may be truncated when shown. The server may choose which part of the message to include in the preview; skipping quoted sections and salutations and collapsing white-space can result in a more useful preview. This MUST NOT be more than 256 characters in length. As this is derived from the message content by the server, and the algorithm for doing so could change over time, fetching this for an email a second time MAY return a different result. However, the previous value is not considered incorrect, and the change SHOULD NOT cause the Email object to be considered as changed by the server.

The exact algorithm for decomposing bodyStructure into textBody, htmlBody and attachments part lists is not mandated, as this is a quality-of-service implementation issue and likely to require workarounds for malformed content discovered over time. However, the following algorithm (expressed here in JavaScript) is suggested as a starting point, based on real-world experience:

```
function isInlineMediaType ( type ) {
  return type.startsWith( 'image/' ) ||
         type.startsWith( 'audio/' ) ||
         type.startsWith( 'video/' );
}

function parseStructure ( parts, multipartType, inAlternative,
                        htmlBody, textBody, attachments ) {

  // For multipartType == alternative
  let textLength = textBody ? textBody.length : -1;
  let htmlLength = htmlBody ? htmlBody.length : -1;

  for ( let i = 0; i < parts.length; i += 1 ) {
    let part = parts[i];
    let isMultipart = part.type.startsWith( 'multipart/' );
    // Is this a body part rather than an attachment
    let isInline = part.disposition !== "attachment" &&
                  // Must be one of the allowed body types
                  ( part.type === "text/plain" ||
                    part.type === "text/html" ||
                    isInlineMediaType( part.type ) ) &&
                  // If multipart/related, only the first part can be inline
                  // If a text part with a filename, and not the first item
                  // in the multipart, assume it is an attachment
                  ( i === 0 ||
                    ( multipartType !== "related" &&
                      ( isInlineMediaType( part.type ) || !part.name ) ) );

    if ( isMultipart ) {
      let subMultiType = part.type.split( '/' )[1];
      parseStructure( part.subParts, subMultiType,
                    inAlternative || ( subMultiType === 'alternative' ),
                    htmlBody, textBody, attachments );
    } else if ( isInline ) {
      if ( multipartType === 'alternative' ) {
        switch ( part.type ) {
          case 'text/plain':
            textBody.push( part );
            break;
          case 'text/html':
```

```
        htmlBody.push( part );
        break;
    default:
        attachments.push( part );
        break;
    }
    continue;
} else if ( inAlternative ) {
    if ( part.type == 'text/plain' ) {
        htmlBody = null;
    }
    if ( part.type == 'text/html' ) {
        textBody = null;
    }
}
if ( textBody ) {
    textBody.push( part );
}
if ( htmlBody ) {
    htmlBody.push( part );
}
if ( ( !textBody || !htmlBody ) &&
    isInlineMediaType( part.type ) ) {
    attachments.push( part );
}
} else {
    attachments.push( part );
}
}

if ( multipartType == 'alternative' && textBody && htmlBody ) {
    // Found HTML part only
    if ( textLength == textBody.length &&
        htmlLength != htmlBody.length ) {
        for ( let i = htmlLength; i < htmlBody.length; i += 1 ) {
            textBody.push( htmlBody[i] );
        }
    }
    // Found plain text part only
    if ( htmlLength == htmlBody.length &&
        textLength != textBody.length ) {
        for ( let i = textLength; i < textBody.length; i += 1 ) {
            htmlBody.push( textBody[i] );
        }
    }
}
}
```



```
// Usage:
let htmlBody = [];
let textBody = [];
let attachments = [];

parseStructure( [ bodyStructure ], 'mixed', false,
  htmlBody, textBody, attachments );
```

For instance, consider a message with both text and HTML versions that's then gone through a list software manager that attaches a header/footer. It might have a MIME structure something like:

```
multipart/mixed
  text/plain, content-disposition=inline - A
  multipart/mixed
    multipart/alternative
      multipart/mixed
        text/plain, content-disposition=inline - B
        image/jpeg, content-disposition=inline - C
        text/plain, content-disposition=inline - D
      multipart/related
        text/html - E
        image/jpeg - F
    image/jpeg, content-disposition=attachment - G
    application/x-excel - H
    message/rfc822 - J
  text/plain, content-disposition=inline - K
```

In this case, the above algorithm would decompose this to:

```
textBody => [ A, B, C, D, K ]
htmlBody => [ A, E, K ]
attachments => [ C, F, G, H, J ]
```

4.2. Email/get

Standard "/get" method as described in [I-D.ietf-jmap-core] section 5.1, with the following additional request arguments:

- o ***bodyProperties***: "String[]" A list of properties to fetch for each EmailBodyPart returned. If omitted, this defaults to:


```
[ "partId", "blobId", "size", "name", "type", "charset",
  "disposition", "cid", "language", "location" ]
```
- o ***fetchTextBodyValues***: "Boolean" (default: false) If "true", the `_bodyValues_` property includes any "text/*" part in the "textBody" property.

- o `*fetchHTMLBodyValues*`: "Boolean" (default: false) If "true", the `_bodyValues_` property includes any "text/*" part in the "htmlBody" property.
- o `*fetchAllBodyValues*`: "Boolean" (default: false) If "true", the `_bodyValues_` property includes any "text/*" part in the "bodyStructure" property.
- o `*maxBodyValueBytes*`: "UnsignedInt" (default: 0) If greater than zero, the `_value_` property of any `EmailBodyValue` object returned in `_bodyValues_` MUST be truncated if necessary so it does not exceed this number of octets in size. If "0" (the default), no truncation occurs. The server MUST ensure the truncation results in valid UTF-8 and does not occur mid-codepoint. If the part is of type "text/html", the server SHOULD NOT truncate inside an HTML tag, e.g. in the middle of "". There is no requirement for the truncated form to be a balanced tree or valid HTML (indeed, the original source may well be neither of these things).

If the standard `_properties_` argument is omitted or "null", the following default MUST be used instead of "all" properties:

```
[ "id", "blobId", "threadId", "mailboxIds", "keywords", "size",  
  "receivedAt", "messageId", "inReplyTo", "references", "sender", "from",  
  "to", "cc", "bcc", "replyTo", "subject", "sentAt", "hasAttachment",  
  "preview", "bodyValues", "textBody", "htmlBody", "attachments" ]
```

The following properties are expected to be fast to fetch in a quality implementation:

- o `id`
- o `blobId`
- o `threadId`
- o `mailboxIds`
- o `keywords`
- o `size`
- o `receivedAt`
- o `messageId`
- o `inReplyTo`

- o sender
- o from
- o to
- o cc
- o bcc
- o replyTo
- o subject
- o sentAt
- o hasAttachment
- o preview

Clients SHOULD take care when fetching any other properties, as there may be significantly longer latency in fetching and returning the data.

As specified above, parsed forms of headers may only be used on appropriate header fields. Attempting to fetch a form that is forbidden (e.g. "header:From:asDate") MUST result in the method call being rejected with an "invalidArguments" error.

Where a specific header is requested as a property, the capitalization of the property name in the response MUST be identical to that used in the request.

4.2.1. Example

Request:

```
[[ "Email/get", {  
  "ids": [ "f123u456", "f123u457" ],  
  "properties": [ "threadId", "mailboxIds", "from", "subject",  
    "receivedAt", "header:List-POST:asURLs",  
    "htmlBody", "bodyValues" ],  
  "bodyProperties": [ "partId", "blobId", "size", "type" ],  
  "fetchHTMLBodyValues": true,  
  "maxBodyValueBytes": 256  
}, "#1" ]]
```

and response:

```

[[ "Email/get", {
  "accountId": "abc",
  "state": "41234123231",
  "list": [
    {
      "id": "f123u457",
      "threadId": "ef1314a",
      "mailboxIds": { "f123": true },
      "from": [{ "name": "Joe Bloggs", "email": "joe@example.com" }],
      "subject": "Dinner on Thursday?",
      "receivedAt": "2013-10-13T14:12:00Z",
      "header:List-POST:asURLs": [
        "mailto:partytime@lists.example.com"
      ],
      "htmlBody": [{
        "partId": "1",
        "blobId": "B841623871",
        "size": 283331,
        "type": "text/html"
      }, {
        "partId": "2",
        "blobId": "B319437193",
        "size": 10343,
        "type": "text/plain"
      }],
      "bodyValues": {
        "1": {
          "isEncodingProblem": false,
          "isTruncated": true,
          "value": "<html><body><p>Hello ...</p></body></html>"
        },
        "2": {
          "isEncodingProblem": false,
          "isTruncated": false,
          "value": "-- Sent by your friendly mailing list ..."
        }
      }
    }
  ],
  "notFound": [ "f123u456" ]
}, "#1" ]]
```

4.3. Email/changes

Standard `/changes` method as described in [I-D.ietf-jmap-core] section 5.2. If generating intermediate states for a large set of changes, it is recommended that newer changes are returned first, as these are generally of more interest to users.

4.4. Email/query

Standard `/query` method as described in [I-D.ietf-jmap-core] section 5.5, but with the following additional request arguments:

- o `*collapseThreads*`: "Boolean" (default: false) If "true", emails in the same thread as a previous email in the list (given the filter and sort order) will be removed from the list. This means only one email at most will be included in the list for any given thread.

In quality implementations, the query `"total"` property is expected to be fast to calculate when the filter consists solely of a single `"inMailbox"` property, as it is the same as the `totalEmails` or `totalThreads` properties (depending on whether `collapseThreads` is true) of the associated Mailbox object.

4.4.1. Filtering

A `*FilterCondition*` object has the following properties, any of which may be omitted:

- o `*inMailbox*`: "Id" A mailbox id. An email must be in this mailbox to match the condition.
- o `*inMailboxOtherThan*`: "Id[]" A list of mailbox ids. An email must be in at least one mailbox not in this list to match the condition. This is to allow messages solely in trash/spam to be easily excluded from a search.
- o `*before*`: "UTCDate" The `_receivedAt_` date-time of the email must be before this date-time to match the condition.
- o `*after*`: "UTCDate" The `_receivedAt_` date-time of the email must be the same or after this date-time to match the condition.
- o `*minSize*`: "UnsignedInt" The `_size_` of the email in octets must be equal to or greater than this number to match the condition.
- o `*maxSize*`: "UnsignedInt" The `_size_` of the email in octets must be less than this number to match the condition.
- o `*allInThreadHaveKeyword*`: "String" All emails (including this one) in the same thread as this email must have the given keyword to match the condition.

- o `*someInThreadHaveKeyword*`: "String" At least one email (possibly this one) in the same thread as this email must have the given keyword to match the condition.
- o `*noneInThreadHaveKeyword*`: "String" All emails (including this one) in the same thread as this email must **not** have the given keyword to match the condition.
- o `*hasKeyword*`: "String" This email must have the given keyword to match the condition.
- o `*notKeyword*`: "String" This email must not have the given keyword to match the condition.
- o `*hasAttachment*`: "Boolean" The "hasAttachment" property of the email must be identical to the value given to match the condition.
- o `*text*`: "String" Looks for the text in emails. The server **MUST** look up text in the `_from_`, `_to_`, `_cc_`, `_bcc_`, `_subject_` header fields of the message, and **SHOULD** look inside any "text/*" or other body parts that may be converted to text by the server. The server **MAY** extend the search to any additional textual property.
- o `*from*`: "String" Looks for the text in the `_From_` header field of the message.
- o `*to*`: "String" Looks for the text in the `_To_` header field of the message.
- o `*cc*`: "String" Looks for the text in the `_Cc_` header field of the message.
- o `*bcc*`: "String" Looks for the text in the `_Bcc_` header field of the message.
- o `*subject*`: "String" Looks for the text in the `_subject_` property of the email.
- o `*body*`: "String" Looks for the text in one of the body parts of the email. The server **MAY** exclude MIME body parts with content media types other than "text/_" and "message/_" from consideration in search matching. Care should be taken to match based on the text content actually presented to an end-user by viewers for that media type, or otherwise identified as appropriate for search indexing. Matching document metadata uninteresting to an end-user (e.g., markup tag and attribute names) is undesirable.

