Internet Engineering Task Force (IETF)
Request for Comments: 8621
Updates: 5788
Category: Standards Track
ISSN: 2070-1721

N. Jenkins
Fastmail
C. Newman
Oracle

August 2019

# The JSON Meta Application Protocol (JMAP) for Mail

## Abstract

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

## Status of This Memo

This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in Section 2 of RFC 7841.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at https://www.rfc-editor.org/info/rfc8621.

Table of Contents

1. Introduction . . . . . 4
1.1. Notational Conventions . . . . . 4
1.2. Terminology . . . . . 5
1.3. Additions to the Capabilities Object . . . . . 5
1.3.1. urn:ietf:params:jmap:mail . . . . . 5
1.3.2. urn:ietf:params:jmap:submission . . . . . 7
1.3.3. urn:ietf:params:jmap:vacationresponse . . . . . 8
1.4. Data Type Support in Different Accounts . . . . . 8
1.5. Push . . . . . 8
1.5.1. Example . . . . . 9
1.6. Ids . . . . . 9
2. Mailboxes . . . . . 9
2.1. Mailbox/get . . . . . 14
2.2. Mailbox/changes . . . . . 14
2.3. Mailbox/query . . . . . 14
2.4. Mailbox/queryChanges . . . . . 15
2.5. Mailbox/set . . . . . 16
2.6. Example . . . . . 17
3. Threads . . . . . 20
3.1. Thread/get . . . . . 22
3.1.1. Example . . . . . 22
3.2. Thread/changes . . . . . 22
4. Emails . . . . . 22
4.1. Properties of the Email Object . . . . . 23
4.1.1. Metadata . . . . . 24
4.1.2. Header Fields Parsed Forms . . . . . 26
4.1.3. Header Fields Properties . . . . . 32
4.1.4. Body Parts . . . . . 35
4.2. Email/get . . . . . 42
4.2.1. Example . . . . . 44
4.3. Email/changes . . . . . 45
4.4. Email/query . . . . . 45
4.4.1. Filtering . . . . . 46
4.4.2. Sorting . . . . . 49
4.4.3. Thread Collapsing . . . . . 50
4.5. Email/queryChanges . . . . . 51
4.6. Email/set . . . . . 51
4.7. Email/copy . . . . . 53
4.8. Email/import . . . . . 54
4.9. Email/parse . . . . . 56
4.10. Examples . . . . . 58
5. Search Snippets . . . . . 68
5.1. SearchSnippet/get . . . . . 69
5.2. Example . . . . . 71

6. Identities . . . . . 72
6.1. Identity/get . . . . . 73
6.2. Identity/changes . . . . . 73
6.3. Identity/set . . . . . 73
6.4. Example . . . . . 73
7. Email Submission . . . . . 74
7.1. EmailSubmission/get . . . . . 80
7.2. EmailSubmission/changes . . . . . 80
7.3. EmailSubmission/query . . . . . 80
7.4. EmailSubmission/queryChanges . . . . . 81
7.5. EmailSubmission/set . . . . . 81
7.5.1. Example . . . . . 84
8. Vacation Response . . . . . 86
8.1. VacationResponse/get . . . . . 87
8.2. VacationResponse/set . . . . . 88
9. Security Considerations . . . . . 88
9.1. EmailBodyPart Value . . . . . 88
9.2. HTML Email Display . . . . . 88
9.3. Multiple Part Display . . . . . 91
9.4. Email Submission . . . . . 91
9.5. Partial Account Access . . . . . 92
9.6. Permission to Send from an Address . . . . . 92
10. IANA Considerations . . . . . 93
10.1. JMAP Capability Registration for "mail" . . . . . 93
10.2. JMAP Capability Registration for "submission" . . . . . 93
10.3. JMAP Capability Registration for "vacationresponse" . . 94
10.4. IMAP and JMAP Keywords Registry . . . . . 94
10.4.1. Registration of JMAP Keyword "$draft" . . . . . 95
10.4.2. Registration of JMAP Keyword "$seen" . . . . . 96
10.4.3. Registration of JMAP Keyword "$flagged" . . . . . 97
10.4.4. Registration of JMAP Keyword "$answered" . . . . . 98
10.4.5. Registration of "$recent" Keyword . . . . . 99
10.5. IMAP Mailbox Name Attributes Registry . . . . . 99
10.5.1. Registration of "inbox" Role . . . . . 99
10.6. JMAP Error Codes Registry . . . . . 100
10.6.1. mailboxHasChild . . . . . 100
10.6.2. mailboxHasEmail . . . . . 100
10.6.3. blobNotFound . . . . . 100
10.6.4. tooManyKeywords . . . . . 101
10.6.5. tooManyMailboxes . . . . . 101
10.6.6. invalidEmail . . . . . 101
10.6.7. tooManyRecipients . . . . . 102
10.6.8. noRecipients . . . . . 102
10.6.9. invalidRecipients . . . . . 102
10.6.10. forbiddenMailFrom . . . . . 103
10.6.11. forbiddenFrom . . . . . 103
10.6.12. forbiddenToSend . . . . . 103

11. References . . . . . 104
11.1. Normative References . . . . . 104
11.2. Informative References . . . . . 107
Authors' Addresses . . . . . 108

## 1. Introduction

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

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

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

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

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

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

### 1.1. Notational Conventions

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

Type signatures, examples, and property descriptions in this document follow the conventions established in Section 1.1 of [RFC8620]. Data types defined in the core specification are also used in this document.

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

## 1.2. Terminology

This document uses the same terminology as in the core JMAP specification.

The terms Mailbox, Thread, Email, SearchSnippet, EmailSubmission and VacationResponse (with that specific capitalisation) are used to refer to the data types defined in this document and instances of those data types.

The term message refers to a document in Internet Message Format, as described in [RFC5322]. The Email data type represents messages in the mail store and associated metadata.

## 1.3. Additions to the Capabilities Object

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

This document defines three additional capability URIs.

### 1.3.1. urn:ietf:params:jmap:mail

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

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

- maxMailboxesPerEmail: "UnsignedInt|null"

  The maximum number of Mailboxes (see Section 2) that can be can assigned to a single Email object (see Section 4). This MUST be an integer >= 1, or null for no limit (or rather, the limit is always the number of Mailboxes in the account).

- maxMailboxDepth: "UnsignedInt|null"

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

- maxSizeMailboxName: "UnsignedInt"

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

- maxSizeAttachmentsPerEmail: "UnsignedInt"

  The maximum total size of attachments, in octets, allowed for a single Email object. A server MAY still reject the import or creation of an Email 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 that this limit is for the sum of unencoded attachment sizes. Users are generally not knowledgeable about encoding overhead, etc., nor should they need to be, so marketing and help materials normally tell them the "max size attachments". This is the unencoded size they see on their hard drive, 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 message [RFC5322], created by combining the attachments (often base64 encoded) with the message headers and bodies. For example, suppose the server advertises "maxSizeAttachmentsPerEmail: 50000000" (50 MB). The enforced server limit may be for a message size of 70000000 octets. Even with base64 encoding and a 2 MB HTML body, 50 MB attachments would fit under this limit.

