JMAP
Internet-Draft
Updates: 5788 (if approved)
Intended status: Standards Track
Expires: January 3, 2019

N. Jenkins
FastMail

July 2, 2018

JMAP for Mail
draft-ietf-jmap-mail-06

Abstract

This document specifies a data model for synchronising email data with a server using JMAP.

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 January 3, 2019.

Table of Contents

1. Introduction 3
1.1. Notational conventions 3
1.2. The Date data types 4
1.3. Terminology 4
1.4. Addition to the capabilities object 4
1.5. Push 6
2. Mailboxes 6
2.1. Mailbox/get 9
2.2. Mailbox/changes 9
2.3. Mailbox/query 10
2.4. Mailbox/queryChanges 10
2.5. Mailbox/set 10
2.6. Example 11
3. Threads 15
3.1. Thread/get 17
3.1.1. Example 17
3.2. Thread/changes 17
4. Emails 17
4.1. Properties of the Email object 17
4.1.1. Metadata 19
4.1.2. Header fields 20
4.1.3. Body parts 26
4.2. Email/get 32
4.2.1. Example 34
4.3. Email/changes 35
4.4. Email/query 36
4.4.1. Filtering 36
4.4.2. Sorting 38
4.4.3. Thread collapsing 40
4.4.4. Response 40
4.5. Email/queryChanges 40
4.6. Email/set 40
4.7. Email/import 43
4.8. Email/copy 44
4.9. Email/parse 46
5. Identities 48
5.1. Identity/get 49
5.2. Identity/changes 49
5.3. Identity/set 49
5.4. Example 49
6. Email submission 50
6.1. EmailSubmission/get 55
6.2. EmailSubmission/changes 55
6.3. EmailSubmission/query 55
6.4. EmailSubmission/queryChanges 56
6.5. EmailSubmission/set 56

6.5.1. Example 58
7. Search snippets 59
7.1. SearchSnippet/get 60
7.2. Example 61
8. Vacation response 62
8.1. VacationResponse/get 63
8.2. VacationResponse/set 63
9. Security considerations 63
9.1. EmailBodyPart value 63
9.2. HTML email display 64
9.3. Email submission 66
10. IANA Considerations 67
10.1. JMAP Capability Registration for "mail" 67
10.2. IMAP and JMAP Keywords Registry 67
10.2.1. Registration of JMAP keyword '$draft' 68
10.2.2. Registration of JMAP keyword '$seen' 69
10.2.3. Registration of JMAP keyword '$flagged' 69
10.2.4. Registration of JMAP keyword '$answered' 70
10.2.5. Registration of '$recent' Keyword 71
11. References 72
11.1. Normative References 72
11.2. URIs 75
Author's Address 75

1. Introduction

JMAP <https://tools.ietf.org/html/draft-ietf-jmap-core-05> 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 synchronising mail between a client and a server using JMAP.

1.1. Notational conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

Type signatures, examples and property descriptions in this document follow the conventions established in Section 1.1 of <https://tools.ietf.org/html/draft-ietf-jmap-core-05>.

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

- *sever-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.

- *immutable*: The value MUST NOT change after the object is created.

- *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 Date 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 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"".

1.3. Terminology

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

1.4. Addition to the capabilities object

The capabilities object is returned as part of the standard JMAP Session object; see the JMAP spec. Servers supporting _this_ specification MUST add a property called "urn:ietf:params:jmap:mail" to the capabilities object. The value of this property is an object which MUST contain the following information on server capabilities:

- *maxMailboxesPerEmail*: "Number|null" The maximum number of mailboxes that can be can assigned to a single email. This MUST be an integer >= 1, or "null" for no limit (or rather, the limit is always the number of mailboxes in the account).

- *maxSizeAttachmentsPerEmail*: "Number" The maximum total size of attachments, in octets, allowed for a single email. A server MAY still reject 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 services

marketing and help materials normally tells 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.

- *maxDelayedSend*: "Number" The number in seconds of the maximum delay the server supports in sending (see the EmailSubmission object). This is "0" if the server does not support delayed send.

- *emailsListSortOptions*: "String[]" A list of all the email properties the server supports for sorting by. 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.

- *submissionExtensions*: "String[String[]]" A JMAP implementation that talks to a Submission [RFC6409] server SHOULD have a configuration setting that allows an administrator to expose a new submission EHLO capability in this field. This allows a JMAP server to gain access to a new submission extension without code changes. By default, the JMAP server should show only known safe-to-expose EHLO capabilities in this field, and hide EHLO capabilities that are only relevant to the JMAP server. Each key in the object is the _ehlo-name_, and the value is a list of _ehlo-args_. Examples of safe-to-expose Submission extensions 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>

The server MUST also include the string "urn:ietf:params:jmap:mail" in the _hasDataFor_ property of any account in which the user may use the data types contained in this specification.