- o `*header*`: `"String[]"` The array **MUST** contain either one or two elements. The first element is the name of the header field to match against. The second (optional) element is the text to look for in the header field value. If not supplied, the message matches simply if it has a header field of the given name.

If zero properties are specified on the `FilterCondition`, the condition **MUST** always evaluate to `"true"`. If multiple properties are specified, **ALL** must apply for the condition to be `"true"` (it is equivalent to splitting the object into one-property conditions and making them all the child of an **AND** filter operator).

The exact semantics for matching `"String"` fields is **deliberately not defined** to allow for flexibility in indexing implementation, subject to the following:

- o Any syntactically correct [RFC2047] encoded sections of header fields with a known encoding **SHOULD** be decoded before attempting to match text.
- o When searching inside a `"text/html"` body part, any text considered markup rather than content **SHOULD** be ignored, including HTML tags and most attributes, anything inside the `"<head>"` tag, CSS and JavaScript. Attribute content intended for presentation to the user such as `"alt"` and `"title"` **SHOULD** be considered in the search.
- o Text **SHOULD** be matched in a case-insensitive manner.
- o Text contained in either (but matched) single or double quotes **SHOULD** be treated as a **phrase search**, that is a match is required for that exact word or sequence of words, excluding the surrounding quotation marks. Use `"\""`, `"\'"` and `"\\"` to match a literal `"\""`, `"\'"` and `"\""` respectively in a phrase.
- o Outside of a phrase, white-space **SHOULD** be treated as dividing separate tokens that may be searched for separately, but **MUST** all be present for the email to match the filter.
- o Tokens (not part of a phrase) **MAY** be matched on a whole-word basis using stemming (so for example a text search for `"bus"` would match `"buses"` but not `"business"`).

4.4.2. Sorting

The following value for the `"property"` field on the `Comparator` object **MUST** be supported for sorting:

- o **receivedAt** - The `_receivedAt_` date as returned in the Email object.

The following values for the "property" field on the Comparator object SHOULD be supported for sorting. When specifying a "hasKeyword", "allInThreadHaveKeyword" or "someInThreadHaveKeyword" sort, the Comparator object MUST also have a `_keyword_` property.

- o **size** - The `_size_` as returned in the Email object.
- o **from** - This is taken to be either the "name" part, or if "null"/empty then the "email" part, of the **first** EmailAddress object in the `_from_` property. If still none, consider the value to be the empty string.
- o **to** - This is taken to be either the "name" part, or if "null"/empty then the "email" part, of the **first** EmailAddress object in the `_to_` property. If still none, consider the value to be the empty string.
- o **subject** - This is taken to be the base subject of the email, as defined in section 2.1 of [RFC5256].
- o **sentAt** - The `_sentAt_` property on the Email object.
- o **hasKeyword** - This value MUST be considered "true" if the email has the keyword given as an additional `_keyword_` property on the `_Comparator_` object, or "false" otherwise.
- o **allInThreadHaveKeyword** - This value MUST be considered "true" for the email if **all** of the emails in the same thread (regardless of mailbox) have the keyword given as an additional `_keyword_` property on the `_Comparator_` object.
- o **someInThreadHaveKeyword** - This value MUST be considered "true" for the email if **any** of the emails in the same thread (regardless of mailbox) have the keyword given as an additional `_keyword_` property on the `_Comparator_` object.

The server MAY support sorting based on other properties as well. A client can discover which properties are supported by inspecting the server's `_capabilities_` object (see section 1.3).

Example sort:


```
[{
  "property": "someInThreadHaveKeyword",
  "keyword": "$flagged",
  "isAscending": false
}, {
  "property": "subject",
  "collation": "i;ascii-casemap"
}, {
  "property": "receivedAt",
  "isAscending": false
}]
```

This would sort emails in flagged threads first (the thread is considered flagged if any email within it is flagged), and then in subject order, then newest first for messages with the same subject. If two emails have both identical flagged status, subject and date, the order is server-dependent but must be stable.

4.4.3. Thread collapsing

When `_collapseThreads_` is "true", then after filtering and sorting the email list, the list is further winnowed by removing any emails for a thread id that has already been seen (when passing through the list sequentially). A thread will therefore only appear *once* in the result, at the position of the first email in the list that belongs to the thread (given the current sort/filter).

4.5. Email/queryChanges

Standard `/queryChanges` method as described in [I-D.ietf-jmap-core] section 5.6, with the following additional request arguments:

- o `*collapseThreads*`: "Boolean" (default: false) The `_collapseThreads_` argument that was used with `_Email/query_`.

4.6. Email/set

Standard `/set` method as described in [I-D.ietf-jmap-core] section 5.3. The `_Email/set_` method encompasses:

- o Creating a draft
- o Changing the keywords of an email (e.g. unread/flagged status)
- o Adding/removing an email to/from mailboxes (moving a message)
- o Deleting emails

The format of the keywords/mailboxIds properties means that when updating an email you can either replace the entire set of keywords/mailboxes (by setting the full value of the property) or add/remove individual ones using the JMAP patch syntax (see [I-D.ietf-jmap-core], section 5.3 for the specification and section 5.7 for an example).

Due to the format of the Email object, when creating an email there are a number of ways to specify the same information. To ensure that the RFC5322 email to create is unambiguous, the following constraints apply to Email objects submitted for creation:

- o The `_headers_` property MUST NOT be given, on either the top-level email or an `EmailBodyPart` - the client must set each header field as an individual property.
- o There MUST NOT be two properties that represent the same header field (e.g. "header:from" and "from") within the Email or particular `EmailBodyPart`.
- o Header fields MUST NOT be specified in parsed forms that are forbidden for that particular field.
- o Header fields beginning "Content-" MUST NOT be specified on the Email object, only on `EmailBodyPart` objects.
- o If a `bodyStructure` property is given, there MUST NOT be `textBody`, `htmlBody` or `attachments` properties.
- o If given, the `bodyStructure` `EmailBodyPart` MUST NOT contain a property representing a header field that is already defined on the top-level Email object.
- o If given, `textBody` MUST contain exactly one body part, of type "text/plain".
- o If given, `htmlBody` MUST contain exactly one body part, of type "text/html".
- o Within an `EmailBodyPart`:
 - * The client may specify a `partId` OR a `blobId` but not both. If a `partId` is given, this `partId` MUST be present in the `bodyValues` property.
 - * The `charset` property MUST be omitted if a `partId` is given (the part's content is included in `bodyValues` and the server may choose any appropriate encoding).

- * The size property MUST be omitted if a partId is given. If a blobId is given, it may be included but is ignored by the server (the size is actually calculated from the blob content itself).
- * A "Content-Transfer-Encoding" header field MUST NOT be given.
- o Within an EmailBodyValue object, isEncodingProblem and isTruncated MUST be either "false" or omitted.

Creation attempts that violate any of this SHOULD be rejected with an "invalidProperties" error, however a server MAY choose to modify the Email (e.g. choose between conflicting headers, use a different content-encoding etc.) to comply with its requirements instead.

The server MAY also choose to set additional headers. If not included, the server MUST generate and set a "Message-ID" header field in conformance with [RFC5322] section 3.6.4, and a "Date" header field in conformance with section 3.6.1.

The final RFC5322 email generated may be invalid. For example, if it is a half-finished draft, the "To" field may have a value that does not conform to the required syntax for this header field. The message will be checked for strict conformance when submitted for sending (see the EmailSubmission object description).

Destroying an email removes it from all mailboxes to which it belonged. To just delete an email to trash, simply change the "mailboxIds" property so it is now in the mailbox with "role == "trash"", and remove all other mailbox ids.

When emptying the trash, clients SHOULD NOT destroy emails which are also in a mailbox other than trash. For those emails, they SHOULD just remove the Trash mailbox from the email.

For successfully created Email objects, the _created_ response contains the _id_, _blobId_, _threadId_ and _size_ properties of the object.

The following extra _SetError_ types are defined:

For *create*:

- o "blobNotFound": At least one blob id given for an EmailBodyPart doesn't exist. An extra _notFound_ property of type "Id[]" MUST be included in the error object containing every _blobId_ referenced by an EmailBodyPart that could not be found on the server.

For **create** and **update**:

- o "tooManyKeywords": The change to the email's keywords would exceed a server-defined maximum.
- o "tooManyMailboxes": The change to the email's mailboxes would exceed a server-defined maximum.

4.7. Email/copy

Standard `/copy` method as described in [I-D.ietf-jmap-core] section 5.4, except only the `_mailboxIds_`, `_keywords_` and `_receivedAt_` properties may be set during the copy. This method cannot modify the RFC5322 representation of an email.

The server MAY forbid two email objects with the same exact [RFC5322] content, or even just with the same [RFC5322] Message-ID, to coexist within an account; if the target account already has the email the copy will be rejected with a standard "alreadyExists" error.

For successfully copied Email objects, the `_created_` response contains the `_id_`, `_blobId_`, `_threadId_` and `_size_` properties of the new object.

4.8. Email/import

The `_Email/import_` method adds [RFC5322] messages to the set of emails in an account. The server MUST support messages with [RFC6532] EAI headers. The messages must first be uploaded as blobs using the standard upload mechanism. It takes the following arguments:

- o **accountId**: "Id" The id of the account to use.
- o **ifInState**: "String|null" This is a state string as returned by the `_Email/get_` method. If supplied, the string must match the current state of the account referenced by the `accountId`, otherwise the method will be aborted and a "stateMismatch" error returned. If "null", any changes will be applied to the current state.
- o **emails**: "Id[EmailImport]" A map of creation id (client specified) to EmailImport objects

An **EmailImport** object has the following properties:

- o **blobId**: "Id" The id of the blob containing the raw [RFC5322] message.

- o `*mailboxIds*`: "Id[Boolean]" The ids of the mailboxes to assign this email to. At least one mailbox MUST be given.
- o `*keywords*`: "String[Boolean]" (default: {}) The keywords to apply to the email.
- o `*receivedAt*`: "UTCDate" (default: time of most recent Received header, or time of import on server if none) The `_receivedAt_` date to set on the email.

Each email to import is considered an atomic unit which may succeed or fail individually. Importing successfully creates a new email object from the data referenced by the `blobId` and applies the given mailboxes, keywords and `receivedAt` date.

The server MAY forbid two email objects with the same exact [RFC5322] content, or even just with the same [RFC5322] Message-ID, to coexist within an account. In this case, it MUST reject attempts to import an email considered a duplicate with an "alreadyExists" SetError. An `_existingId_` property of type "Id" MUST be included on the error object with the id of the existing email. If duplicates are allowed, the newly created Email object MUST have a separate id and independent mutable properties to the existing object.

If the `_blobId_`, `_mailboxIds_`, or `_keywords_` properties are invalid (e.g. missing, wrong type, id not found), the server MUST reject the import with an "invalidProperties" SetError.