- emailQuerySortOptions: "String[]"

  A list of all the values the server supports for the "property" field of the Comparator object in an "Email/query" sort (see Section 4.4.2). 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.

- mayCreateTopLevelMailbox: "Boolean"

  If true, the user may create a Mailbox (see Section 2) in this account with a null parentId. (Permission for creating a child of an existing Mailbox is given by the "myRights" property on that Mailbox.)

#### 1.3.2. urn:ietf:params:jmap:submission

This represents support for the Identity and EmailSubmission data types and associated API methods. The value of this property in the JMAP session "capabilities" property is an empty object.

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

- maxDelayedSend: "UnsignedInt"

  The number in seconds of the maximum delay the server supports in sending (see the EmailSubmission object description). This is 0 if the server does not support delayed send.

- submissionExtensions: "String[String[]]"

  The set of SMTP submission extensions supported by the server, which the client may use when creating an EmailSubmission object (see Section 7). Each key in the object is the "ehlo-name", and the value is a list of "ehlo-args".

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

  Examples of Submission extensions to include:

  * FUTURERELEASE [RFC4865]

  * SIZE [RFC1870]

  * DSN [RFC3461]

  * DELIVERYBY [RFC2852]

* MT-PRIORITY [RFC6710]

A JMAP server MAY advertise an extension and implement the semantics of that extension locally on the JMAP server even if a submission server used by JMAP doesn't implement it.

The full IANA registry of submission extensions can be found at <https://www.iana.org/assignments/mail-parameters>.

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

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

### 1.4. Data Type Support in Different Accounts

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

### 1.5. Push

Servers MUST support the JMAP push mechanisms, as specified in [RFC8620], Section 7, to receive notifications when the state changes for any of the types defined in this specification.

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

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

#### 1.5.1. Example

The client has registered for push notifications (see [RFC8620]) just for the EmailDelivery type. The user marks an Email as read on another device, causing the state string for the Email type to change; however, as nothing new was added to the store, the EmailDelivery state does not change and nothing is pushed to the client. A new message arrives in the user's inbox, again causing the Email state to change. This time, the EmailDelivery state also changes, and a StateChange object is pushed to the client with the new state string. The client may then resync to fetch the new Email immediately.

### 1.6. Ids

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

## 2. Mailboxes

A Mailbox represents a named set of Email objects. This is the primary mechanism for organising messages 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: "Id" (immutable; server-set)

  The id of the Mailbox.

- name: "String"

  User-visible name for the Mailbox, e.g., "Inbox". This MUST be a Net-Unicode string [RFC5198] of at least 1 character in length, subject to the maximum size given in the capability object. There MUST NOT be two sibling Mailboxes with both the same parent and the same name. Servers MAY reject names that violate server policy (e.g., names containing a slash (/) or control characters).

- parentId: "Id|null" (default: null)

  The Mailbox id for the parent of this Mailbox, or null if this Mailbox is at the top level. Mailboxes form acyclic graphs (forests) directed by the child-to-parent relationship. There MUST NOT be a loop.

- role: "String|null" (default: null)

  Identifies Mailboxes that have a particular common purpose (e.g., the "inbox"), regardless of the "name" property (which may be localised).

  This value is shared with IMAP (exposed in IMAP via the SPECIAL-USE extension [RFC6154]). However, unlike in IMAP, a Mailbox MUST only have a single role, and there MUST NOT be two Mailboxes in the same account with the same role. Servers providing IMAP access to the same data are encouraged to enforce these extra restrictions in IMAP as well. Otherwise, modifying the IMAP attributes to ensure compliance when exposing the data over JMAP is implementation dependent.

  The value MUST be one of the Mailbox attribute names listed in the IANA "IMAP Mailbox Name Attributes" registry at <https://www.iana.org/assignments/imap-mailbox-name-attributes/>, as established in [RFC8457], converted to lowercase. New roles may be established here in the future.

  An account is not required to have Mailboxes with any particular roles.

- sortOrder: "UnsignedInt" (default: 0)

  Defines the sort order of Mailboxes when presented in the client's UI, so it is consistent between devices. The number MUST be an integer in the range 0 <= 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: "UnsignedInt" (server-set)

  The number of Emails in this Mailbox.

- unreadEmails: "UnsignedInt" (server-set)

  The number of Emails in this Mailbox that have neither the "$seen" keyword nor the "$draft" keyword.

- totalThreads: "UnsignedInt" (server-set)

  The number of Threads where at least one Email in the Thread is in this Mailbox.

- unreadThreads: "UnsignedInt" (server-set)

  An indication of the number of "unread" Threads in the Mailbox.

  For compatibility with existing implementations, the way "unread Threads" is determined is not mandated in this document. The simplest solution to implement is simply the number of Threads where at least one Email in the Thread is both in this Mailbox and has neither the "$seen" nor "$draft" keywords.

  However, a quality implementation will return the number of unread items the user would see if they opened that Mailbox. A Thread is shown as unread if it contains any unread Emails that will be displayed when the Thread is opened. Therefore, "unreadThreads" should be the number of Threads where at least one Email in the Thread has neither the "$seen" nor the "$draft" keyword AND at least one Email in the Thread is in this Mailbox. Note that the unread Email does not need to be the one in this Mailbox. In addition, the trash Mailbox (that is, a Mailbox whose "role" is "trash") requires special treatment:

  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.

  For example, suppose you have an account where the entire contents is a single Thread with 2 Emails: an unread Email in the trash and a read Email in the inbox. The "unreadThreads" count would be 1 for the trash and 0 for the inbox.

- myRights: "MailboxRights" (server-set)

  The set of rights (Access Control Lists (ACLs)) the user has in relation to this Mailbox. These are backwards compatible with IMAP ACLs, as defined in [RFC4314]. A *MailboxRights* object has the following properties:

  * mayReadItems: "Boolean"

    If true, the user may use this Mailbox as part of a filter in an "Email/query" call, and the Mailbox may be included in the "mailboxIds" property of Email objects. Email objects may be fetched if they are in *at least one* Mailbox with this permission. If a sub-Mailbox is shared but not the parent Mailbox, this may be false. Corresponds to IMAP ACLs "lr" (if mapping from IMAP, both are required for this to be true).

  * mayAddItems: "Boolean"

    The user may add mail to this Mailbox (by either creating a new Email or moving an existing one). Corresponds to IMAP ACL "i".

  * mayRemoveItems: "Boolean"

    The user may remove mail from this Mailbox (by either changing the Mailboxes of an Email or destroying the Email). Corresponds to IMAP ACLs "te" (if mapping from IMAP, both are required for this to be true).

  * maySetSeen: "Boolean"

    The user may add or remove the "$seen" keyword to/from an Email. If an Email belongs to multiple Mailboxes, the user may only modify "$seen" if they have this permission for *all* of the Mailboxes. Corresponds to IMAP ACL "s".