1.5. Push

Servers MUST support the standard JMAP push mechanisms to receive notifications when the state changes for any of the types defined in this specification.

In addition, servers MUST support a psuedo-type called "EmailDelivery" in the push mechanisms. 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.

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.

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:

- *id*: "String" (immutable; server-set) The id of the mailbox.
- *name*: "String" User-visible name for the mailbox, e.g. "Inbox". This may be any Net-Unicode string ([RFC5198]) of at least 1 character in length and maximum 255 octets in size. Servers MUST forbid sibling Mailboxes with the same name. Servers MAY reject names that violate server policy (e.g., names containing slash (/) or control characters).
- *parentId*: "String|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.

- *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 may only have a single role, and no two mailboxes in the same account may have the same role. The value MUST be one of the mailbox attribute names listed in the IANA Mailbox Name Attributes Registry [1], as established in [TODO:being established in EXTRA], 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.

- *sortOrder*: "Number" (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 <= 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.

- *totalEmails*: "Number" (server-set) The number of emails in this mailbox.

- *unreadEmails*: "Number" (server-set) The number of emails in this mailbox that have neither the "$seen" keyword nor the "$draft" keyword.

- *totalThreads*: "Number" (server-set) The number of threads where at least one email in the thread is in this mailbox.

- *unreadThreads*: "Number" (server-set) The number of threads where at least one email in the thread has neither the "$seen" keyword nor the "$draft" keyword AND at least one email in the thread is in this mailbox (but see below for special case handling of Trash). Note, the unread email does not need to be the one in this mailbox.

- *myRights*: "MailboxRights" (server-set) The set of rights (ACLs) the user has in relation to this mailbox. 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. If a sub-mailbox is shared but not the parent mailbox, this may be "false". Corresponds to IMAP ACLs "lr".

  * *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".

  * *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 *all* of the mailboxes have this permission. 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 *all* of the mailboxes have this permission. 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".

  * *mayDelete*: "Boolean" The user may delete the mailbox itself. Corresponds to IMAP ACL "x".

  * *maySubmit*: "Boolean" Messages may be submitted directly to this mailbox. Corresponds to IMAP ACL "p".

- *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 themself. 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 the primary account (which is normally the user's own, rather than a shared account).

The Trash mailbox (that is a mailbox with "role == "trash"") MUST be treated specially for the purpose of unread counts:

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 conversation 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.

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. The _ids_ argument may be "null" to fetch all at once.

2.2. Mailbox/changes

Standard _/changes_ method, but with one extra argument to the response:

- *changedProperties*: "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 the rest of the mailboxes state for most use cases changes 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 _changedProperties_ array may be used directly via a result reference in a subsequent Mailbox/get call in a single request.

2.3. Mailbox/query

Standard _/query_ method.

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

- *parentId*: "String|null" The Mailbox _parentId_ property must match the given value exactly.
- *hasRole*: "Boolean" If this is "true", a Mailbox matches if it has a non-"null" value for its _role_ property. If "false", it must has a "null" _role_ value to match.
- *isSubscribed*: "Boolean" The "isSubscribed" property of the mailbox must be identical to the value given to match the condition.

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

The following properties MUST be supported for sorting:

- "sortOrder"
- "name"
- "parent/name": This is a pseudo-property, just for sorting, with the following semantics: if two mailboxes have a common parent, sort them by name. Otherwise, find the nearest ancestors of each that share a common parent and sort by their names instead. (i.e. This sorts the mailbox list in tree order).

2.4. Mailbox/queryChanges

Standard _/queryChanges_ method.

2.5. Mailbox/set

Standard _/set_ method, but with the following additional argument:

- *onDestroyRemoveMessages*: "Boolean" (default: "false") If "false", attempts to destroy a mailbox that still has any 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*:

- "mailboxHasChild": The mailbox still has at least one child mailbox. The client MUST remove these before it can delete the parent mailbox.

- "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": "23cfa8094c0f41e6",
      "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": "674cc24095db49ce",
      "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": "/changedProperties"
    }
  },
  "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,
    "changedProperties": [
      "totalEmails", "unreadEmails",
      "totalThreads", "unreadThreads"
    ],
    "created": [],
    "updated": ["23cfa8094c0f41e6"],
    "destroyed": []
  },
  "0"
],
["Mailbox/get", {
  "accountId": "u33084183",
  "state": "78542",
  "list": [],
  "notFound": []
}, "1"],
["Mailbox/get", {
  "accountId": "u33084183",
  "state": "78542",
  "list": [{
    "id": "23cfa8094c0f41e6",
    "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": {
      "674cc24095db49ce": {
        "name": "Maybe important mail"
      }
    },
    "destroy": [ "23cfa8094c0f41e6" ]
  },
  "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",
    "created": null,
    "notCreated": null,
    "updated": {
        "674cc24095db49ce": null
    },
    "notUpdated": null,
    "destroyed": null,
    "notDestroyed": {
      "23cfa8094c0f41e6": {
        "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*: "String" (immutable) The id of the thread.

o *emailIds*: "String[]" The ids of the emails in the thread, sorted such that:

  * Any email with the "$draft" keyword that has an "In-Reply-To" header is sorted after the _first_ non-draft email in the thread with the corresponding "Message-Id" header, but before any subsequent non-draft emails.