If the email cannot be imported because it would take the account over quota, the import should be rejected with an "overQuota" SetError.

If the blob referenced is not a valid [RFC5322] message, the server MAY modify the message to fix errors (such as removing NUL octets or fixing invalid headers). If it does this, the `_blobId_` on the response MUST represent the new representation and therefore be different to the `_blobId_` on the EmailImport object. Alternatively, the server MAY reject the import with an "invalidEmail" SetError.

The response has the following arguments:

- o `*accountId*`: "Id" The id of the account used for this call.
- o `*oldState*`: "String|null" The state string that would have been returned by `_Email/get_` on this account before making the requested changes, or "null" if the server doesn't know what the previous state string was.

- o `*newState*`: "String" The state string that will now be returned by `_Email/get_` on this account.
- o `*created*`: "Id[Email]|null" A map of the creation id to an object containing the `_id_`, `_blobId_`, `_threadId_` and `_size_` properties for each successfully imported Email, or "null" if none.
- o `*notCreated*`: "Id[SetError]|null" A map of creation id to a SetError object for each Email that failed to be created, or "null" if all successful. The possible errors are defined above.

The following additional errors may be returned instead of the `_Email/import_` response:

"stateMismatch": An "ifInState" argument was supplied and it does not match the current state.

4.9. Email/parse

This method allows you to parse blobs as [RFC5322] messages to get Email objects. The server MUST support messages with [RFC6532] EAI headers. This can be used to parse and display attached emails without having to import them as top-level email objects in the mail store in their own right.

The following metadata properties on the Email objects will be "null" if requested:

- o `id`
- o `mailboxIds`
- o `keywords`
- o `receivedAt`

The `_threadId_` property of the Email MAY be present if the server can calculate which thread the Email would be assigned to were it to be imported. Otherwise, this too is "null" if fetched.

The `_Email/parse_` method takes the following arguments:

- o `*accountId*`: "Id" The id of the account to use.
- o `*blobIds*`: "Id[]" The ids of the blobs to parse.
- o `*properties*`: "String[]" If supplied, only the properties listed in the array are returned for each Email object. If omitted,

defaults to: ["messageId", "inReplyTo", "references", "sender", "from", "to", "cc", "bcc", "replyTo", "subject", "sentAt", "hasAttachment", "preview", "bodyValues", "textBody", "htmlBody", "attachments"]

- o `*bodyProperties*`: "String[]" A list of properties to fetch for each EmailBodyPart returned. If omitted, defaults to the same value as the Email/get "bodyProperties" default argument.
- o `*fetchTextBodyValues*`: "Boolean" (default: false) If "true", the `_bodyValues_` property includes any "text/*" part in the "textBody" property.
- o `*fetchHTMLBodyValues*`: "Boolean" (default: false) If "true", the `_bodyValues_` property includes any "text/*" part in the "htmlBody" property.
- o `*fetchAllBodyValues*`: "Boolean" (default: false) If "true", the `_bodyValues_` property includes any "text/*" part in the "bodyStructure" property.
- o `*maxBodyValueBytes*`: "UnsignedInt" (default: 0) If greater than zero, the `_value_` property of any EmailBodyValue object returned in `_bodyValues_` MUST be truncated if necessary so it does not exceed this number of octets in size. If "0" (the default), no truncation occurs. The server MUST ensure the truncation results in valid UTF-8 and does not occur mid-codepoint. If the part is of type "text/html", the server SHOULD NOT truncate inside an HTML tag, e.g. in the middle of "". There is no requirement for the truncated form to be a balanced tree or valid HTML (indeed, the original source may well be neither of these things).

The response has the following arguments:

- o `*accountId*`: "Id" The id of the account used for the call.
- o `*parsed*`: "Id[Email]|null" A map of blob id to parsed Email representation for each successfully parsed blob, or "null" if none.
- o `*notParsable*`: "Id[]|null" A list of ids given that corresponded to blobs that could not be parsed as emails, or "null" if none.
- o `*notFound*`: "Id[]|null" A list of blob ids given that could not be found, or "null" if none.

As specified above, parsed forms of headers may only be used on appropriate header fields. Attempting to fetch a form that is forbidden (e.g. "header:From:asDate") MUST result in the method call being rejected with an "invalidArguments" error.

Where a specific header is requested as a property, the capitalization of the property name in the response MUST be identical to that used in the request.

4.10. Examples

A client logs in for the first time. It first fetches the set of mailboxes. Now it will display the inbox to the user, which we will presume has mailbox id "fb666a55". The inbox may be (very!) large, but the user's screen is only so big, so the client will just load the start and then can load in more as necessary. The client sends this request:

```
[ [ "Email/query", {
  "accountId": "ue150411c",
  "filter": {
    "inMailbox": "fb666a55"
  },
  "sort": [{
    "isAscending": false,
    "property": "receivedAt"
  }],
  "collapseThreads": true,
  "position": 0,
  "limit": 30,
  "calculateTotal": true
}, "0" ],
[ "Email/get", {
  "accountId": "ue150411c",
  "#ids": {
    "resultOf": "0",
    "name": "Email/query",
    "path": "/ids"
  },
  "properties": [
    "threadId"
  ]
}, "1" ],
[ "Thread/get", {
  "accountId": "ue150411c",
  "#ids": {
    "resultOf": "1",
    "name": "Email/get",
```



```

        "path": "/list/*/threadId"
      }
    }, "2" ],
    [ "Email/get", {
      "accountId": "ue150411c",
      "#ids": {
        "resultOf": "2",
        "name": "Thread/get",
        "path": "/list/*/emailIds"
      },
      "properties": [
        "threadId",
        "mailboxIds",
        "keywords",
        "hasAttachment",
        "from",
        "subject",
        "receivedAt",
        "size",
        "preview"
      ]
    }
  ], "3" ]]

```

Let's break down the 4 method calls to see what they're doing:

"0": This asks the server for the ids of the first 30 Email objects in the inbox, sorted newest first, ignoring messages from the same thread as a newer message in the mailbox (i.e. it is the first 30 unique threads).

"1": Now we use a back-reference to fetch the thread ids for each of these email ids.

"2": Another back-reference fetches the Thread object for each of these thread ids.

"3": Finally, we fetch the information we need to display the mailbox listing (but no more!) for every message in each of these 30 threads. The client may aggregate this data for display, for example showing the thread as "flagged" if any of the messages in it contain the "\$flagged" keyword.

The response from the server may look something like this:

```

[[ "Email/query", {
  "accountId": "ue150411c",
  "queryState": "09aa9a075588-780599:0",
  "canCalculateChanges": true,

```

```
"position": 0,
"total": 115,
"ids": [ "Ma783e5cdf5f2deffbc97930a",
        "M9bd17497e2a99cb345fcl0a", ... ]
}, "0" ],
[ "Email/get", {
  "accountId": "ue150411c",
  "state": "780599",
  "list": [{
    "id": "Ma783e5cdf5f2deffbc97930a",
    "threadId": "T36703c2cfe9bd5ed"
  }, {
    "id": "M9bd17497e2a99cb345fcl0a",
    "threadId": "T0a22ad76e9c097a1"
  }, ... ],
  "notFound": []
}, "1" ],
[ "Thread/get", {
  "accountId": "ue150411c",
  "state": "22a8728b",
  "list": [{
    "id": "T36703c2cfe9bd5ed",
    "emailIds": [ "Ma783e5cdf5f2deffbc97930a" ]
  }, {
    "id": "T0a22ad76e9c097a1",
    "emailIds": [ "M3b568670a63e5d100f518fa5",
                  "M9bd17497e2a99cb345fcl0a" ]
  }, ... ],
  "notFound": []
}, "2" ],
[ "Email/get", {
  "accountId": "ue150411c",
  "state": "780599",
  "list": [{
    "id": "Ma783e5cdf5f2deffbc97930a",
    "threadId": "T36703c2cfe9bd5ed",
    "mailboxIds": {
      "fb666a55": true
    },
  },
  "keywords": {
    "$seen": true,
    "$flagged": true
  },
  "hasAttachment": true,
  "from": [{
    "email": "jdoe@example.com",
    "name": "Jane Doe"
  }],
}
```

```
"subject": "The Big Reveal",
"receivedAt": "2018-06-27T00:20:35Z",
"size": 175047,
"preview": "As you may be aware, we are required to prepare a
  presentation where we wow a panel of 5 random members of the
  public, on or before 30 June each year. We have drafted ..."
},
...
],
"notFound": []
}, "3" ]]
```

Now, on another device the user marks the first message as unread, sending this API request:

```
[[ "Email/set", {
  "accountId": "ue150411c",
  "update": {
    "Ma783e5cdf5f2deffbc97930a": {
      "keywords/$seen": null
    }
  }
}, "0" ]]
```

The server applies this and sends the success response:

```
[[ "Email/set", {
  "accountId": "ue150411c",
  "oldState": "780605",
  "newState": "780606",
  "updated": {
    "Ma783e5cdf5f2deffbc97930a": null
  },
  ...
}, "0" ]]
```

The user also deletes a few messages, and then a new message arrives.

Back on our original machine, we receive a push update that the state string for Email is now "780800". As this does not match the client's current state, it issues a request for the changes:

```
[ [ "Email/changes", {
  "accountId": "ue150411c",
  "sinceState": "780605",
  "maxChanges": 50
}, "3" ],
[ "Email/queryChanges", {
  "accountId": "ue150411c",
  "filter": {
    "inMailbox": "fb666a55"
  },
  "sort": [{
    "property": "receivedAt",
    "isAscending": false
  }],
  "collapseThreads": true,
  "sinceQueryState": "09aa9a075588-780599:0",
  "upToId": "Mc2781d5e856a908d8a35a564",
  "maxChanges": 25,
  "calculateTotal": true
}, "11" ]]
```

The response:

```
[ [ "Email/changes", {
  "accountId": "ue150411c",
  "oldState": "780605",
  "newState": "780800",
  "hasMoreChanges": false,
  "created": [ "Me8de6c9f6de198239b982ea2" ],
  "updated": [ "Ma783e5cdf5f2deffbc97930a" ],
  "destroyed": [ "M9bd17497e2a99cb345fc1d0a", ... ]
}, "3" ],
[ "Email/queryChanges", {
  "accountId": "ue150411c",
  "oldQueryState": "09aa9a075588-780599:0",
  "newQueryState": "e35e9facf117-780615:0",
  "added": [{
    "id": "Me8de6c9f6de198239b982ea2",
    "index": 0
  }],
  "removed": [ "M9bd17497e2a99cb345fc1d0a" ],
  "total": 115
}, "11" ]]
```

The client can update its local cache of the query results by removing "M9bd17497e2a99cb345fc1d0a" and then splicing in "Me8de6c9f6de198239b982ea2" at position 0. As it does not have the

data for this new email, it will then fetch it (it also could have done this in the same request using back-references).

It knows something has changed about "Ma783e5cdf5f2deffbc97930a", so it will refetch the mailboxes and keywords (the only mutable properties) for this email too.