  * maySetKeywords: "Boolean"

    The user may add or remove any keyword other than "$seen" to/from an Email. If an Email belongs to multiple Mailboxes, the user may only modify keywords if they have this permission for *all* of the Mailboxes. Corresponds to IMAP ACL "w".

  * mayCreateChild: "Boolean"

    The user may create a Mailbox with this Mailbox as its parent. Corresponds to IMAP ACL "k".

* mayRename: "Boolean"

  The user may rename the Mailbox or make it a child of another Mailbox. Corresponds to IMAP ACL "x" (although this covers both rename and delete permissions).

* mayDelete: "Boolean"

  The user may delete the Mailbox itself. Corresponds to IMAP ACL "x" (although this covers both rename and delete permissions).

* maySubmit: "Boolean"

  Messages may be submitted directly to this Mailbox. Corresponds to IMAP ACL "p".

o isSubscribed: "Boolean"

  Has the user indicated they wish to see this Mailbox in their client? This SHOULD default to false for Mailboxes in shared accounts the user has access to and true for any new Mailboxes created by the user 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 to only display Mailboxes where the "isSubscribed" property is set to true, and offer a separate UI to allow the user to see and subscribe/unsubscribe from the full set of Mailboxes. However, clients MAY choose to ignore this property, either entirely for ease of implementation or just for an account where "isPersonal" is true (indicating it is the user’s own rather than a shared account).

  This property corresponds to IMAP [RFC3501] mailbox subscriptions.

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

The following JMAP methods are supported.

### 2.1. Mailbox/get

This is a standard "/get" method as described in [RFC8620], Section 5.1. The "ids" argument may be "null" to fetch all at once.

### 2.2. Mailbox/changes

This is a standard "/changes" method as described in [RFC8620], Section 5.2 but with one extra argument to the response:

- updatedProperties: "String[]|null"

  If only the "totalEmails", "unreadEmails", "totalThreads", and/or "unreadThreads" Mailbox properties have changed since the old state, this will be the list of properties that may have changed. If the server is unable to tell if only counts have changed, it MUST just be null.

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

### 2.3. Mailbox/query

This is a standard "/query" method as described in [RFC8620], Section 5.5 but with the following additional request argument:

- sortAsTree: "Boolean" (default: false)

  If true, when sorting the query results and comparing Mailboxes A and B:

  * If A is an ancestor of B, it always comes first regardless of the sort comparators. Similarly, if A is descendant of B, then B always comes first.

  * Otherwise, if A and B do not share a "parentId", find the nearest ancestors of each that do have the same "parentId" and compare the sort properties on those Mailboxes instead.

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

- filterAsTree: "Boolean" (default: false)

  If true, a Mailbox is only included in the query if all its ancestors are also included in the query according to the filter.

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

- parentId: "Id|null"

  The Mailbox "parentId" property must match the given value exactly.

- name: "String"

  The Mailbox "name" property contains the given string.

- role: "String|null"

  The Mailbox "role" property must match the given value exactly.

- hasAnyRole: "Boolean"

  If true, a Mailbox matches if it has any non-null value for its "role" property.

- isSubscribed: "Boolean"

  The "isSubscribed" property of the Mailbox must be identical to the value given to match the condition.

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

The following Mailbox properties MUST be supported for sorting:

- "sortOrder"

- "name"

## 2.4. Mailbox/queryChanges

This is a standard "/queryChanges" method as described in [RFC8620], Section 5.6.

## 2.5. Mailbox/set

This is a standard "/set" method as described in [RFC8620], Section 5.3 but with the following additional request argument:

- onDestroyRemoveEmails: "Boolean" (default: false)

  If false, any attempt to destroy a Mailbox that still has Emails in it will be rejected with a "mailboxHasEmail" SetError. If true, any Emails that were in the Mailbox will be removed from it, and if in no other Mailboxes, they 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 Email assigned to it, and the "onDestroyRemoveEmails" argument was false.

## 2.6. Example

Fetching all Mailboxes in an account:

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

And the response:

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

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

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

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

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

Here’s an example where we try to rename one Mailbox and destroy another:

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

Suppose the rename succeeds, but we don’t have permission to destroy the Mailbox we tried to destroy; we might get back:

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

## 3. Threads

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

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

1. An identical message id [RFC5322] appears in both messages in any of the Message-Id, In-Reply-To, and References header fields.

2. After stripping automatically added prefixes such as "Fwd:", "Re:", "[List-Tag]", etc., and ignoring white space, 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 messages are delivered out of order for some reason, a user may have two Emails in the same Thread but without headers that associate them with each other. The arrival of a third Email may provide the missing references to join them all together into a single Thread. Since the "threadId" of an Email is immutable, if the server wishes to merge the Threads, it MUST handle this by deleting and reinserting (with a new Email id) the Emails that change "threadId".

A *Thread* object has the following properties:

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

  The id of the Thread.

o emailIds: "Id[]" (server-set)

  The ids of the Emails in the Thread, sorted by the "receivedAt" date of the Email, oldest first. If two Emails have an identical date, the sort is server dependent but MUST be stable (sorting by id is recommended).

The following JMAP methods are supported.

### 3.1. Thread/get

This is a standard "/get" method as described in [RFC8620], Section 5.1.

#### 3.1.1. Example

Request:

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

with response:

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

### 3.2. Thread/changes

This is a standard "/changes" method as described in [RFC8620], Section 5.2.

## 4. Emails

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

### 4.1. Properties of the Email Object

Broadly, a message consists of two parts: a list of header fields and then a body. The Email data type 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 header fields it wants to process, as this allows it to avoid the complexities of various encodings that are required in a valid message per RFC 5322.

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 a message body (normally plaintext 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 when they are transmitted, and some clients send text and images intended to be displayed inline in the body (or even videos and sound clips) as multiple parts rather than a single HTML part with referenced images.

  Because MIME allows for multiple representations of the same data (using "multipart/alternative"), there is a "textBody" property (which prefers a plaintext 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 to have the server handle decoding the charset into unicode. This data is in a separate property rather than on the EmailBodyPart object to avoid duplication of large amounts of data, as the same part may be included twice if the client fetches more than one of bodyStructure, textBody, and htmlBody.

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

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

#### 4.1.1. Metadata

These properties represent metadata about the message in the mail store and are not derived from parsing the message itself.

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

  The id of the Email object. Note that this is the JMAP object id, NOT the Message-ID header field value of the message [RFC5322].

- blobId: "Id" (immutable; server-set)

  The id representing the raw octets of the message [RFC5322] for this Email. This may be used to download the raw original message or to attach it directly to another Email, etc.

- threadId: "Id" (immutable; server-set)

  The id of the Thread to which this Email belongs.

- mailboxIds: "Id[Boolean]"

  The set of Mailbox ids this Email belongs to. An Email in the mail store MUST belong to one or more Mailboxes at all times (until it is destroyed). 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 get special treatment. 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 (and so are not counted in the "totalEmails", "unreadEmails", "totalThreads", and "unreadThreads" Mailbox properties).