  * Other than that, everything is sorted by the _receivedAt_ date of the email, oldest first.

  * If two emails are identical under the above two conditions, 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.

3.1.1. Example

Request:

```
[ "Thread/get", {
  "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.

4. Emails

The *Email* object is a representation of an [RFC5322] message, which allows clients to avoid the complexities of MIME parsing, transport 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:

- _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, or even videos and sound clips, intended to be displayed inline in the body 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.

- _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.

Due to the number of properties involved, the set of _Email_ properties is specified over the following three sub-sections.

4.1.1. Metadata

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

- *id*: "String" (immutable; server-set) The id of the Email object. Note, this is the JMAP object id, NOT the [RFC5322] Message-ID header field value.

- *blobId*: "String" (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.

- *threadId*: "String" (immutable; server-set) The id of the Thread to which this Email belongs.

- *mailboxIds*: "String[Boolean]" The set of mailbox ids this email belongs to. An email 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".

- *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. 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*: "Number" (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.

4.1.2. Header fields

These 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:

o *Raw* ("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 MAY use heuristics to determine a charset and decode the octets, or MAY 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.

- *Text* ("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. Any [RFC6532] UTF-8 values decoded.

  6. 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:

  * Subject

  * Comment

  * List-Id

  * Any header not defined in [RFC5322] or [RFC2369]

- *Addresses* ("EmailAddress[]") The header is parsed as an "address-list" value, as specified in [RFC5322] section 3.4, into the "EmailAddress[]" type. The *EmailAddress* object has the following properties:

  * *name*: "String|null" The _display-name_ of the [RFC5322] _mailbox_ or _group_, or "null" if none. 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 or trailing white-space is removed.

* *email*: "String|null" The _addr-spec_ of the [RFC5322] _mailbox_, or "null" if a _group_.

Any syntactically correct [RFC2047] encoded sections with a known encoding MUST be decoded, following the same rules as for the _Text_ form. Any [RFC6532] UTF-8 values MUST be decoded. 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). To prevent obviously nonsense behaviour, which can lead to interoperability issues, this form may only be fetched or set for the following header fields:

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

o *MessageIds* ("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:

    * Message-ID

    * In-Reply-To

    * References

    * Resent-Message-ID

    * Any header not defined in [RFC5322] or [RFC2369]

o *Date* ("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:

    * Date

    * Resent-Date

    * Any header not defined in [RFC5322] or [RFC2369]

o *URLs* ("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:

    * List-Help

    * List-Unsubscribe

    * List-Subscribe

    * List-Post

    * List-Owner

    * List-Archive

    * Any header not defined in [RFC5322] or [RFC2369]

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

- *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:

- *: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.

- *: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:

- *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.

- *inReplyTo*: "String[]|null" (immutable) The value is identical to the value of _header:In-Reply-To:asMessageIds_.

- *references*: "String[]|null" (immutable) The value is identical to the value of _header:References:asMessageIds_.

- *sender*: "EmailAddress[]|null" (immutable) The value is identical to the value of _header:Sender:asAddresses_.

- *from*: "EmailAddress[]|null" (immutable) The value is identical to the value of _header:From:asAddresses_.

- *to*: "EmailAddress[]|null" (immutable) The value is identical to the value of _header:To:asAddresses_.

- *cc*: "EmailAddress[]|null" (immutable) The value is identical to the value of _header:Cc:asAddresses_.

- *bcc*: "EmailAddress[]|null" (immutable) The value is identical to the value of _header:Bcc:asAddresses_.

- *replyTo*: "EmailAddress[]|null" (immutable) The value is identical to the value of _header:Reply-To:asAddresses_.

- *subject*: "String|null" (immutable) The value is identical to the value of _header:Subject:asText_.

- *sentAt*: "Date|null" (immutable; default on creation: current server time) The value is identical to the value of _header:Date:asDate_.

4.1.3. Body parts

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

A *EmailBodyPart* object has the following properties:

- *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/*".

- *blobId*: "String|null" The id representing the raw octets of the contents of the part after decoding any _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 same 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.

- *size*: "Number" 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).

- *headers*: "EmailHeader[]" This is a list of all header fields in the part, in the order they appear. The values are in _Raw_ form.

- *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.

- *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.

- *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 has no charset parameter. If there is no _Content-Type_ header field, this is the implicit charset as per the MIME standard ("us-ascii").

- *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.

- *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.

- *language*: "String[]|null" The list of language tags, as defined in [RFC3282], in the _Content-Language_ header field of the part, if present.

- *location*: "String|null" The URI, as defined in [RFC2557], in the _Content-Location_ header field of the part, if present.

- *subParts*: "EmailBodyPart[]" (optional) 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:

- *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/*".

- *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-Transport-Encoding_ and decoding the _Content-Type_ charset, if 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.

  * *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.

- *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.

- *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.

- *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_ or by referencing the _Content-Location_.

- *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 if either:

  * 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.

- *preview*: "String" (immutable; server-set) Up to 255 octets of plain text, summarising 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, for example skipping quoted sections and salutations and collapsing white-space can result in a more useful preview.

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, with the following additional arguments:

- *bodyProperties*: "String[]" (optional) 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"]

- *fetchTextBodyValues*: "Boolean" (default: "false") If "true", the _bodyValues_ property includes any "text/*" part in the "textBody" property.

- *fetchHTMLBodyValues*: "Boolean" (default: "false") If "true", the _bodyValues_ property includes any "text/*" part in the "htmlBody" property.

- *fetchAllBodyValues*: "Boolean" (default: "false") If "true", the _bodyValues_ property includes any "text/*" part in the "bodyStructure" property.

- *maxBodyValueBytes*: "Number" (optional) 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. When given, 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. 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 "<a href="https://example.com">". 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:

- id

- blobId
- threadId
- mailboxIds
- keywords
- size
- receivedAt
- messageId
- inReplyTo
- sender
- from
- to
- cc
- bcc
- replyTo
- subject
- sentAt
- hasAttachment
- 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@bloggs.com"}],
      "subject": "Dinner on Thursday?",
      "receivedAt": "2013-10-13T14:12:00Z",
      "header:List-POST:asURLs": [ "mailto:partytime@lists.example.com" ],
      "htmlBody": [{
        "partId": "1",
        "blobId": "841623871",
        "size": 283331,
        "type": "text/html"
      }, {
        "partId": "2",
        "blobId": "319437193",
        "size": 10343,
        "type": "text/plain"
      }],
      "bodyValues": {
        "1": {
          "isEncodingProblem": false,
          "isTruncated": true,
          "value": "<html><body><p>Hello ..."
        },
        "2": {
          "isEncodingProblem": false,
          "isTruncated": false,
          "value": "-- \nSent by your friendly mailing list ..."
        }
      }
    }
  ],
  notFound: [ "f123u456" ]
}, "#1"]
```