The user composes a new message and saves a draft. The client sends:

```
[["Email/set", {
  "accountId": "ue150411c",
  "create": {
    "kl546": {
      "mailboxIds": {
        "2ealca41b38e": true
      },
      "keywords": {
        "$seen": true,
        "$draft": true
      },
    },
    "from": [{
      "name": "Joe Bloggs",
      "email": "joe@example.com"
    }],
    "to": [{
      "name": "John",
      "email": "john@example.com"
    }],
    "subject": "World domination",
    "receivedAt": "2018-07-10T01:05:08Z",
    "sentAt": "2018-07-10T11:05:08+10:00",
    "bodyStructure": {
      "type": "multipart/alternative",
      "subParts": [{
        "partId": "a49d",
        "type": "text/html"
      }, {
        "partId": "bd48",
        "type": "text/plain"
      }]
    },
    "bodyValues": {
      "bd48": {
        "value": "I have the most brilliant plan. Let me tell you
          all about it. What we do is, we",
        "isTruncated": false
      },
      "49db": {
```

```

    "value": "<!DOCTYPE html><html><head><title></title>
      <style type=\"text/css\">div{font-size:16px}</style></head>
      <body><div>I have the most brilliant plan. Let me tell you
        all about it. What we do is, we</div></body></html>",
    "isTruncated": false
  }
}
}
}, "0" ]]

```

The server creates the message and sends the success response:

```

[[ "Email/set", {
  "accountId": "ue150411c",
  "oldState": "780823",
  "newState": "780839",
  "created": {
    "k1546": {
      "id": "Md45b47b4877521042cec0938",
      "blobId": "Ge8de6c9f6de198239b982ea214e0f3a704e4af74",
      "threadId": "Td957e72e89f516dc",
      "size": 11721
    }
  },
  ...
}, "0" ]]

```

The client moves this draft to a different account. The only way to do this is via the `/copy` method. It MUST set a new `mailboxIds` property, since the current value will not be valid mailbox ids in the destination account:

```

[[ "Email/copy", {
  "fromAccountId": "ue150411c",
  "accountId": "u6c6c41ac",
  "create": {
    "k45": {
      "id": "Md45b47b4877521042cec0938",
      "mailboxIds": {
        "75a4c956": true
      }
    }
  },
  "onSuccessDestroyOriginal": true
}, "0" ]]

```

The server successfully copies the email and deletes the original. Due to the implicit call to "Email/set", there are two responses to the single method call, both with the same method call id:

```
[["Email/copy", {
  "fromAccountId": "ue150411c",
  "accountId": "u6c6c41ac",
  "oldState": "7ee7e9263a6d",
  "newState": "5a0d2447ed26",
  "created": {
    "k45": {
      "id": "M138f9954a5cd2423daeafa55",
      "blobId": "G6b9fb047cba722c48c611e79233d057c6b0b74e8",
      "threadId": "T2f242ea424a4079a",
      "size": 11721
    }
  },
  "notCreated": null
}], "0" ],
[ "Email/set", {
  "accountId": "ue150411c",
  "oldState": "780839",
  "newState": "780871",
  "destroyed": [ "Md45b47b4877521042cec0938" ],
  ...
}, "0" ]]
```

5. Search snippets

When doing a search on a "String" property, the client may wish to show the relevant section of the body that matches the search as a preview instead of the beginning of the message, and to highlight any matching terms in both this and the subject of the email. Search snippets represent this data.

A **SearchSnippet** object has the following properties:

- o **emailId**: "Id" The email id the snippet applies to.
- o **subject**: "String|null" If text from the filter matches the subject, this is the subject of the email with the following transformations:
 1. Any instance of the following three characters MUST be replaced by an appropriate [HTML] entity: & (ampersand), < (less-than sign), and > (greater-than sign). Other characters MAY also be replaced with an HTML entity form.

2. The matching words/phrases from the filter are wrapped in HTML "<mark></mark>" tags.

If the subject does not match text from the filter, this property is "null".

- o `*preview*`: "String|null" If text from the filter matches the plain-text or HTML body, this is the relevant section of the body (converted to plain text if originally HTML), with the same transformations as the `_subject_` property. It MUST NOT be bigger than 255 octets in size. If the body does not contain a match for the text from the filter, this property is "null".

It is server-defined what is a relevant section of the body for preview. If the server is unable to determine search snippets, it MUST return "null" for both the `_subject_` and `_preview_` properties.

Note, unlike most data types, a SearchSnippet DOES NOT have a property called "id".

The following JMAP method is supported:

5.1. SearchSnippet/get

To fetch search snippets, make a call to "SearchSnippet/get". It takes the following arguments:

- o `*accountId*`: "Id" The id of the account to use.
- o `*filter*`: "FilterOperator|FilterCondition|null" The same filter as passed to Email/query; see the description of this method in section 4.4 for details.
- o `*emailIds*`: "Id[]" The ids of the emails to fetch snippets for.

The response has the following arguments:

- o `*accountId*`: "Id" The id of the account used for the call.
- o `*list*`: "SearchSnippet[]" An array of SearchSnippet objects for the requested email ids. This may not be in the same order as the ids that were in the request.
- o `*notFound*`: "Id[]|null" An array of email ids requested which could not be found, or "null" if all ids were found.

As the search snippets are derived from the message content and the algorithm for doing so could change over time, fetching the same

snippets a second time MAY return a different result. However, the previous value is not considered incorrect, so there is no state string or update mechanism needed.

The following standard errors may be returned instead of the `_searchSnippets_` response:

"requestTooLarge": The number of `_emailIds_` requested by the client exceeds the maximum number the server is willing to process in a single method call.

"unsupportedFilter": The server is unable to process the given `_filter_` for any reason.

5.2. Example

Here we did an Email/query to search for any email in the account containing the word "foo", now we are fetching the search snippets for some of the ids that were returned in the results:

```
[[ "SearchSnippet/get", {
  "accountId": "ue150411c",
  "filter": {
    "text": "foo"
  },
  "emailIds": [
    "M44200ec123de277c0c1ce69c",
    "M7bcbcb0b58d7729686e83d99",
    "M28d12783a0969584b6deaac0",
    ...
  ]
}, "0" ]]
```

Example response:

```
[["SearchSnippet/get", {
  "accountId": "uel50411c",
  "list": [{
    "emailId": "M44200ec123de277c0c1ce69c",
    "subject": null,
    "preview": null
  }, {
    "emailId": "M7bcbcb0b58d7729686e83d99",
    "subject": "The <mark>Foo</mark>sball competition",
    "preview": "...year the <mark>foo</mark>sball competition will
      be held in the Stadium de ..."
  }, {
    "emailId": "M28d12783a0969584b6deaac0",
    "subject": null,
    "preview": "...the <mark>Foo</mark>/bar method results often
      returns &lt;1 widget rather than the complete..."
  },
  ...
],
  "notFound": null
}, "0" ]]
```

6. Identities

An **Identity** object stores information about an email address (or domain) the user may send from. It has the following properties:

- o **id**: "Id" (immutable; server-set) The id of the identity.
- o **name**: "String" (default: "") The "From" *_name_* the client SHOULD use when creating a new message from this identity.
- o **email**: "String" (immutable) The "From" email address the client MUST use when creating a new message from this identity. If the mailbox part of the address (the section before the "@") is the single character "*" (e.g. "*@example.com") then the client may use any valid address ending in that domain (e.g. "foo@example.com").
- o **replyTo**: "EmailAddress[]|null" (default: null) The Reply-To value the client SHOULD set when creating a new message from this identity.
- o **bcc**: "EmailAddress[]|null" (default: null) The Bcc value the client SHOULD set when creating a new message from this identity.
- o **textSignature**: "String" (default: "") Signature the client SHOULD insert into new plain-text messages that will be sent from

this identity. Clients MAY ignore this and/or combine this with a client-specific signature preference.

- o `*htmlSignature*`: "String" (default: "") Signature the client SHOULD insert into new HTML messages that will be sent from this identity. This text MUST be an HTML snippet to be inserted into the "<body></body>" section of the new email. Clients MAY ignore this and/or combine this with a client-specific signature preference.
- o `*mayDelete*`: "Boolean" (server-set) Is the user allowed to delete this identity? Servers may wish to set this to "false" for the user's username or other default address. Attempts to destroy an identity with "mayDelete: false" will be rejected with a standard "forbidden" SetError.

See the "Addresses" header form description in the Email object for the definition of `_EmailAddress_`.

Multiple identities with the same email address MAY exist, to allow for different settings the user wants to pick between (for example with different names/signatures).

The following JMAP methods are supported:

6.1. Identity/get

Standard `"/get"` method as described in [I-D.ietf-jmap-core] section 5.1. The `_ids_` argument may be "null" to fetch all at once.

6.2. Identity/changes

Standard `"/changes"` method as described in [I-D.ietf-jmap-core] section 5.2.

6.3. Identity/set

Standard `"/set"` method as described in [I-D.ietf-jmap-core] section 5.3. The following extra `_SetError_` types are defined:

For `*create*`:

- o `"forbiddenFrom"`: The user is not allowed to send from the address given as the `_email_` property of the identity.

6.4. Example

Request:

```
[ "Identity/get", {  
  "accountId": "acme"  
}, "0" ]
```

with response:

```
[ "Identity/get", {  
  "accountId": "acme",  
  "state": "99401312ae-11-333",  
  "list": [  
    {  
      "id": "XD-3301-222-11_22AAz",  
      "name": "Joe Bloggs",  
      "email": "joe@example.com",  
      "replyTo": null,  
      "bcc": [{  
        "name": null,  
        "email": "joe+archive@example.com"  
      }],  
      "textSignature": "-- \nJoe Bloggs\nMaster of Email",  
      "htmlSignature": "<div><b>Joe Bloggs</b></div>  
      <div>Master of Email</div>",  
      "mayDelete": false  
    },  
    {  
      "id": "XD-9911312-11_22AAz",  
      "name": "Joe B",  
      "email": "*@example.com",  
      "replyTo": null,  
      "bcc": null,  
      "textSignature": "",  
      "htmlSignature": "",  
      "mayDelete": true  
    }  
  ],  
  "notFound": []  
}, "0" ]
```

7. Email submission

An *EmailSubmission* object represents the submission of an email for delivery to one or more recipients. It has the following properties:

- o **id**: "Id" (immutable; server-set) The id of the email submission.

- o `*identityId*`: "Id" (immutable) The id of the identity to associate with this submission.
- o `*emailId*`: "Id" (immutable) The id of the email to send. The email being sent does not have to be a draft, for example when "redirecting" an existing email to a different address.
- o `*threadId*`: "Id" (immutable; server-set) The thread id of the email to send. This is set by the server to the `_threadId_` property of the email referenced by the `_emailId_`.
- o `*envelope*`: "Envelope|null" (immutable) Information for use when sending via SMTP. An `*Envelope*` object has the following properties:
 - * `*mailFrom*`: "Address" The email address to use as the return address in the SMTP submission, plus any parameters to pass with the MAIL FROM address. The JMAP server MAY allow the address to be the empty string. When a JMAP server performs an SMTP message submission, it MAY use the same id string for the [RFC3461] ENVID parameter and the EmailSubmission object id. Servers that do this MAY replace a client-provided value for ENVID with a server-provided value.
 - * `*rcptTo*`: "Address[]" The email addresses to send the message to, and any RCPT TO parameters to pass with the recipient.

An `*Address*` object has the following properties:

- * `*email*`: "String" The email address being represented by the object. This is a "Mailbox" as used in the Reverse-path or Forward-path of the MAIL FROM or RCPT TO command in [RFC5321].
- * `*parameters*`: "Object|null" Any parameters to send with the email (either mail-parameter or rcpt-parameter as appropriate, as specified in [RFC5321]). If supplied, each key in the object is a parameter name, and the value either the parameter value (type "String") or if the parameter does not take a value then "null". For both name and value, any xtext or unitext encodings are removed ([RFC3461], [RFC6533]) and JSON string encoding applied.