  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 it MUST NOT include any of these characters:

  ```
  ( ) { ] % * " \
  ```

  Because JSON is case sensitive, servers MUST return keywords in lowercase.

  The IANA "IMAP and JMAP Keywords" registry at <https://www.iana.org/assignments/imap-jmap-keywords/> 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.

- size: "UnsignedInt" (immutable; server-set)

  The size, in octets, of the raw data for the message [RFC5322] (as referenced by the "blobId", i.e., the number of octets in the file the user would download).

- receivedAt: "UTCDate" (immutable; default: time of creation on server)

  The date the Email was received by the message store. This is the "internal date" in IMAP [RFC3501].

#### 4.1.2. Header Fields Parsed Forms

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

##### 4.1.2.1. Raw

Type: "String"

The raw octets of the header field value from the first octet following the header field name terminating colon, up to but excluding the header field terminating CRLF. Any standards-compliant message MUST be either ASCII (RFC 5322) or UTF-8 (RFC 6532); however, other encodings exist in the wild. A server SHOULD replace any octet or octet run with the high bit set that violates UTF-8 syntax with the unicode replacement character (U+FFFD). Any NUL octet MUST be dropped.

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

#### 4.1.2.2. Text

Type: "String"

The header field value with:

1. White space unfolded (as defined in [RFC5322], Section 2.2.3).

2. The terminating CRLF at the end of the value removed.

3. Any SP characters at the beginning of the value removed.

4. Any syntactically correct encoded sections [RFC2047] with a known character set decoded. Any NUL octets or control characters encoded per [RFC2047] are dropped from the decoded value. Any text that looks like syntax per [RFC2047] but violates placement or white space rules per [RFC2047] MUST NOT be decoded.

5. The resulting unicode converted to Normalization Form C (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
- Comments
- Keywords
- List-Id
- Any header field not defined in [RFC5322] or [RFC2369]

#### 4.1.2.3. Addresses

Type: "EmailAddress[]"

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

An *EmailAddress* object has the following properties:

- name: "String|null"

  The "display-name" of the "mailbox" [RFC5322]. If this is a "quoted-string":

  1. The surrounding DQUOTE characters are removed.

  2. Any "quoted-pair" is decoded.

  3. White space is unfolded, and then any leading and trailing white space is removed.

  If there is no "display-name" but there is a "comment" immediately following the "addr-spec", the value of this SHOULD be used instead. Otherwise, this property is null.

- email: "String"

  The "addr-spec" of the "mailbox" [RFC5322].

Any syntactically correct encoded sections [RFC2047] with a known encoding MUST be decoded, following the same rules as for the Text form (see Section 4.1.2.2).

Parsing SHOULD be best effort in the face of invalid structure to accommodate invalid messages and semi-complete drafts. EmailAddress objects MAY have an "email" property that does not conform to the "addr-spec" form (for example, may not contain an @ symbol).

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

```
            "  James Smythe" <james@example.com>, Friends:
              jane@example.com, John Smth               <john@example.com>;
```

would be parsed as:

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

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

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

#### 4.1.2.4. GroupedAddresses

Type: "EmailAddressGroup[]"

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

An *EmailAddressGroup* object has the following properties:

o name: "String|null"

The "display-name" of the "group" [RFC5322], or null if the addresses are not part of a group. If this is a "quoted-string", it is processed the same as the "name" in the EmailAddress type.

o addresses: "EmailAddress[]"

The "mailbox" values that belong to this group, represented as EmailAddress objects.

Any syntactically correct encoded sections [RFC2047] with a known encoding MUST be decoded, following the same rules as for the Text form (see Section 4.1.2.2).

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

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

```
            "  James Smythe" <james@example.com>, Friends:
              jane@example.com, John Smth               <john@example.com>;
```

would be parsed as:

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

To prevent obviously nonsense behaviour, which can lead to interoperability issues, this form may only be fetched or set for the same header fields as the Addresses form (see Section 4.1.2.3).

#### 4.1.2.5. MessageIds

Type: "String[]|null"

The header field is parsed as a list of "msg-id" values, as specified in [RFC5322], Section 3.6.4, into the "String[]" type. Comments and/or folding white space (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 field not defined in [RFC5322] or [RFC2369]

#### 4.1.2.6. Date

Type: "Date|null"

The header field 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 field not defined in [RFC5322] or [RFC2369]

#### 4.1.2.7. URLs

Type: "String[]|null"

The header field 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 field 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 field not defined in [RFC5322] or [RFC2369]

### 4.1.3. Header Fields Properties

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

- headers: "EmailHeader[]" (immutable)

  This is a list of all header fields [RFC5322], 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 for 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 to 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 an 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 field 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 RFC 5322, 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".

#### 4.1.4. Body Parts

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

An *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: "Id|null"

  The id representing the raw octets of the contents of the part, after decoding any known Content-Transfer-Encoding (as defined in [RFC2045]), or null if, and only if, the part is of type "multipart/*". Note that two parts may be transfer-encoded differently but have the same blob id if their decoded octets are identical and the server is using a secure hash of the data for the blob id. If the transfer encoding is unknown, it is treated as though it had no transfer encoding.

- size: "UnsignedInt"

  The size, in octets, of the raw data after content transfer decoding (as referenced by the "blobId", i.e., the number of octets in the file the user would download).

- headers: "EmailHeader[]"

  This is a list of all header fields in the part, in the order they appear in the message. The values are in Raw form.

- name: "String|null"

  This is the decoded "filename" parameter of the Content-Disposition header field per [RFC2231], or (for compatibility with existing systems) if not present, then it’s the decoded "name" parameter of the Content-Type header field per [RFC2047].

- 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 not of type "text/*". If there is no Content-Type header field, or it exists and is of type "text/*" but has no charset parameter, this is the implicit charset as per the MIME standard: "us-ascii".

- disposition: "String|null"

  The value of the Content-Disposition header field of the part, if present; otherwise, it's 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, it's null. CFWS and surrounding angle brackets ("<>") are removed. This may be used to reference the content from within a "text/html" body part [HTML] using the "cid:" protocol, as defined in [RFC2392].

- 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[]|null"

  If the 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, 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-Transfer-Encoding and the Content-Type charset, if both known to the server, and with any CRLF replaced with a single LF. The server MAY use heuristics to determine the charset to use for decoding if the charset is unknown, no charset is given, or it believes the charset given is incorrect. Decoding is best effort; the server 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, the charset was unknown, or the content-transfer-encoding was unknown.

  * isTruncated: "Boolean" (default: false)

    This is true if the "value" has been truncated.

  See the Security Considerations section for issues related to truncation and heuristic determination of the 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, traversing depth-first, of all parts in "bodyStructure" that 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 that a "text/html" body part [HTML] may reference image parts in attachments by using "cid:" links to reference the Content-Id, as defined in [RFC2392], 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 to true if the "attachments" list contains at least one item that does not have "Content-Disposition: inline". The server MAY ignore parts in this list that are processed automatically in some way or are referenced as embedded images in one of the "text/html" parts of the message.