4.3. Email/changes

Standard _/changes_ method.

4.4. Email/query

Standard _/query_ method, but with the following additional arguments:

- *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 at most only one email will be included in the list for any given thread.

4.4.1. Filtering

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

- *inMailbox*: "String" A mailbox id. An email must be in this mailbox to match the condition.

- *inMailboxOtherThan*: "String[]" 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.

- *before*: "UTCDate" The _receivedAt_ date of the email must be before this date to match the condition.

- *after*: "UTCDate" The _receivedAt_ date of the email must be on or after this date to match the condition.

- *minSize*: "Number" The _size_ of the email in octets must be equal to or greater than this number to match the condition.

- *maxSize*: "Number" The size of the email in octets must be less than this number to match the condition.

- *allInThreadHaveKeyword*: "String" All emails (including this one) in the same thread as this email must have the given keyword to match the condition.

- *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.

- *noneInThreadHaveKeyword*: "String" All emails (including this one) in the same thread as this email must *not* have the given keyword to match the condition.

- *hasKeyword*: "String" This email must have the given keyword to match the condition.

- *notKeyword*: "String" This email must not have the given keyword to match the condition.

- *hasAttachment*: "Boolean" The "hasAttachment" property of the email must be identical to the value given to match the condition.

- *text*: "String" Looks for the text in emails. The server SHOULD look up text in the _from_, _to_, _cc_, _bcc_, _subject_ header fields of the message, and 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.

- *from*: "String" Looks for the text in the _From_ header field of the message.

- *to*: "String" Looks for the text in the _To_ header field of the message.

- *cc*: "String" Looks for the text in the _Cc_ header field of the message.

- *bcc*: "String" Looks for the text in the _Bcc_ header field of the message.

- *subject*: "String" Looks for the text in the _subject_ property of the email.

- *body*: "String" Looks for the text in one of the "text/*" body parts of the email.

- *attachments*: "String" Looks for the text in the attachments of the email. Servers MAY handle text extraction when possible for the different kinds of media.

- *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:

- Any syntactically correct [RFC2047] encoded sections of header fields with a known encoding SHOULD be decoded before attempting to match text.

- 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.