If the `_envelope_` property is "null" or omitted on creation, the server MUST generate this from the referenced email as follows:

- * `*mailFrom*`: The email in the `_Sender_` header, if present, otherwise the `_From_` header, if present, and no parameters. If multiple addresses are present in one of these headers, or

there is more than one `_Sender_/_From_` header, the server SHOULD reject the email as invalid but otherwise MUST take the first address in the last `_Sender_/_From_` header in the [RFC5322] version of the message. If the address found from this is not allowed by the identity associated with this submission, the `_email_` property from the identity MUST be used instead.

- * `*rcptTo*`: The deduplicated set of email addresses from the `_To_`, `_Cc_` and `_Bcc_` headers, if present, with no parameters for any of them.
- o `*sendAt*`: "UTCDate" (immutable; server-set) The date the email was/will be released for delivery. If the client successfully used [RFC4865] FUTURERELEASE with the email, this MUST be the time when the server will release the email; otherwise it MUST be the time the EmailSubmission was created.
- o `*undoStatus*`: "String" This represents whether the submission may be canceled. This is server set and MUST be one of the following values:
 - * `"pending"`: It may be possible to cancel this submission.
 - * `"final"`: The email has been relayed to at least one recipient in a manner that cannot be recalled. It is no longer possible to cancel this submission.
 - * `"canceled"`: The email submission was canceled and will not be delivered to any recipient.

On systems that do not support unsending, the value of this property will always be "final". On systems that do support canceling submission, it will start as "pending", and MAY transition to "final" when the server knows it definitely cannot recall the email, but MAY just remain "pending". If in pending state, a client can attempt to cancel the submission by setting this property to "canceled"; if the update succeeds, the submission was successfully canceled and the email has not been delivered to any of the original recipients.

- o `*deliveryStatus*`: "String[DeliveryStatus]|null" (server-set) This represents the delivery status for each of the email's recipients, if known. This property MAY not be supported by all servers, in which case it will remain "null". Servers that support it SHOULD update the EmailSubmission object each time the status of any of the recipients changes, even if some recipients are still being retried. This value is a map from the email address of each

recipient to a `_DeliveryStatus_` object. A `*DeliveryStatus*` object has the following properties:

- * `*smtpReply*`: "String" The SMTP reply string returned for this recipient when the server last tried to relay the email, or in a later DSN (Delivery Status Notification, as defined in [RFC3464]) response for the email. This SHOULD be the response to the RCPT TO stage, unless this was accepted and the email as a whole rejected at the end of the DATA stage, in which case the DATA stage reply SHOULD be used instead. Multi-line SMTP responses should be concatenated to a single string as follows:
 - + The hyphen following the SMTP code on all but the last line is replaced with a space.
 - + Any prefix in common with the first line is stripped from lines after the first.
 - + CRLF is replaced by a space.

For example:

```
550-5.7.1 Our system has detected that this message is
550 5.7.1 likely spam.
```

would become:

```
550 5.7.1 Our system has detected that this message is likely spam.
```

For emails relayed via an alternative to SMTP, the server MAY generate a synthetic string representing the status instead. If it does this, the string MUST be of the following form:

- + A 3-digit SMTP reply code, as defined in [RFC5321], section 4.2.3.
- + Then a single space character.
- + Then an SMTP Enhanced Mail System Status Code as defined in [RFC3463], with a registry defined in [RFC5248].
- + Then a single space character.
- + Then an implementation-specific information string with a human readable explanation of the response.

- * ***delivered***: "String" Represents whether the email has been successfully delivered to the recipient. This MUST be one of the following values:
 - + **"queued"**: The email is in a local mail queue and status will change once it exits the local mail queues. The `_smtpReply_` property may still change.
 - + **"yes"**: The email was successfully delivered to the mailbox of the recipient. The `_smtpReply_` property is final.
 - + **"no"**: Delivery to the recipient permanently failed. The `_smtpReply_` property is final.
 - + **"unknown"**: The final delivery status is unknown, (e.g. it was relayed to an external machine and no further information is available). The `_smtpReply_` property may still change if a DSN arrives.

Note, successful relaying to an external SMTP server SHOULD NOT be taken as an indication that the email has successfully reached the final mailbox. In this case though, the server may receive a DSN response, if requested. If a DSN is received for the recipient with Action equal to "delivered", as per [RFC3464] section 2.3.3, then the `_delivered_` property SHOULD be set to "yes"; if the Action equals "failed", the property SHOULD be set to "no". Receipt of any other DSN SHOULD NOT affect this property. The server MAY also set this property based on other feedback channels.

- * ***displayed***: "String" Represents whether the email has been displayed to the recipient. This MUST be one of the following values:
 - + **"unknown"**: The display status is unknown. This is the initial value.
 - + **"yes"**: The recipient's system claims the email content has been displayed to the recipient. Note, there is no guarantee that the recipient has noticed, read, or understood the content.

If an MDN is received for this recipient with Disposition-Type (as per [RFC8098] section 3.2.6.2) equal to "displayed", this property SHOULD be set to "yes". The server MAY also set this property based on other feedback channels.

- o ***dsnBlobIds***: "Id[]" (server-set) A list of blob ids for Delivery Status Notifications ([RFC3464]) received for this submission, in order of receipt, oldest first. The blob is the whole MIME message (with a top-level content-type of multipart/report), as received.
- o ***mdnBlobIds***: "Id[]" (server-set) A list of blob ids for Message Disposition Notifications ([RFC8098]) received for this submission, in order of receipt, oldest first. The blob is the whole MIME message (with a top-level content-type of multipart/report), as received.

JMAP servers MAY choose not to expose DSN and MDN responses as Email objects if they correlate to an EmailSubmission object. It SHOULD only do this if it exposes them in the `_dsnBlobIds_` and `_mdnBlobIds_` fields instead, and expects the user to be using clients capable of fetching and displaying delivery status via the EmailSubmission object.

For efficiency, a server MAY destroy EmailSubmission objects a certain amount of time after the email is successfully sent or it has finished retrying sending the email. For very basic SMTP proxies, this MAY be immediately after creation, as it has no way to assign a real id and return the information again if fetched later.

The following JMAP methods are supported:

7.1. EmailSubmission/get

Standard `"/get"` method as described in [I-D.ietf-jmap-core] section 5.1.

7.2. EmailSubmission/changes

Standard `"/changes"` method as described in [I-D.ietf-jmap-core] section 5.2.

7.3. EmailSubmission/query

Standard `"/query"` method as described in [I-D.ietf-jmap-core] section 5.5.

A ***FilterCondition*** object has the following properties, any of which may be omitted:

- o ***identityIds***: "Id[]" The EmailSubmission `_identityId_` property must be in this list to match the condition.

- o `*emailIds*`: "Id[]" The EmailSubmission `_emailId_` property must be in this list to match the condition.
- o `*threadIds*`: "Id[]" The EmailSubmission `_threadId_` property must be in this list to match the condition.
- o `*undoStatus*`: "String" The EmailSubmission `_undoStatus_` property must be identical to the value given to match the condition.
- o `*before*`: "UTCDate" The `_sendAt_` property of the EmailSubmission object must be before this date-time to match the condition.
- o `*after*`: "UTCDate" The `_sendAt_` property of the EmailSubmission object must be the same as or after this date-time to match the condition.

An EmailSubmission object matches the FilterCondition if and only if all of the given conditions match. If zero properties are specified, it is automatically "true" for all objects.

The following EmailSubmission properties MUST be supported for sorting:

- o "emailId"
- o "threadId"
- o "sentAt"

7.4. EmailSubmission/queryChanges

Standard `"/queryChanges"` method as described in [I-D.ietf-jmap-core] section 5.6.

7.5. EmailSubmission/set

Standard `"/set"` method as described in [I-D.ietf-jmap-core] section 5.3, with the following two additional request arguments:

- o `*onSuccessUpdateEmail*`: "Id[PatchObject]|null" A map of `_EmailSubmission id_` to an object containing properties to update on the Email object referenced by the EmailSubmission if the create/update/destroy succeeds. (For references to EmailSubmissions created in the same `"/set"` invocation, this is equivalent to a creation-reference so the id will be the creation id prefixed with a "#".)

- o `*onSuccessDestroyEmail*`: `"Id[]|null"` A list of `_EmailSubmission` `ids_` for which the email with the corresponding `emailId` should be destroyed if the create/update/destroy succeeds. (For references to `EmailSubmission` creations, this is equivalent to a creation-reference so the id will be the creation id prefixed with a `"#"`.)

A single implicit `_Email/set_` call MUST be made after all `EmailSubmissions` cred in the same `"/set"` invocation/update/destroy requests have been processed to perform any changes requested in these two arguments. The response to this MUST be returned after the `_EmailSubmission/set_` response.

An email is sent by creating an `EmailSubmission` object. When processing each create, the server must check that the email is valid, and the user has sufficient authorization to send it. If the creation succeeds, the email will be sent to the recipients given in the envelope `_rcptTo_` parameter. The server MUST remove any `_Bcc_` header present on the email during delivery. The server MAY add or remove other headers from the submitted email, or make further alterations in accordance with the server's policy during delivery.

If the referenced email is destroyed at any point after the `EmailSubmission` object is created, this MUST NOT change the behaviour of the email submission (i.e. it does not cancel a future send). The `_emailId_` and `_threadId_` properties of the submission object remain, but trying to fetch them (with a standard `Email/get` call) will return a `"notFound"` error if the corresponding objects have been destroyed.

Similarly, destroying an `EmailSubmission` object MUST NOT affect the deliveries it represents. It purely removes the record of the email submission. The server MAY automatically destroy `EmailSubmission` objects after a certain time or in response to other triggers, and MAY forbid the client from manually destroying `EmailSubmission` objects.

If the email to be sent is larger than the server supports sending, a standard `"tooLarge"` `SetError` MUST be returned. A `_maxSize_` `"UnsignedInt"` property MUST be present on the `SetError` specifying the maximum size of an email that may be sent, in octets.

If the email or identity id given cannot be found, the submission creation is rejected with a standard `"invalidProperties"` `SetError`.

The following extra `_SetError_` types are defined:

For `*create*`:

- o "invalidEmail" - The email to be sent is invalid in some way. The SetError SHOULD contain a property called `_properties_` of type "String[]" that lists **all** the properties of the email that were invalid.
- o "tooManyRecipients" - The envelope (supplied or generated) has more recipients than the server allows. A `_maxRecipients_` "UnsignedInt" property MUST also be present on the SetError specifying the maximum number of allowed recipients.
- o "noRecipients" - The envelope (supplied or generated) does not have any rcptTo emails.
- o "invalidRecipients" - The `_rcptTo_` property of the envelope (supplied or generated) contains at least one rcptTo value which is not a valid email for sending to. An `_invalidRecipients_` "String[]" property MUST also be present on the SetError, which is a list of the invalid addresses.
- o "forbiddenMailFrom" - The server does not permit the user to send an email with the [RFC5321] envelope From.
- o "forbiddenFrom" - The server does not permit the user to send an email with the [RFC5322] From header of the email to be sent.
- o "forbiddenToSend" - The user does not have permission to send at all right now for some reason. A `_description_` "String" property MAY be present on the SetError object to display to the user why they are not permitted.

For **update**:

- o "cannotUnsend": The client attempted to update the `_undoStatus_` of a valid EmailSubmission object from "pending" to "canceled", but the email cannot be unsent.