  The server MAY set hasAttachment based on implementation-defined or site-configurable heuristics.

- preview: "String" (immutable; server-set)

  A plaintext fragment of the message body. This is intended to be shown as a preview line when listing messages in the mail store and may be truncated when shown. The server may choose which part of the message to include in the preview; skipping quoted sections and salutations and collapsing white space can result in a more useful preview.

  This MUST NOT be more than 256 characters in length.

  As this is derived from the message content by the server, and the algorithm for doing so could change over time, fetching this for an Email a second time MAY return a different result. However, the previous value is not considered incorrect, and the change SHOULD NOT cause the Email object to be considered as changed by the server.

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

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

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

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

    for ( let i = 0; i < parts.length; i += 1 ) {
        let part = parts[i];
        let isMultipart = part.type.startsWith( 'multipart/' );
        // Is this a body part rather than an attachment
        let isInline = part.disposition != "attachment" &&
            // Must be one of the allowed body types
            ( part.type == "text/plain" ||
              part.type == "text/html" ||
              isInlineMediaType( part.type ) ) &&
```

```
            // If multipart/related, only the first part can be inline
            // If a text part with a filename, and not the first item
            // in the multipart, assume it is an attachment
            ( i === 0 ||
              ( multipartType != "related" &&
                ( isInlineMediaType( part.type ) || !part.name ) ) );

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

```
      if ( multipartType == 'alternative' && textBody && htmlBody ) {
          // Found HTML part only
          if ( textLength == textBody.length &&
                  htmlLength != htmlBody.length ) {
              for ( let i = htmlLength; i < htmlBody.length; i += 1 ) {
                  textBody.push( htmlBody[i] );
              }
          }
          // Found plaintext 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 has gone through a list software manager that attaches a header and 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

This is a standard "/get" method as described in [RFC8620], Section 5.1 with the following additional request arguments:

- bodyProperties: "String[]"

  A list of properties to fetch for each EmailBodyPart returned. If omitted, this defaults to:

  ```
  [ "partId", "blobId", "size", "name", "type", "charset",
    "disposition", "cid", "language", "location" ]
  ```

- 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: "UnsignedInt" (default: 0)

  If greater than zero, the "value" property of any EmailBodyValue object returned in "bodyValues" MUST be truncated if necessary so it does not exceed this number of octets in size. If 0 (the default), no truncation occurs.

  The server MUST ensure the truncation results in valid UTF-8 and does not occur mid-codepoint. If the part is of type "text/html", the server SHOULD NOT truncate inside an HTML tag, e.g., in the middle of "<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 field is requested as a property, the capitalization of the property name in the response MUST be identical to that used in the request.

#### 4.2.1. Example

Request:

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

and response:

```
[[ "Email/get", {
  "accountId": "abc",
  "state": "41234123231",
  "list": [
    {
      "id": "f123u457",
      "threadId": "ef1314a",
      "mailboxIds": { "f123": true },
      "from": [{ "name": "Joe Bloggs", "email": "joe@example.com" }],
      "subject": "Dinner on Thursday?",
      "receivedAt": "2013-10-13T14:12:00Z",
      "header:List-POST:asURLs": [
        "mailto:partytime@lists.example.com"
      ],
      "htmlBody": [{
        "partId": "1",
        "blobId": "B841623871",
        "size": 283331,
        "type": "text/html"
```