- Text SHOULD be matched in a case-insensitive manner.

- 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.

- 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.

- Tokens 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 properties MUST be supported for sorting:

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

The following properties SHOULD be supported for sorting:

- *size* - The size as returned in the Email object.

- *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.

- *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.

- *subject* - This is taken to be the base subject of the email, as defined in section 2.1 of [RFC5256].

- *sentAt* - The _sentAt_ property on the Email object.

- *hasKeyword* - This value MUST be considered "true" if the email has the keyword given as the _keyword_ property on this _Comparator_ object, or "false" otherwise.

- *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 the _keyword_ property on this _Comparator_ object.

- *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 the _keyword_ property on this _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).

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 == 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 "threadIds" list of the result, at the position of the first email in the list that belongs to the thread.

4.4.4. Response

The response has the following additional argument:

- *collapseThreads*: "Boolean" The _collapseThreads_ value that was used when calculating the email list for this call.

4.5. Email/queryChanges

Standard _/queryChanges_ method, with the following additional arguments:

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

The response has the following additional argument:

- *collapseThreads*: "Boolean" The _collapseThreads_ value that was used when calculating the email list for this call.

4.6. Email/set

Standard _/set_ method. The _Email/set_ method encompasses:

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

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:

- 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.

- There MUST NOT be two properties that represent the same header field (e.g. "header:from" and "from") within the Email or particular EmailBodyPart.

- Header fields MUST NOT be specified in parsed forms that are forbidden for that particular field.

- Header fields beginning "Content-" MUST NOT be specified on the Email object, only on EmailBodyPart objects.

- If a bodyStructure property is given, there MUST NOT be textBody, htmlBody or attachments properties.

- If given, the bodyStructure EmailBodyPart MUST NOT contain a property representing a header field that is already defined on the top-level Email object.

- If given, textBody MUST contain exactly one body part, of type "text/plain".

- If given, htmlBody MUST contain exactly one body part, of type "text/html".

- 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 omitted, but otherwise MUST match the size of the blob.

  * A "Content-Transfer-Encoding" header field MUST NOT be given.

- 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 data that does not currently 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 MUST contain the _id_, _blobId_, _threadId_ and _size_ properties of the object.

The following extra _SetError_ types are defined:

For *create*:

- "blobNotFound": At least one blob id given for an EmailBodyPart doesn't exist. An extra _notFound_ property of type "String[]" 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*:

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

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

4.7. Email/import

The _Email/import_ method adds [RFC5322] messages to a user's set of emails. The messages must first be uploaded as a file using the standard upload mechanism. It takes the following arguments:

- *accountId*: "String|null" The id of the account to use for this call. If "null", defaults to the "urn:ietf:params:jmap:mail" primary account.

- *emails*: "String[EmailImport]" A map of creation id (client specified) to EmailImport objects

An *EmailImport* object has the following properties:

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

- *mailboxIds* "String[Boolean]" The ids of the mailboxes to assign this email to. At least one mailbox MUST be given.

- *keywords*: "String[Boolean]" (default: "{}") The keywords to apply to the email.

- *receivedAt*: "UTCDate" (default: time of import on server) 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 reference 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 _emailId_ property of type "String" MUST be included on the error object with the id of the existing email.

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 a "maxQuotaReached" 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:

- *accountId*: "String" The id of the account used for this call.
- *created*: "String[Email]" A map of the creation id to an object containing the _id_, _blobId_, _threadId_ and _size_ properties for each successfully imported Email.
- *notCreated*: "String[SetError]" A map of creation id to a SetError object for each Email that failed to be created. The possible errors are defined above.

4.8. Email/copy

The only way to move messages *between* two different accounts is to copy them using the _Email/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 _Email/copy_ method takes the following arguments:

- *fromAccountId*: "String|null" The id of the account to copy emails from. If "null", defaults to the "urn:ietf:params:jmap:mail" primary account.
- *toAccountId*: "String|null" The id of the account to copy emails to. If "null", defaults to the "urn:ietf:params:jmap:mail" primary account.
- *create*: "String[EmailCopy]" A map of _creation id_ to an EmailCopy object.
- *onSuccessDestroyOriginal*: "Boolean" (default: "false") If "true", an attempt will be made to destroy the emails that were successfully copied: after emitting the _Email/copy_ response, but before processing the next method, the server MUST make a single call to _Email/set_ to destroy the original of each successfully

copied message; the output of this is added to the responses as normal to be returned to the client.

An *EmailCopy* object has the following properties:

- *id*: "String" The id of the email to be copied in the "from" account.
- *mailboxIds*: "String[Boolean]" The ids of the mailboxes (in the "to" account) to add the copied email to. At least one mailbox MUST be given.
- *keywords*: "String[Boolean]" (default: "{}") The _keywords_ property for the copy.
- *receivedAt*: "UTCDate" (default: _receivedAt_ date of original) The _receivedAt_ date to set on the copy.