7.5.1. Example

The following example presumes a draft of the message to be sent has already been saved, and its Email id is "M7f6ed5bcfd7e2604d1753f6c". This call then sends the email immediately, and if successful removes the draft flag and moves it from the Drafts folder (which has Mailbox id "7cb4e8ee-df87-4757-b9c4-2ealca41b38e") to the Sent folder (which we presume has Mailbox id "73dbcb4b-bffc-48bd-8c2a-a2e91ca672f6").

```
[["EmailSubmission/set", {
  "accountId": "ue411d190",
  "create": {
    "k1490": {
      "identityId": "I64588216",
      "emailId": "M7f6ed5bcfd7e2604d1753f6c",
      "envelope": {
        "mailFrom": {
          "email": "john@example.com",
          "parameters": null
        },
        "rcptTo": [{
          "email": "jane@example.com",
          "parameters": null
        },
        ...
      ]
    }
  }
}],
"onSuccessUpdateEmail": {
  "#k1490": {
    "mailboxIds/7cb4e8ee-df87-4757-b9c4-2ealca41b38e": null,
    "mailboxIds/73dbcb4b-bffc-48bd-8c2a-a2e91ca672f6": true,
    "keywords/$draft": null
  }
}], "0" ]]
```

A successful response might look like this. Note there are two responses due to the implicit Email/set call, but both have the same method call id as they are due to the same call in the request:

```

[[ "EmailSubmission/set", {
  "accountId": "ue411d190",
  "oldState": "012421s6-8nrq-4ps4-n0p4-9330r951ns21",
  "newState": "355421f6-8aed-4cf4-a0c4-7377e951af36",
  "created": {
    "k1490": {
      "id": "ES-3bab7f9a-623e-4acf-99a5-2e67facb02a0"
    }
  }
}, "0" ],
[ "Email/set", {
  "accountId": "ue411d190",
  "oldState": "778193",
  "newState": "778197",
  "updated": {
    "M7f6ed5bcfd7e2604d1753f6c": null
  }
}, "0" ]]

```

Suppose instead an admin has removed sending rights for the user, and so the email submission is rejected with a "forbiddenToSend" error. The description argument of the error is intended for display to the user, so should be localised appropriately. Let's suppose the request was sent with an Accept-Language header like this:

Accept-Language: de;q=0.9,en;q=0.8

The server should attempt to choose the best localisation from those it has available based on the Accept-Language header, as described in [I-D.ietf-jmap-core], section 3.7. If the server has English, French and German translations it would choose German as the preferred language and return a response like this:

```

[[ "EmailSubmission/set", {
  "accountId": "ue411d190",
  "oldState": "012421s6-8nrq-4ps4-n0p4-9330r951ns21",
  "newState": "012421s6-8nrq-4ps4-n0p4-9330r951ns21",
  "notCreated": {
    "k1490": {
      "type": "forbiddenToSend",
      "description": "Verzeihung, wegen verdaechtiger Aktivitaeten Ihres Benutze
rkontos haben wir den Versand von Nachrichten gesperrt. Bitte wenden Sie sich fu
er Hilfe an unser Support Team."
    }
  }
}, "0" ]]

```

8. Vacation response

A vacation response automatically sends a reply to messages sent to a particular account, to inform the original sender that their message may not be read for some time. Automated message sending can produce undesirable behaviour. To avoid this, implementors MUST follow the recommendations set forth in [RFC3834].

The **VacationResponse** object represents the state of vacation-response related settings for an account. It has the following properties:

- o **id**: "Id" (immutable; server-set) The id of the object. There is only ever one vacation response object, and its id is `"singleton"`.
- o **isEnabled**: "Boolean" Should a vacation response be sent if an email arrives between the `_fromDate_` and `_toDate_`?
- o **fromDate**: "UTCDate|null" If `_isEnabled_` is "true" emails that arrive on or after this date-time (but before the `_toDate_` if defined) should receive the user's vacation response. If "null", the vacation response is effective immediately.
- o **toDate**: "UTCDate|null" If `_isEnabled_` is "true", emails that arrive before this date-time (but on or after the `_fromDate_` if defined) should receive the user's vacation response. If "null", the vacation response is effective indefinitely.
- o **subject**: "String|null" The subject that will be used by the message sent in response to emails when the vacation response is enabled. If "null", an appropriate subject SHOULD be set by the server.
- o **textBody**: "String|null" The plain text body to send in response to emails when the vacation response is enabled. If this is "null", when the vacation message is sent a plain-text body part SHOULD be generated from the `_htmlBody_` but the server MAY choose to send the response as HTML only. If both `_textBody_` and `_htmlBody_` are "null", an appropriate default body SHOULD be generated for responses by the server.
- o **htmlBody**: "String|null" The HTML body to send in response to emails when the vacation response is enabled. If this is "null", when the vacation message is sent an HTML body part MAY be generated from the `_textBody_`, or the server MAY choose to send the response as plain-text only.

The following JMAP methods are supported:

8.1. VacationResponse/get

Standard `"/get"` method as described in [I-D.ietf-jmap-core] section 5.1.

There **MUST** only be exactly one VacationResponse object in an account. It **MUST** have the id `"singleton"`.

8.2. VacationResponse/set

Standard `"/set"` method as described in [I-D.ietf-jmap-core] section 5.3.

9. Security considerations

All security considerations of JMAP ([I-D.ietf-jmap-core]) apply to this specification. Additional considerations specific to the data types and functionality introduced by this document are described in the following subsections.

9.1. EmailBodyPart value

Service providers typically perform security filtering on incoming email and it's important that the detection of content-type and charset for the security filter aligns with the heuristics performed by JMAP servers. Servers that apply heuristics to determine the content-type or charset for `_EmailBodyValue_` **SHOULD** document the heuristics and provide a mechanism to turn them off in the event they are misaligned with the security filter used at a particular mailbox host.

Automatic conversion of charsets that allow hidden channels for ASCII text, such as UTF-7, have been problematic for security filters in the past so server implementations can mitigate this risk by having such conversions off-by-default and/or separately configurable.

To allow the client to restrict the volume of data it can receive in response to a request, a maximum length may be requested for the data returned for a textual body part. However, truncating the data may change the semantic meaning, for example truncating a URL changes its location. Servers that scan for links to malicious sites should take care to either ensure truncation is not at a semantically significant point, or to rescan the truncated value for malicious content before returning it.

9.2. HTML email display

HTML message bodies provide richer formatting for emails but present a number of security challenges, especially when embedded in a webmail context in combination with interface HTML. Clients that render HTML email should make careful consideration of the potential risks, including:

- o Embedded JavaScript can rewrite the email to change its content on subsequent opening, allowing users to be misled. In webmail systems, if run in the same origin as the interface it can access and exfiltrate all private data accessible to the user, including all other emails and potentially contacts, calendar events, settings, and credentials. It can also rewrite the interface to undetectably phish passwords. A compromise is likely to be persistent, not just for the duration of page load, due to exfiltration of session credentials or installation of a service worker that can intercept all subsequent network requests (this however would only be possible if blob downloads are also available on the same origin, and the service worker script is attached to the message).
- o HTML documents may load content directly from the internet, rather than just referencing attached resources. For example you may have an "" tag with an external "src" attribute. This may leak to the sender when a message is opened, as well as the IP address of the recipient. Cookies may also be sent and set by the server, allowing tracking between different emails and even website visits and advertising profiles.
- o In webmail systems, CSS can break the layout or create phishing vulnerabilities. For example, the use of "position:fixed" can allow an email to draw content outside of its normal bounds, potentially clickjacking a real interface element.
- o If in a webmail context and not inside a separate frame, any styles defined in CSS rules will apply to interface elements as well if the selector matches, allowing the interface to be modified. Similarly, any interface styles that match elements in the email will alter their appearance, potentially breaking the layout of the email.
- o The link text in HTML has no necessary correlation with the actual target of the link, which can be used to make phishing attacks more convincing.

- o Links opened from an email or embedded external content may leak private info in the "Referer" header sent by default in most systems.
- o Forms can be used to mimic login boxes, providing a potent phishing vector if allowed to submit directly from the email display.

There are a number of ways clients can mitigate these issues, and a defence-in-depth approach that uses a combination of techniques will provide the strongest security.

- o HTML can be filtered before rendering, stripping potentially malicious content. Sanitizing HTML correctly is tricky, and implementers are strongly recommended to use a well-tested library with a carefully vetted whitelist-only approach. New features with unexpected security characteristics may be added to HTML rendering engines in the future; a blacklist approach is likely to result in security issues.

Subtle differences in parsing of HTML can introduce security flaws: to filter with 100% accuracy you need to use the same parser when sanitizing that the HTML rendering engine will use.

- o Encapsulating the message in an "<iframe sandbox>", as defined in [HTML], section 4.7.6, can help mitigate a number of risks. This will:
 - * Disable JavaScript.
 - * Disable form submission.
 - * Prevent drawing outside of its bounds, or conflict with interface CSS.
 - * Establish a unique anonymous origin, separate to the containing origin.
- o A strong Content Security Policy [3] can, among other things, block JavaScript and loading of external content should it manage to evade the filter.
- o The leakage of information in the Referer header can be mitigated with the use of a referrer policy [4].
- o A "crossorigin=anonymous" attribute on tags that load remote content can prevent cookies from being sent.

- o If adding "target=_blank" to open links in new tabs, also add "rel=noopener" to ensure the page that opens cannot change the URL in the original tab to redirect the user to a phishing site.

As highly complex software components, HTML rendering engines increase the attack surface of a client considerably, especially when being used to process untrusted, potentially malicious content. Serious bugs have been found in image decoders, JavaScript engines and HTML parsers in the past, which could lead to full system compromise. Clients using an engine should ensure they get the latest version and continue to incorporate any security patches released by the vendor.

9.3. Multiple part display

Messages may consist of multiple parts to be displayed sequentially as a body. Clients MUST render each part in isolation and MUST NOT concatenate the raw text values to render. Doing so may change the overall semantics of the message. If the client or server is decrypting a PGP or S/MIME encrypted part, concatenating with other parts may leak the decrypted text to an attacker, as described in [EFAIL].

9.4. Email submission

SMTP submission servers [RFC6409] use a number of mechanisms to mitigate damage caused by compromised user accounts and end-user systems including rate limiting, anti-virus/anti-spam filters (mail filters) and other technologies. The technologies work better when they have more information about the client connection. If JMAP email submission is implemented as a proxy to an SMTP Submission server, it is useful to communicate this information from the JMAP proxy to the submission server. The de-facto [XCLIENT] extension to SMTP can be used to do this, but use of an authenticated channel is recommended to limit use of that extension to explicitly authorized proxies.

JMAP servers that proxy to an SMTP Submission server SHOULD allow use of the _submissions_ port [RFC8314]. Implementation of a mechanism similar to SMTP XCLIENT is strongly encouraged. While SASL PLAIN over TLS [RFC4616] is presently the mandatory-to-implement mechanism for interoperability with SMTP submission servers [RFC4954], a JMAP submission proxy SHOULD implement and prefer a stronger mechanism for this use case such as TLS client certificate authentication with SASL EXTERNAL ([RFC4422] appendix A) or SCRAM [RFC7677].

In the event the JMAP server directly relays mail to SMTP servers in other administrative domains, then implementation of the de-facto

[milter] protocol is strongly encouraged to integrate with third-party products that address security issues including anti-virus/anti-spam, reputation protection, compliance archiving, and data loss prevention. Proxying to a local SMTP Submission server may be a simpler way to provide such security services.