```
        }, {
          "partId": "2",
          "blobId": "B319437193",
          "size": 10343,
          "type": "text/plain"
        }],
        "bodyValues": {
          "1": {
            "isEncodingProblem": false,
            "isTruncated": true,
            "value": "<html><body><p>Hello ..."
          },
          "2": {
            "isEncodingProblem": false,
            "isTruncated": false,
            "value": "-- Sent by your friendly mailing list ..."
          }
        }
      }
    ],
    "notFound": [ "f123u456" ]
  }, "#1" ]]
```

### 4.3. Email/changes

This is a standard "/changes" method as described in [RFC8620], Section 5.2. If generating intermediate states for a large set of changes, it is recommended that newer changes be returned first, as these are generally of more interest to users.

### 4.4. Email/query

This is a standard "/query" method as described in [RFC8620], Section 5.5 but with the following additional request 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 only one Email at most will be included in the list for any given Thread.

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

#### 4.4.1. Filtering

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

- inMailbox: "Id"

  A Mailbox id. An Email must be in this Mailbox to match the condition.

- inMailboxOtherThan: "Id[]"

  A list of Mailbox ids. An Email must be in at least one Mailbox not in this list to match the condition. This is to allow messages solely in trash/spam to be easily excluded from a search.

- before: "UTCDate"

  The "receivedAt" date-time of the Email must be before this date-time to match the condition.

- after: "UTCDate"

  The "receivedAt" date-time of the Email must be the same or after this date-time to match the condition.

- minSize: "UnsignedInt"

  The "size" property of the Email must be equal to or greater than this number to match the condition.

- maxSize: "UnsignedInt"

  The "size" property of the Email 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 MUST look up text in the From, To, Cc, Bcc, and Subject header fields of the message and SHOULD look inside any "text/*" or other body parts that may be converted to text by the server. The server MAY extend the search to any additional textual property.

- 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 header field of the message.

- body: "String"

  Looks for the text in one of the body parts of the message. The server MAY exclude MIME body parts with content media types other than "text/*" and "message/*" from consideration in search matching. Care should be taken to match based on the text content actually presented to an end user by viewers for that media type or otherwise identified as appropriate for search indexing. Matching document metadata uninteresting to an end user (e.g., markup tag and attribute names) is undesirable.

- 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 encoded sections [RFC2047] 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, Cascading Style Sheets (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.

Within a phrase, to match one of the following characters you MUST escape it by prefixing it with a backslash (\):

’ " \

- 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 (not part of a phrase) MAY be matched on a whole-word basis using stemming (for example, a text search for "bus" would match "buses" but not "business").

#### 4.4.2. Sorting

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

- "receivedAt" - The "receivedAt" date as returned in the Email object.

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

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

- "from" - This is taken to be either the "name" property or if null/empty, the "email" property of the *first* EmailAddress object in the Email’s "from" property. If still none, consider the value to be the empty string.

- "to" - This is taken to be either the "name" property or if null/empty, the "email" property of the *first* EmailAddress object in the Email’s "to" property. If still none, consider the value to be the empty string.

- "subject" - This is taken to be the base subject of the message, 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 an additional "keyword" property on the Comparator object, or false otherwise.

- "allInThreadHaveKeyword" - This value MUST be considered true for the Email if *all* of the Emails in the same Thread have the keyword given as an additional "keyword" property on the Comparator object.

- "someInThreadHaveKeyword" - This value MUST be considered true for the Email if *any* of the Emails in the same Thread have the keyword given as an additional "keyword" property on the Comparator object.

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

Example sort:

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

This would sort Emails in flagged Threads first (the Thread is considered flagged if any Email within it is flagged), in subject order second, and then from newest first for messages with the same subject. If two Emails have identical values for all three properties, then the order is server dependent but must be stable.

### 4.4.3. Thread Collapsing

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

### 4.5. Email/queryChanges

This is a standard "/queryChanges" method as described in [RFC8620], Section 5.6 with the following additional request argument:

- collapseThreads: "Boolean" (default: false)

  The "collapseThreads" argument that was used with "Email/query".

### 4.6. Email/set

This is a standard "/set" method as described in [RFC8620], Section 5.3. 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

The format of the "keywords"/"mailboxIds" properties means that when updating an Email, you can either replace the entire set of keywords/Mailboxes (by setting the full value of the property) or add/remove individual ones using the JMAP patch syntax (see [RFC8620], Section 5.3 for the specification and Section 5.7 for an example).

Due to the format of the Email object, when creating an Email, there are a number of ways to specify the same information. To ensure that the message [RFC5322] 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 with "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 and it MUST be of type "text/plain".
- If given, htmlBody MUST contain exactly one body part and it MUST be 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 included but is ignored by the server (the size is actually calculated from the blob content itself).
  - A Content-Transfer-Encoding header field MUST NOT be given.
- 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 message generated may be invalid per RFC 5322. For example, if it is a half-finished draft, the To header field may have a value that does not conform to the required syntax for this header. 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 a "role" property equal to "trash", and remove all other Mailbox ids.

When emptying the trash, clients SHOULD NOT destroy Emails that are also in a Mailbox other than trash. For those Emails, they SHOULD just remove the trash Mailbox from the Email.

For successfully created Email objects, the "created" response contains the "id", "blobId", "threadId", and "size" properties of the object.

The following extra SetError types are defined:

For "create":

- "blobNotFound": At least one blob id given for an EmailBodyPart doesn't exist. An extra "notFound" property of type "Id[]" MUST be included in the SetError 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 set of Mailboxes that this Email is in would exceed a server-defined maximum.

## 4.7. Email/copy

This is a standard "/copy" method as described in [RFC8620], Section 5.4, except only the "mailboxIds", "keywords", and "receivedAt" properties may be set during the copy. This method cannot modify the message represented by the Email.

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

For successfully copied Email objects, the "created" response contains the "id", "blobId", "threadId", and "size" properties of the new object.

### 4.8. Email/import

The "Email/import" method adds messages [RFC5322] to the set of Emails in an account. The server MUST support messages with Email Address Internationalization (EAI) headers [RFC6532]. The messages must first be uploaded as blobs using the standard upload mechanism. The method takes the following arguments:

- accountId: "Id"

  The id of the account to use.

- ifInState: "String|null"

  This is a state string as returned by the "Email/get" method. If supplied, the string must match the current state of the account referenced by the accountId; otherwise, the method will be aborted and a "stateMismatch" error returned. If null, any changes will be applied to the current state.

- emails: "Id[EmailImport]"

  A map of creation id (client specified) to EmailImport objects.

An *EmailImport* object has the following properties:

- blobId: "Id"

  The id of the blob containing the raw message [RFC5322].

- mailboxIds: "Id[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 most recent Received header, or time of import on server if none)

  The "receivedAt" date to set on the Email.

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

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

If the "blobId", "mailboxIds", or "keywords" properties are invalid (e.g., missing, wrong type, id not found), the server MUST reject the import with an "invalidProperties" SetError.

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

If the blob referenced is not a valid message [RFC5322], 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: "Id"

  The id of the account used for this call.

- oldState: "String|null"

  The state string that would have been returned by "Email/get" on this account before making the requested changes, or null if the server doesn’t know what the previous state string was.

- newState: "String"

  The state string that will now be returned by "Email/get" on this account.

- created: "Id[Email]|null"

  A map of the creation id to an object containing the "id", "blobId", "threadId", and "size" properties for each successfully imported Email, or null if none.

- notCreated: "Id[SetError]|null"

  A map of the creation id to a SetError object for each Email that failed to be created, or null if all successful. The possible errors are defined above.

The following additional errors may be returned instead of the "Email/import" response:

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

## 4.9. Email/parse

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

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

- 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: "Id"

  The id of the account to use.

- blobIds: "Id[]"

  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[]"

  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: "UnsignedInt" (default: 0)

  If greater than zero, the "value" property of any EmailBodyValue object returned in "bodyValues" MUST be truncated if necessary so it does not exceed this number of octets in size. If 0 (the default), no truncation occurs.

  The server MUST ensure the truncation results in valid UTF-8 and does not occur mid-codepoint. If the part is of type "text/html", the server SHOULD NOT truncate inside an HTML tag, e.g., in the middle of "<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).

The response has the following arguments:

- accountId: "Id"

  The id of the account used for the call.

- parsed: "Id[Email]|null"

  A map of blob id to parsed Email representation for each successfully parsed blob, or null if none.

- notParsable: "Id[]|null"

  A list of ids given that corresponded to blobs that could not be parsed as Emails, or null if none.

- notFound: "Id[]|null"

  A list of blob ids given that could not be found, or null if none.

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

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

## 4.10. Examples

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

```
[[ "Email/query",{
  "accountId": "ue150411c",
  "filter": {
    "inMailbox": "fb666a55"
  },
  "sort": [{
    "isAscending": false,
    "property": "receivedAt"
  }],
  "collapseThreads": true,
```

```
    "position": 0,
    "limit": 30,
    "calculateTotal": true
  }, "0" ],
  [ "Email/get", {
    "accountId": "ue150411c",
    "#ids": {
      "resultOf": "0",
      "name": "Email/query",
      "path": "/ids"
    },
    "properties": [
      "threadId"
    ]
  }, "1" ],
  [ "Thread/get", {
    "accountId": "ue150411c",
    "#ids": {
      "resultOf": "1",
      "name": "Email/get",
      "path": "/list/*/threadId"
    }
  }, "2" ],
  [ "Email/get", {
    "accountId": "ue150411c",
    "#ids": {
      "resultOf": "2",
      "name": "Thread/get",
      "path": "/list/*/emailIds"
    },
    "properties": [
      "threadId",
      "mailboxIds",
      "keywords",
      "hasAttachment",
      "from",
      "subject",
      "receivedAt",
      "size",
      "preview"
    ]
  }, "3" ]]
```

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

"0": This asks the server for the ids of the first 30 Email objects in the inbox, sorted newest first, ignoring Emails from the same Thread as a newer Email in the Mailbox (i.e., it is the first 30 unique Threads).

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

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

"3": Finally, we fetch the information we need to display the Mailbox listing (but no more!) for every Email in each of these 30 Threads. The client may aggregate this data for display, for example, by showing the Thread as "flagged" if any of the Emails in it has the "$flagged" keyword.

The response from the server may look something like this:

```
 [[ "Email/query", {
   "accountId": "ue150411c",
   "queryState": "09aa9a075588-780599:0",
   "canCalculateChanges": true,
   "position": 0,
   "total": 115,
   "ids": [ "Ma783e5cdf5f2deffbc97930a",
     "M9bd17497e2a99cb345fc1d0a", ... ]
 }, "0" ],
 [ "Email/get", {
   "accountId": "ue150411c",
   "state": "780599",
   "list": [{
     "id": "Ma783e5cdf5f2deffbc97930a",
     "threadId": "T36703c2cfe9bd5ed"
   }, {
     "id": "M9bd17497e2a99cb345fc1d0a",
     "threadId": "T0a22ad76e9c097a1"
   }, ... ],
   "notFound": []
 }, "1" ],
 [ "Thread/get", {
   "accountId": "ue150411c",
   "state": "22a8728b",
   "list": [{
     "id": "T36703c2cfe9bd5ed",
     "emailIds": [ "Ma783e5cdf5f2deffbc97930a" ]
```

```
      }, {
        "id": "T0a22ad76e9c097a1",
        "emailIds": [ "M3b568670a63e5d100f518fa5",
          "M9bd17497e2a99cb345fc1d0a" ]
      },  ... ],
      "notFound": []
    }, "2" ],
    [ "Email/get", {
      "accountId": "ue150411c",
      "state": "780599",
      "list": [{
        "id": "Ma783e5cdf5f2deffbc97930a",
        "threadId": "T36703c2cfe9bd5ed",
        "mailboxIds": {
          "fb666a55": true
        },
        "keywords": {
          "$seen": true,
          "$flagged": true
        },
        "hasAttachment": true,
        "from": [{
          "email": "jdoe@example.com",
          "name": "Jane Doe"
        }],
        "subject": "The Big Reveal",
        "receivedAt": "2018-06-27T00:20:35Z",
        "size": 175047,
        "preview": "As you may be aware, we are required to prepare a
          presentation where we wow a panel of 5 random members of the
          public, on or before 30 June each year.  We have drafted..."
      },
      ...
      ],
      "notFound": []
    }, "3" ]]
```

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

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

The server applies this and sends the success response:

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

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

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

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

The response:

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

The client can update its local cache of the query results by removing "M9bd17497e2a99cb345fc1d0a" and then splicing in "Me8de6c9f6de198239b982ea2" at position 0. As it does not have the data for this new Email, it will then fetch it (it also could have done this in the same request using back-references).

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

The user starts composing a new Email. The email is plaintext and the client knows the email in English so adds this metadata to the body part. The user saves a draft while the composition is still in progress. The client sends:

```
  [[ "Email/set", {
    "accountId": "ue150411c",
    "create": {
      "k192": {
        "mailboxIds": {
          "2ea1ca41b38e": true
        },
        "keywords": {
          "$seen": true,
          "$draft": true
        },
        "from": [{
          "name": "Joe Bloggs",
          "email": "joe@example.com"
        }],
        "subject": "World domination",
        "receivedAt": "2018-07-10T01:03:11Z",
        "sentAt": "2018-07-10T11:03:11+10:00",
        "bodyStructure": {
          "type": "text/plain",
          "partId": "bd48",
          "header:Content-Language": "en"
        },
        "bodyValues": {
          "bd48": {
            "value": "I have the most brilliant plan.  Let me tell
              you all about it.  What we do is, we",
            "isTruncated": false
          }
        }
      }
    }
  }, "0" ]]
```

The server creates the message and sends the success response:

```
[[ "Email/set", {
  "accountId": "ue150411c",
  "oldState": "780823",
  "newState": "780839",
  "created": {
    "k192": {
      "id": "Mf40b5f831efa7233b9eb1c7f",
      "blobId": "Gf40b5f831efa7233b9eb1c7f8f97d84eeeee64f7",
      "threadId": "Td957e72e89f516dc",
      "size": 359
    }
  },
  ...
}, "0" ]]
```

The message created on the server looks something like this:

```
Message-Id: <bbce0ae9-58be-4b24-ac82-deb840d58016@sloti7d1t02>
User-Agent: Cyrus-JMAP/3.1.6-736-gdfb8e44
Mime-Version: 1.0
Date: Tue, 10 Jul 2018 11:03:11 +1000
From: "Joe Bloggs" <joe@example.com>
Subject: World domination
Content-Language: en
Content-Type: text/plain

I have the most brilliant plan.  Let me tell you all about it.  What we
do is, we
```

The user adds a recipient and converts the message to HTML so they can add formatting, then saves an updated draft:

```
[[ "Email/set", {
  "accountId": "ue150411c",
  "create": {
    "k1546": {
      "mailboxIds": {
        "2ea1ca41b38e": true
      },
      "keywords": {
        "$seen": true,
        "$draft": true
      },
      "from": [{
        "name": "Joe Bloggs",
        "email": "joe@example.com"
```

```
      }],
      "to": [{
        "name": "John",
        "email": "john@example.com"
      }],
      "subject": "World domination",
      "receivedAt": "2018-07-10T01:05:08Z",
      "sentAt": "2018-07-10T11:05:08+10:00",
      "bodyStructure": {
        "type": "multipart/alternative",
        "subParts": [{
          "partId": "a49d",
          "type": "text/html",
          "header:Content-Language": "en"
        }, {
          "partId": "bd48",
          "type": "text/plain",
          "header:Content-Language": "en"
        }]
      },
      "bodyValues": {
        "bd48": {
          "value": "I have the most brilliant plan.  Let me tell
            you all about it.  What we do is, we",
          "isTruncated": false
        },
        "a49d": {
          "value": "<!DOCTYPE html><html><head><title></title>
            <style type=\"text/css\">div{font-size:16px}</style></head>
            <body><div>I have the most <b>brilliant</b> plan.  Let me
            tell you all about it.  What we do is, we</div></body>
            </html>",
          "isTruncated": false
        }
      }
    }
  },
  "destroy": [ "Mf40b5f831efa7233b9eb1c7f" ]
}, "0" ]]
```

The server creates the new draft, deletes the old one, and sends the success response:

```
    [[ "Email/set", {
      "accountId": "ue150411c",
      "oldState": "780839",
      "newState": "780842",
      "created": {
        "k1546": {
          "id": "Md45b47b4877521042cec0938",
          "blobId": "Ge8de6c9f6de198239b982ea214e0f3a704e4af74",
          "threadId": "Td957e72e89f516dc",
          "size": 11721
        }
      },
      "destroyed": [ "Mf40b5f831efa7233b9eb1c7f" ],
      ...
    }, "0" ]]
```

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

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

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