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 duplicates are allowed though, the "from" account may be the same as the "to" account to copy emails within an account.

Each email copy is considered an atomic unit which may succeed or fail individually. Copying successfully MUST create a new email object, with separate ids and mutable properties (e.g. mailboxes and keywords) to the original email.

The response has the following arguments:

- *fromAccountId*: "String" The id of the account emails were copied from.
- *toAccountId*: "String" The id of the account emails were copied to.
- *created*: "String[Email]|null" A map of the creation id to an object containing the _id_, _blobId_, _threadId_ and _size_ properties for each successfully copied Email.
- *notCreated*: "String[SetError]|null" A map of creation id to a SetError object for each Email 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_. The following extra _SetError_ type is also defined:

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

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

"fromAccountNotFound": A _fromAccountId_ was explicitly included with the request, but it does not correspond to a valid account; or, _fromAccountId_ was null but there is no primary account for "urn:ietf:params:jmap:mail".

"toAccountNotFound": A _toAccountId_ was explicitly included with the request, but it does not correspond to a valid account; or, _toAccountId_ was null but there is no primary account for "urn:ietf:params:jmap:mail".

"fromAccountNotSupportedByMethod": The _fromAccountId_ given corresponds to a valid account, but does not contain any mail data.

"toAccountNotSupportedByMethod": The _toAccountId_ given corresponds to a valid account, but does not contain any mail data.

4.9. Email/parse

This method allows you to parse blobs as [RFC5322] messages to get Email objects. The following metadata properties on the Email objects will be "null" if requested:

- id
- mailboxIds
- keywords
- 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:

- *accountId*: "String|null" The id of the Account to use. If "null", the primary account is used.
- *blobIds*: "String[]" The ids of the blobs to parse.

- *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"]

- *bodyProperties*: "String[]" (optional) A list of properties to fetch for each EmailBodyPart returned. If omitted, defaults to the same value as the Email/get "bodyProperties" default argument.

- *fetchTextBodyValues*: "Boolean" (default: "false") If "true", the _bodyValues_ property includes any "text/*" part in the "textBody" property.

- *fetchHTMLBodyValues*: "Boolean" (default: "false") If "true", the _bodyValues_ property includes any "text/*" part in the "htmlBody" property.

- *fetchAllBodyValues*: "Boolean" (default: "false") If "true", the _bodyValues_ property includes any "text/*" part in the "bodyStructure" property.

- *maxBodyValueBytes*: "Number" (optional) 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. When given, 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. 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.

The response has the following arguments:

- *accountId*: "String" The id of the account used for the call.

- *parsed*: "String[Email]|null" A map of blob id to parsed Email representation for each successfully parsed blob, or "null" if none.

- *notParsable*: "String[]|null" A list of ids given that corresponded to blobs that could not be parsed as emails, or "null" if none.

- *notFound*: "String[]|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.

5. Identities

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

- *id*: "String" (immutable; server-set) The id of the identity.
- *name*: "String" (default: """) The "From" _name_ the client SHOULD use when creating a new message from this identity.
- *email*: "String" (immutable) The "From" email address the client MUST use when creating a new message from this identity. The value MAY alternatively be of the form "*@example.com", in which case the client may use any valid email address ending in "@example.com".
- *replyTo*: "EmailAddress[]|null" (default: "null") The Reply-To value the client SHOULD set when creating a new message from this identity.
- *bcc*: "EmailAddress[]|null" (default: "null") The Bcc value the client SHOULD set when creating a new message from this identity.
- *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.
- *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.
- *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.

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:

5.1. Identity/get

Standard _/get_ method. The _ids_ argument may be "null" to fetch all at once.

5.2. Identity/changes

Standard _/changes_ method.

5.3. Identity/set

Standard _/set_ method. The following extra _SetError_ types are defined:

For *create*:

- "maxQuotaReached": The user has reached a server-defined limit on the number of identities.
- "emailNotPermitted": The user is not allowed to send from the address given as the _email_ property of the identity.

For *destroy*:

- "forbidden": Returned if the identity's _mayDelete_ value is "false".

5.4. Example

Request:

```
[ "Identity/get", {}, "0" ]
```

with response:

```
[ "Identity/get", {
  "accountId": "acme",
  "state": "99401312ae-11-333",
  "list": [
    {
      "id": "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": "9911312-11_22AAz",
      "name": "Joe B",
      "email": "joebloggs@example.com",
      "replyTo": null,
      "bcc": null,
      "textSignature": "",
      "htmlSignature": "",
      "mayDelete": true
    }
  ],
  "notFound": []
}, "0" ]
```

6. Email submission

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

- *id*: "String" (immutable; server-set) The id of the email submission.