9.5. Partial account access

A user may only have permission to access a subset of the data that exists in an account. To avoid leaking unauthorised information, in such a situation the server **MUST** treat any data the user does not have permission to access the same as if it did not exist.

For example, suppose user A has an account with two mailboxes, Inbox and Sent, but only shares the Inbox with user B. In this case, when user B fetches mailboxes for this account, the server **MUST** behave as though the Sent mailbox did not exist. Similarly when querying or fetching Email objects, it **MUST** treat any messages that just belong to the Sent mailbox as though they did not exist. Fetching Thread objects **MUST** only return ids for Email objects the user has permission to access; if none, the Thread again **MUST** be treated the same as if it did not exist.

If the server forbids a single account from having two identical messages, or two messages with the same "Message-Id" header field, a user with write access can use the error returned trying to create/import such a message to detect whether it already exists in an inaccessible portion of the account.

9.6. Permission to send from an address

The email ecosystem has in recent years moved towards associating trust with the From address in the [RFC5322] message, particularly with schemes such as DMARC ([RFC7489]).

The set of Identity objects (see section 6) in an account lets the client know which email addresses the user has permission to send from. Each email submission is associated with an identity, and servers **SHOULD** reject submissions where the "From" header field of the email does not correspond to the associated identity.

The server **MAY** allow an exception to send an exact copy of an existing message received into the mail store to another address (otherwise known as "redirecting" or "bouncing"), although it is **RECOMMENDED** the server limit this to destinations the user has verified they also control.

If the user attempts to create a new Identity, the server MUST reject it with the appropriate error if the user does not have permission to use that email address to send from.

The [RFC5321] SMTP MAIL FROM address is often confused with the [RFC5322] message header. The user generally only ever sees the message header address, and this is the primary one to enforce. However the server MUST also enforce appropriate restrictions on the [RFC5321] MAIL FROM address to stop the user from flooding a 3rd party address with bounces and non-delivery notices.

The JMAP submission model provides separate errors for impermissible addresses in either context.

10. IANA considerations

10.1. JMAP capability registration for "mail"

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

Capability Name: "urn:ietf:params:jmap:mail"

Specification document: this document

Intended use: common

Change Controller: IETF

Security and privacy considerations: this document, section 9

10.2. JMAP capability registration for "submission"

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

Capability Name: "urn:ietf:params:jmap:submission"

Specification document: this document

Intended use: common

Change Controller: IETF

Security and privacy considerations: this document, section 9

10.3. JMAP capability registration for "vacationresponse"

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

Capability Name: "urn:ietf:params:jmap:vacationresponse"

Specification document: this document

Intended use: common

Change Controller: IETF

Security and privacy considerations: this document, section 9

10.4. IMAP and JMAP keywords registry

This document makes two changes to the IMAP keywords registry as defined in [RFC5788].

First, the name of the registry is changed to the "IMAP and JMAP keywords Registry".

Second, a scope column is added to the template and registry indicating whether a keyword applies to IMAP-only, JMAP-only, both, or reserved. All keywords presently in the IMAP keyword registry will be marked with a scope of both. The "reserved" status can be used to prevent future registration of a name that would be confusing if registered. Registration of keywords with scope 'reserved' omit most fields in the registration template (see registration of "\$recent" below for an example); such registrations are intended to be infrequent.

IMAP clients MAY silently ignore any keywords marked JMAP-only or reserved in the event they appear in protocol. JMAP clients MAY silently ignore any keywords marked IMAP-only or reserved in the event they appear in protocol.

New JMAP-only keywords are registered in the following sub-sections. These keywords correspond to IMAP system keywords and are thus not appropriate for use in IMAP. These keywords can not be subsequently registered for use in IMAP except via standards action.

10.4.1. Registration of JMAP keyword '\$draft'

This registers the JMAP-only keyword '\$draft' in the "IMAP and JMAP keywords Registry".

Keyword name: "\$draft"

Scope: JMAP-only

Purpose (description): This is set when the user wants to treat the message as a draft the user is composing. This is the JMAP equivalent of the IMAP \Draft flag.

Private or Shared on a server: BOTH

Is it an advisory keyword or may it cause an automatic action: Automatic. If the account has a mailbox marked with the \Drafts special use [RFC6154], setting this flag MAY cause the message to appear in that mailbox automatically. Certain JMAP computed values such as `_unreadEmails_` will change as a result of changing this flag. In addition, mail clients typically will present draft messages in a composer window rather than a viewer window.

When/by whom the keyword is set/cleared: This is typically set by a JMAP client when referring to a draft message. One model for draft emails would result in clearing this flag in an `EmailSubmission/set` operation with an `onSuccessUpdateEmail` attribute. In a mailstore shared by JMAP and IMAP, this is also set and cleared as necessary so it matches the IMAP \Draft flag.

Related keywords: None

Related IMAP/JMAP Capabilities: SPECIAL-USE [RFC6154]

Security Considerations: A server implementing this keyword as a shared keyword may disclose that a user considers the message a draft message. This information would be exposed to other users with read permission for the mailbox keywords.

Published specification (recommended): this document

Person & email address to contact for further information: (editor-contact-goes-here)

Intended usage: COMMON

Owner/Change controller: IESG

10.4.2. Registration of JMAP keyword '\$seen'

This registers the JMAP-only keyword '\$seen' in the "IMAP and JMAP keywords Registry".

Keyword name: "\$seen"

Scope: JMAP-only

Purpose (description): This is set when the user wants to treat the message as read. This is the JMAP equivalent of the IMAP \Seen flag.

Private or Shared on a server: BOTH

Is it an advisory keyword or may it cause an automatic action: Advisory. However, certain JMAP computed values such as `_unreadEmails_` will change as a result of changing this flag.

When/by whom the keyword is set/cleared: This is set by a JMAP client when it presents the message content to the user; clients often offer an option to clear this flag. In a mailstore shared by JMAP and IMAP, this is also set and cleared as necessary so it matches the IMAP \Seen flag.

Related keywords: None

Related IMAP/JMAP Capabilities: None

Security Considerations: A server implementing this keyword as a shared keyword may disclose that a user considers the message to have been read. This information would be exposed to other users with read permission for the mailbox keywords.

Published specification (recommended): this document

Person & email address to contact for further information: (editor-contact-goes-here)

Intended usage: COMMON

Owner/Change controller: IESG

10.4.3. Registration of JMAP keyword '\$flagged'

This registers the JMAP-only keyword '\$flagged' in the "IMAP and JMAP keywords Registry".

Keyword name: "\$flagged"

Scope: JMAP-only

Purpose (description): This is set when the user wants to treat the message as flagged for urgent/special attention. This is the JMAP equivalent of the IMAP \Flagged flag.

Private or Shared on a server: BOTH

Is it an advisory keyword or may it cause an automatic action:
Automatic. If the account has a mailbox marked with the \Flagged special use [RFC6154], setting this flag MAY cause the message to appear in that mailbox automatically.

When/by whom the keyword is set/cleared: JMAP clients typically allow a user to set/clear this flag as desired. In a mailstore shared by JMAP and IMAP, this is also set and cleared as necessary so it matches the IMAP \Flagged flag.

Related keywords: None

Related IMAP/JMAP Capabilities: SPECIAL-USE [RFC6154]

Security Considerations: A server implementing this keyword as a shared keyword may disclose that a user considers the message as flagged for urgent/special attention. This information would be exposed to other users with read permission for the mailbox keywords.

Published specification (recommended): this document

Person & email address to contact for further information: (editor-contact-goes-here)

Intended usage: COMMON

Owner/Change controller: IESG

10.4.4. Registration of JMAP keyword '\$answered'

This registers the JMAP-only keyword '\$answered' in the "IMAP and JMAP keywords Registry".

Keyword name: "\$answered"

Scope: JMAP-only

Purpose (description): This is set when the message has been answered.

Private or Shared on a server: BOTH

Is it an advisory keyword or may it cause an automatic action:
Advisory.

When/by whom the keyword is set/cleared: JMAP clients typically set this when submitting a reply or answer to the message. It may be set by the EmailSubmission/set operation with an onSuccessUpdateEmail attribute. In a mailstore shared by JMAP and IMAP, this is also set and cleared as necessary so it matches the IMAP \Answered flag.

Related keywords: None

Related IMAP/JMAP Capabilities: None

Security Considerations: A server implementing this keyword as a shared keyword may disclose that a user has replied to a message. This information would be exposed to other users with read permission for the mailbox keywords.

Published specification (recommended): this document

Person & email address to contact for further information: (editor-contact-goes-here)

Intended usage: COMMON

Owner/Change controller: IESG

10.4.5. Registration of '\$recent' keyword

This registers the keyword '\$recent' in the "IMAP and JMAP keywords Registry".

Keyword name: "\$recent"

Scope: reserved

Purpose (description): This keyword is not used to avoid confusion with the IMAP \Recent system flag.

Published specification (recommended): this document

Person & email address to contact for further information: (editor-contact-goes-here)

Owner/Change controller: IESG

10.5. Registration of "inbox" role in

This registers the JMAP-only "inbox" attribute in the "IMAP Mailbox Name Attributes Registry", as established in [RFC8457].

Attribute Name: Inbox

Description: New mail is delivered here by default.

Reference: This document, section 10.5.

Usage Notes: JMAP only

10.6. JMAP Error Codes registry

The following sub-sections register several new error codes in the JMAP Error Codes registry, as defined in [I-D.ietf-jmap-core].

10.6.1. mailboxHasChild

JMAP Error Code: mailboxHasChild

Intended use: common

Change controller: IETF

Reference: This document, section 2.5

Description: The mailbox still has at least one child mailbox. The client MUST remove these before it can delete the parent mailbox.

10.6.2. mailboxHasEmail

JMAP Error Code: mailboxHasEmail

Intended use: common

Change controller: IETF

Reference: This document, section 2.5

Description: The mailbox has at least one message assigned to it and the onDestroyRemoveMessages argument was false.

10.6.3. blobNotFound

JMAP Error Code: blobNotFound

Intended use: common

Change controller: IETF

Reference: This document, section 4.6

Description: At least one blob id referenced in the object doesn't exist.

10.6.4. tooManyKeywords

JMAP Error Code: tooManyKeywords

Intended use: common

Change controller: IETF

Reference: This document, section 4.6

Description: The change to the email's keywords would exceed a server-defined maximum.

10.6.5. tooManyMailboxes

JMAP Error Code: tooManyMailboxes

Intended use: common

Change controller: IETF

Reference: This document, section 4.6

Description: The change to the email's mailboxes would exceed a server-defined maximum.

10.6.6. invalidEmail

JMAP Error Code: invalidEmail

Intended use: common

Change controller: IETF

Reference: This document, section 7.5

Description: The email to be sent is invalid in some way.

10.6.7. tooManyRecipients

JMAP Error Code: tooManyRecipients

Intended use: common

Change controller: IETF

Reference: This document, section 7.5

Description: The RFC5321 envelope (supplied or generated) has more recipients than the server allows.

10.6.8. noRecipients

JMAP Error Code: noRecipients

Intended use: common

Change controller: IETF

Reference: This document, section 7.5

Description: The RFC5321 envelope (supplied or generated) does not have any rcptTo emails.

10.6.9. invalidRecipients

JMAP Error Code: invalidRecipients

Intended use: common

Change controller: IETF

Reference: This document, section 7.5

Description: The rcptTo property of the RFC5321 envelope (supplied or generated) contains at least one rcptTo value which is not a valid email for sending to.