- *identityId*: "String" (immutable) The id of the identity to associate with this submission.

- *emailId*: "String" (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.

- *threadId*: "String" (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_.

- *envelope*: "Envelope|null" (immutable; default: "null") 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 as 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.

- *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.

- *undoStatus*: "String" (server-set) 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.

- *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 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, sorry.
```

would become:

550 5.7.1 Our system has detected that this message is likely spam, sorry.

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 [RFC3798] 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*: "String[]" (server-set) A list of blob ids for DSNs received for this submission, in order of receipt, oldest first.

o *mdnBlobIds*: "String[]" (server-set) A list of blob ids for MDNs received for this submission, in order of receipt, oldest first.

JMAP servers MAY choose not to expose DSN and MDN responses as Email objects if they correlate to a 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:

6.1. EmailSubmission/get

Standard _/get_ method.

6.2. EmailSubmission/changes

Standard _/changes_ method.

6.3. EmailSubmission/query

Standard _/query_ method.

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

- *emailIds*: "String[]" The EmailSubmission _emailId_ property must be in this list to match the condition.
- *threadIds*: "String[]" The EmailSubmission _threadId_ property must be in this list to match the condition.
- *undoStatus*: "String" The EmailSubmission _undoStatus_ property must be identical to the value given to match the condition.
- *before*: "UTCDate" The _sendAt_ property of the EmailSubmission object must be before this date to match the condition.
- *after*: "UTCDate" The _sendAt_ property of the EmailSubmission object must be after this date to match the condition.

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

The following properties MUST be supported for sorting:

- "emailId"
- "threadId"

- "sentAt"

6.4. EmailSubmission/queryChanges

Standard _/queryChanges_ method.

6.5. EmailSubmission/set

Standard _/set_ method, with the following two extra arguments:

- *onSuccessUpdateEmail*: "String[Email]|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 EmailSubmission creations, this is equivalent to a back reference so the id will be the creation id prefixed with a "#".)

- *onSuccessDestroyEmail*: "String[]|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 back reference so the id will be the creation id prefixed with a "#".)

A single implicit _Email/set_ call MUST be made after all EmailSubmission create/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 a 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).

Similarly, destroying a 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.

The following extra _SetError_ types are defined:

For *create*:

- "tooLarge" - The email size is larger than the server supports sending. A _maxSize_ "Number" property MUST be present on the SetError specifying the maximum size of an email that may be sent, in octets.

- "tooManyRecipients" - The envelope (supplied or generated) has more recipients than the server allows. A _maxRecipients_ "Number" property MUST be present on the SetError specifying the maximum number of allowed recipients.

- "noRecipients" - The envelope (supplied or generated) does not have any rcptTo emails.

- "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 be present on the SetError, which is a list of the invalid addresses.

- "forbiddenMailFrom" - The server does not permit the user to send an email with the [RFC5321] envelope From.

- "forbiddenFrom" - The server does not permit the user to send an email with the [RFC5322] From header of the email to be sent.

- "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. The server MAY choose to localise this string into the user's preferred language, if known.

- "emailNotFound" - The _emailId_ is not a valid id for an email in the account.

- "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.

For *update*:

- "cannotUnsend": The client attempted to update the _undoStatus_ of a valid EmailSubmission object from "pending" to "canceled", but the email cannot be unsent.

6.5.1. 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-b9c4-2ea1ca41b38e") to the Sent folder (which we presume has Mailbox id "73dbcb4b-bffc-48bd-8c2a-a2e91ca672f6").

```
[
  "EmailSubmission/set",
  {
    "accountId": "ue411d190",
    "create": {
      "k1490": {
        "identityId": "64588216",
        "emailId": "M7f6ed5bcfd7e2604d1753f6c",
        "envelope": {
          "mailFrom": {
            "email": "john@example.com",
            "parameters": null
          },
          "rcptTo": [
            {
              "email": "jane@example.com",
              "parameters": null
            }
          ]
        }
      }
    },
    "onSuccessUpdateEmail": {
      "#k1490": {
        "mailboxIds/7cb4e8ee-df87-4757-b9c4-2ea1ca41b38e": 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 tag 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": "3bab7f9a-623e-4acf-99a5-2e67facb02a0"
      }
    },
    "notCreated": null,
    "updated": null,
    "notUpdated": null,
    "destroyed": null,
    "notDestroyed": null
  },
  "0"
],
[
  "Email/set",
  {
    "accountId": "neilj@fastmail.fm",
    "oldState": "778193",
    "newState": "778197",
    "created": null,
    "notCreated": null,
    "updated": {
        "M7f6ed5bcfd7e2604d1753f6c": null
    },
    "notUpdated": null,
    "destroyed": null,
    "notDestroyed": null
  },
  "0"
]
```

7. 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*: "String" The email id the snippet applies to.

- *subject*: "String|null" If text from the filter matches the subject, this is the subject of the email HTML-escaped, with matching words/phrases wrapped in "<mark></mark>" tags. If it does not match, this is "null".

- *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), HTML-escaped, with matching words/phrases wrapped in "<mark></mark>" tags. It MUST NOT be bigger than 255 octets in size. If it does not match, this is "null".

- *attachments*: "String|null" If text from the filter matches the text extracted from an attachment, this is the relevant section of the attachment (converted to plain text), with matching words/phrases wrapped in "<mark></mark>" tags. It MUST NOT be bigger than 255 octets in size. If it does not match, this 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_, _preview_ and _attachments_ properties.

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

The following JMAP method is supported:

7.1. SearchSnippet/get

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

- *accountId*: "String|null" The id of the account to use for this call. If "null", defaults to the "urn:ietf:params:jmap:mail" primary account.

- *filter*: "FilterOperator|FilterCondition|null" The same filter as passed to Email/query; see the description of this method for details.

- *emailIds*: "String[]" The list of ids of emails to fetch the snippets for.

The response has the following arguments:

- *accountId*: "String" The id of the account used for the call.

- *filter*: "FilterOperator|FilterCondition|null" Echoed back from the call.

- *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.

- *notFound*: "String[]|null" An array of email ids requested which could not be found, or "null" if all ids were found.

Since snippets are only based on immutable properties, there is no state string or update mechanism needed.

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

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

"unsupportedFilter": Returned if the server is unable to process the given _filter_ for any reason.

7.2. 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",
      ...
    ]
  },
  "tag-0"
]
```

Example response:

```
[
  "SearchSnippet/get", {
    "accountId": "ue150411c",
    "filter": {
      "text": "foo"
    },
    "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": "...mail <mark>Foo</mark>/changes results often return
current-state-minus-1 rather than new..."
    },
    ...
    ],
    "notFound": null
  },
  "0"
]
```

8. Vacation response

The *VacationResponse* object represents the state of vacation-response related settings for an account. It has the following properties:

- *id*: "String" (immutable) The id of the object. There is only ever one vacation response object, and its id is ""singleton"".
- *isEnabled* "Boolean" Should a vacation response be sent if an email arrives between the _fromDate_ and _toDate_?
- *fromDate*: "UTCDate|null" If _isEnabled_ is "true", the date/time in UTC after which emails that arrive should receive the user's vacation response. If "null", the vacation response is effective immediately.
- *toDate*: "UTCDate|null" If _isEnabled_ is "true", the date/time in UTC after which emails that arrive should no longer receive the user's vacation response. If "null", the vacation response is

effective indefinitely.

- *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.

- *textBody*: "String|null" The plain text part of the message 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.

- *htmlBody*: "String|null" The HTML message 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.

There MUST only be exactly one VacationResponse object in an account. It MUST have the id ""singleton"".

8.2. VacationResponse/set

Standard _/set_ method.

9. Security considerations

All security considerations of JMAP {TODO: insert RFC ref} apply to this specification.

9.1. EmailBodyPart value

Service providers typically perform security filtering on incoming email and it's important 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:

- Embedded JavaScript can rewrite the email to change its content on subsequent opening, allowing users to be mislead. 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).

- HTML documents may load content directly from the internet, rather than just referencing attached resources. For example you may have an "<img>" 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.

- 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.

- 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.

- 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.

- Links opened from an email or embedded external content may leak private info in the "Referer" header sent by default in most systems.

- 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.

- 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% accurately you need to use the same parser when sanitizing that the HTML rendering engine will use.

- Encapsulating the message in an "<iframe sandbox>" 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.

- A strong Content Security Policy [3] can, among other things, block JavaScript and loading of external content should it manage to evade the filter.

- The leakage of information in the Referer header can be mitigated with the use of a referrer policy [4].

- A "crossorigin=anonymous" attribute on tags that load remote content can prevent cookies from being sent.

- 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. 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 milters 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] and SHOULD implement SASL PLAIN over TLS [RFC4616] and/or TLS client certificate authentication with SASL EXTERNAL [RFC4422] appendix A. Implementation of a mechanism similar to SMTP XCLIENT is strongly encouraged.

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.

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. 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 example for "$recent" subsection of this section); 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.

10.2.1. 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

10.2.2. 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

10.2.3. 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

10.2.4. 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 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

10.2.5. 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

11. References

11.1. Normative References

[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>.

[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>.

[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>.

[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>.

[RFC3798] Hansen, T., Ed. and G. Vaudreuil, Ed., "Message Disposition Notification", RFC 3798, DOI 10.17487/RFC3798, May 2004, <https://www.rfc-editor.org/info/rfc3798>.

[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>.

[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>.

[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>.

11.2. URIs

[1] TODO

[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/

Author's Address

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

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