10.6.10. forbiddenMailFrom

JMAP Error Code: forbiddenMailFrom

Intended use: common

Change controller: IETF

Reference: This document, section 7.5

Description: The server does not permit the user to send an email with the RFC5321 envelope From.

10.6.11. forbiddenFrom

JMAP Error Code: forbiddenFrom

Intended use: common

Change controller: IETF

Reference: This document, sections 6.3 and 7.5

Description: The server does not permit the user to send an email with the RFC5322 From header field of the email to be sent.

10.6.12. forbiddenToSend

JMAP Error Code: forbiddenToSend

Intended use: common

Change controller: IETF

Reference: This document, section 7.5

Description: The user does not have permission to send at all right now.

11. References

11.1. Normative References

- [HTML] Faulkner, S., Eicholz, A., Leithead, T., Danilo, A., and S. Moon, "HTML 5.2", 2017, <<https://www.w3.org/TR/html52/>>.
- [I-D.ietf-jmap-core] Jenkins, N. and C. Newman, "JSON Meta Application Protocol", draft-ietf-jmap-core-14 (work in progress), January 2019.
- [RFC1870] Klensin, J., Freed, N., and K. Moore, "SMTP Service Extension for Message Size Declaration", STD 10, RFC 1870, DOI 10.17487/RFC1870, November 1995, <<https://www.rfc-editor.org/info/rfc1870>>.
- [RFC2045] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies", RFC 2045, DOI 10.17487/RFC2045, November 1996, <<https://www.rfc-editor.org/info/rfc2045>>.

- [RFC2047] Moore, K., "MIME (Multipurpose Internet Mail Extensions) Part Three: Message Header Extensions for Non-ASCII Text", RFC 2047, DOI 10.17487/RFC2047, November 1996, <<https://www.rfc-editor.org/info/rfc2047>>.
- [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>>.
- [RFC2231] Freed, N. and K. Moore, "MIME Parameter Value and Encoded Word Extensions: Character Sets, Languages, and Continuations", RFC 2231, DOI 10.17487/RFC2231, November 1997, <<https://www.rfc-editor.org/info/rfc2231>>.
- [RFC2369] Neufeld, G. and J. Baer, "The Use of URLs as Meta-Syntax for Core Mail List Commands and their Transport through Message Header Fields", RFC 2369, DOI 10.17487/RFC2369, July 1998, <<https://www.rfc-editor.org/info/rfc2369>>.
- [RFC2392] Levinson, E., "Content-ID and Message-ID Uniform Resource Locators", RFC 2392, DOI 10.17487/RFC2392, August 1998, <<https://www.rfc-editor.org/info/rfc2392>>.
- [RFC2557] Palme, J., Hopmann, A., and N. Shelness, "MIME Encapsulation of Aggregate Documents, such as HTML (MHTML)", RFC 2557, DOI 10.17487/RFC2557, March 1999, <<https://www.rfc-editor.org/info/rfc2557>>.
- [RFC2852] Newman, D., "Deliver By SMTP Service Extension", RFC 2852, DOI 10.17487/RFC2852, June 2000, <<https://www.rfc-editor.org/info/rfc2852>>.
- [RFC3282] Alvestrand, H., "Content Language Headers", RFC 3282, DOI 10.17487/RFC3282, May 2002, <<https://www.rfc-editor.org/info/rfc3282>>.
- [RFC3461] Moore, K., "Simple Mail Transfer Protocol (SMTP) Service Extension for Delivery Status Notifications (DSNs)", RFC 3461, DOI 10.17487/RFC3461, January 2003, <<https://www.rfc-editor.org/info/rfc3461>>.
- [RFC3463] Vaudreuil, G., "Enhanced Mail System Status Codes", RFC 3463, DOI 10.17487/RFC3463, January 2003, <<https://www.rfc-editor.org/info/rfc3463>>.

- [RFC3464] Moore, K. and G. Vaudreuil, "An Extensible Message Format for Delivery Status Notifications", RFC 3464, DOI 10.17487/RFC3464, January 2003, <<https://www.rfc-editor.org/info/rfc3464>>.
- [RFC3834] Moore, K., "Recommendations for Automatic Responses to Electronic Mail", RFC 3834, DOI 10.17487/RFC3834, August 2004, <<https://www.rfc-editor.org/info/rfc3834>>.
- [RFC4314] Melnikov, A., "IMAP4 Access Control List (ACL) Extension", RFC 4314, DOI 10.17487/RFC4314, December 2005, <<https://www.rfc-editor.org/info/rfc4314>>.
- [RFC4422] Melnikov, A., Ed. and K. Zeilenga, Ed., "Simple Authentication and Security Layer (SASL)", RFC 4422, DOI 10.17487/RFC4422, June 2006, <<https://www.rfc-editor.org/info/rfc4422>>.
- [RFC4616] Zeilenga, K., Ed., "The PLAIN Simple Authentication and Security Layer (SASL) Mechanism", RFC 4616, DOI 10.17487/RFC4616, August 2006, <<https://www.rfc-editor.org/info/rfc4616>>.
- [RFC4865] White, G. and G. Vaudreuil, "SMTP Submission Service Extension for Future Message Release", RFC 4865, DOI 10.17487/RFC4865, May 2007, <<https://www.rfc-editor.org/info/rfc4865>>.
- [RFC4954] Siemborski, R., Ed. and A. Melnikov, Ed., "SMTP Service Extension for Authentication", RFC 4954, DOI 10.17487/RFC4954, July 2007, <<https://www.rfc-editor.org/info/rfc4954>>.
- [RFC5198] Klensin, J. and M. Padlipsky, "Unicode Format for Network Interchange", RFC 5198, DOI 10.17487/RFC5198, March 2008, <<https://www.rfc-editor.org/info/rfc5198>>.
- [RFC5248] Hansen, T. and J. Klensin, "A Registry for SMTP Enhanced Mail System Status Codes", BCP 138, RFC 5248, DOI 10.17487/RFC5248, June 2008, <<https://www.rfc-editor.org/info/rfc5248>>.
- [RFC5256] Crispin, M. and K. Murchison, "Internet Message Access Protocol – SORT and THREAD Extensions", RFC 5256, DOI 10.17487/RFC5256, June 2008, <<https://www.rfc-editor.org/info/rfc5256>>.

- [RFC5321] Klensin, J., "Simple Mail Transfer Protocol", RFC 5321, DOI 10.17487/RFC5321, October 2008, <<https://www.rfc-editor.org/info/rfc5321>>.
- [RFC5322] Resnick, P., Ed., "Internet Message Format", RFC 5322, DOI 10.17487/RFC5322, October 2008, <<https://www.rfc-editor.org/info/rfc5322>>.
- [RFC5788] Melnikov, A. and D. Cridland, "IMAP4 Keyword Registry", RFC 5788, DOI 10.17487/RFC5788, March 2010, <<https://www.rfc-editor.org/info/rfc5788>>.
- [RFC6154] Leiba, B. and J. Nicolson, "IMAP LIST Extension for Special-Use Mailboxes", RFC 6154, DOI 10.17487/RFC6154, March 2011, <<https://www.rfc-editor.org/info/rfc6154>>.
- [RFC6409] Gellens, R. and J. Klensin, "Message Submission for Mail", STD 72, RFC 6409, DOI 10.17487/RFC6409, November 2011, <<https://www.rfc-editor.org/info/rfc6409>>.
- [RFC6532] Yang, A., Steele, S., and N. Freed, "Internationalized Email Headers", RFC 6532, DOI 10.17487/RFC6532, February 2012, <<https://www.rfc-editor.org/info/rfc6532>>.
- [RFC6533] Hansen, T., Ed., Newman, C., and A. Melnikov, "Internationalized Delivery Status and Disposition Notifications", RFC 6533, DOI 10.17487/RFC6533, February 2012, <<https://www.rfc-editor.org/info/rfc6533>>.
- [RFC6710] Melnikov, A. and K. Carlberg, "Simple Mail Transfer Protocol Extension for Message Transfer Priorities", RFC 6710, DOI 10.17487/RFC6710, August 2012, <<https://www.rfc-editor.org/info/rfc6710>>.
- [RFC7677] Hansen, T., "SCRAM-SHA-256 and SCRAM-SHA-256-PLUS Simple Authentication and Security Layer (SASL) Mechanisms", RFC 7677, DOI 10.17487/RFC7677, November 2015, <<https://www.rfc-editor.org/info/rfc7677>>.
- [RFC8098] Hansen, T., Ed. and A. Melnikov, Ed., "Message Disposition Notification", STD 85, RFC 8098, DOI 10.17487/RFC8098, February 2017, <<https://www.rfc-editor.org/info/rfc8098>>.
- [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>>.

- [RFC8314] Moore, K. and C. Newman, "Cleartext Considered Obsolete: Use of Transport Layer Security (TLS) for Email Submission and Access", RFC 8314, DOI 10.17487/RFC8314, January 2018, <<https://www.rfc-editor.org/info/rfc8314>>.
- [RFC8457] Leiba, B., Ed., "IMAP "\$Important" Keyword and "\Important" Special-Use Attribute", RFC 8457, DOI 10.17487/RFC8457, September 2018, <<https://www.rfc-editor.org/info/rfc8457>>.
- [RFC8474] Gondwana, B., Ed., "IMAP Extension for Object Identifiers", RFC 8474, DOI 10.17487/RFC8474, September 2018, <<https://www.rfc-editor.org/info/rfc8474>>.

11.2. Informative References

- [EFAIL] Poddebniak, D., Dresen, C., Mueller, J., Ising, F., Schinzel, S., Friedberger, S., Somorovsky, J., and J. Schwenk, "Efail: Breaking S/MIME and OpenPGP Email Encryption using Exfiltration Channels", 2018, <<https://www.usenix.org/system/files/conference/usenixsecurity18/sec18-poddebniak.pdf>>.
- [milter] Unknown, "Postfix before-queue Milter support", 2019, <http://www.postfix.org/MILTER_README.html>.
- [RFC3501] Crispin, M., "INTERNET MESSAGE ACCESS PROTOCOL - VERSION 4rev1", RFC 3501, DOI 10.17487/RFC3501, March 2003, <<https://www.rfc-editor.org/info/rfc3501>>.
- [RFC7489] Kucherawy, M., Ed. and E. Zwicky, Ed., "Domain-based Message Authentication, Reporting, and Conformance (DMARC)", RFC 7489, DOI 10.17487/RFC7489, March 2015, <<https://www.rfc-editor.org/info/rfc7489>>.
- [XCLIENT] Unknown, "Postfix XCLIENT Howto", 2019, <http://www.postfix.org/XCLIENT_README.html>.

11.3. URIs

- [1] <https://www.iana.org/assignments/imap-mailbox-name-attributes/imap-mailbox-name-attributes.xhtml>
- [2] <https://www.iana.org/assignments/imap-keywords/imap-keywords.xhtml>
- [3] <https://www.w3.org/TR/CSP3/>

[4] <https://www.w3.org/TR/referrer-policy/>

Authors' Addresses

Neil Jenkins
FastMail
PO Box 234, Collins St West
Melbourne VIC 8007
Australia

Email: neilj@fastmailteam.com
URI: <https://www.fastmail.com>

Chris Newman
Oracle
440 E. Huntington Dr., Suite 400
Arcadia CA 91006
United States of America

Email: chris.newman@oracle.com