```
    [[ "Email/copy", {
      "fromAccountId": "ue150411c",
      "accountId": "u6c6c41ac",
      "oldState": "7ee7e9263a6d",
      "newState": "5a0d2447ed26",
      "created": {
        "k45": {
          "id": "M138f9954a5cd2423daeafa55",
          "blobId": "G6b9fb047cba722c48c611e79233d057c6b0b74e8",
          "threadId": "T2f242ea424a4079a",
          "size": 11721
        }
      },
      "notCreated": null
    }, "0" ],
    [ "Email/set", {
      "accountId": "ue150411c",
      "oldState": "780842",
      "newState": "780871",
      "destroyed": [ "Md45b47b4877521042cec0938" ],
      ...
    }, "0" ]]
```

## 5. Search Snippets

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

A *SearchSnippet* object has the following properties:

o emailId: "Id"

The Email id the snippet applies to.

- subject: "String|null"

  If text from the filter matches the subject, this is the subject of the Email with the following transformations:

  1. Any instance of the following three characters MUST be replaced by an appropriate HTML entity: & (ampersand), < (less-than sign), and > (greater-than sign) [HTML]. Other characters MAY also be replaced with an HTML entity form.

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

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

- preview: "String|null"

  If text from the filter matches the plaintext or HTML body, this is the relevant section of the body (converted to plaintext if originally HTML), with the same transformations as the "subject" property. It MUST NOT be bigger than 255 octets in size. If the body does not contain a match for the text from the filter, this property is null.

What is a relevant section of the body for preview is server defined. If the server is unable to determine search snippets, it MUST return null for both the "subject" and "preview" properties.

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

The following JMAP method is supported.

## 5.1. SearchSnippet/get

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

- accountId: "Id"

  The id of the account to use.

- filter: "FilterOperator|FilterCondition|null"

  The same filter as passed to "Email/query"; see the description of this method in Section 4.4 for details.

- emailIds: "Id[]"

  The ids of the Emails to fetch snippets for.

The response has the following arguments:

- accountId: "Id"

  The id of the account used for 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: "Id[]|null"

  An array of Email ids requested that could not be found, or null if all ids were found.

As the search snippets are derived from the message content and the algorithm for doing so could change over time, fetching the same snippets a second time MAY return a different result. However, the previous value is not considered incorrect, so there is no state string or update mechanism needed.

The following additional errors may be returned instead of the "SearchSnippet/get" response:

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

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

### 5.2. Example

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

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

Example response:

```
[[ "SearchSnippet/get", {
  "accountId": "ue150411c",
  "list": [{
      "emailId": "M44200ec123de277c0c1ce69c",
      "subject": null,
      "preview": null
  }, {
      "emailId": "M7bcbcb0b58d7729686e83d99",
      "subject": "The <mark>Foo</mark>sball competition",
      "preview": "...year the <mark>foo</mark>sball competition will
        be held in the Stadium de ..."
  }, {
      "emailId": "M28d12783a0969584b6deaac0",
      "subject": null,
      "preview": "...the <mark>Foo</mark>/bar method results often
        returns &lt;1 widget rather than the complete..."
  },
  ...
  ],
  "notFound": null
}, "0" ]]
```

## 6. Identities

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

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

  The id of the Identity.

- name: "String" (default: "")

  The "From" name the client SHOULD use when creating a new Email from this Identity.

- email: "String" (immutable)

  The "From" email address the client MUST use when creating a new Email from this Identity. If the "mailbox" part of the address (the section before the "@") is the single character "*" (e.g., "*@example.com"), the client may use any valid address ending in that domain (e.g., "foo@example.com").

- replyTo: "EmailAddress[]|null" (default: null)

  The Reply-To value the client SHOULD set when creating a new Email from this Identity.

- bcc: "EmailAddress[]|null" (default: null)

  The Bcc value the client SHOULD set when creating a new Email from this Identity.

- textSignature: "String" (default: "")

  A signature the client SHOULD insert into new plaintext 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: "")

  A 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 HTML. 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. Attempts to destroy an Identity with "mayDelete: false" will be rejected with a standard "forbidden" SetError.

See the "Addresses" header form description in the Email object (Section 4.1.2.3) for the definition of EmailAddress.

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

The following JMAP methods are supported.

## 6.1. Identity/get

This is a standard "/get" method as described in [RFC8620], Section 5.1. The "ids" argument may be null to fetch all at once.

## 6.2. Identity/changes

This is a standard "/changes" method as described in [RFC8620], Section 5.2.

## 6.3. Identity/set

This is a standard "/set" method as described in [RFC8620], Section 5.3. The following extra SetError types are defined:

For "create":

- "forbiddenFrom": The user is not allowed to send from the address given as the "email" property of the Identity.

## 6.4. Example

Request:

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

with response:

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

## 7. Email Submission

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

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

  The id of the EmailSubmission.

- identityId: "Id" (immutable)

  The id of the Identity to associate with this submission.

- emailId: "Id" (immutable)

  The id of the Email to send. The Email being sent does not have to be a draft, for example, when "redirecting" an existing Email to a different address.

- threadId: "Id" (immutable; server-set)

  The Thread id of the Email to send. This is set by the server to the "threadId" property of the Email referenced by the "emailId".

- envelope: "Envelope|null" (immutable)

  Information for use when sending via SMTP. An *Envelope* object has the following properties:

  * mailFrom: "Address"

    The email address to use as the return address in the SMTP submission, plus any parameters to pass with the MAIL FROM address. The JMAP server MAY allow the address to be the empty string.

    When a JMAP server performs an SMTP message submission, it MAY use the same id string for the ENVID parameter [RFC3461] and the EmailSubmission object id. Servers that do this MAY replace a client-provided value for ENVID with a server-provided value.

  * rcptTo: "Address[]"

    The email addresses to send the message to, and any RCPT TO parameters to pass with the recipient.

  An *Address* object has the following properties:

  * email: "String"

    The email address being represented by the object. This is a "Mailbox" as used in the Reverse-path or Forward-path of the MAIL FROM or RCPT TO command in [RFC5321].

  * parameters: "Object|null"

    Any parameters to send with the email address (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 is either the parameter value (type

"String") or null if the parameter does not take a value. For both name and value, any xtext or unitext encodings are removed (see [RFC3461] and [RFC6533]) and JSON string encoding is 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 address in the Sender header field, if present; otherwise, it's the email address in the From header field, if present. In either case, no parameters are added.

  If multiple addresses are present in one of these header fields, or there is more than one Sender/From header field, the server SHOULD reject the EmailSubmission as invalid; otherwise, it MUST take the first address in the last Sender/From header field.

  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 header fields, if present, with no parameters for any of them.

o sendAt: "UTCDate" (immutable; server-set)

The date the submission was/will be released for delivery. If the client successfully used FUTURERELEASE [RFC4865] with the submission, this MUST be the time when the server will release the message; otherwise, it MUST be the time the EmailSubmission was created.

o undoStatus: "String"

This represents whether the submission may be canceled. This is server set on create and MUST be one of the following values:

* "pending": It may be possible to cancel this submission.

* "final": The message 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 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 message, but it 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 message 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 submission'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 message, or in a later Delivery Status Notification (DSN, as defined in [RFC3464]) response for the message. This SHOULD be the response to the RCPT TO stage, unless this was accepted and the message as a whole was rejected at the end of the DATA stage, in which case the DATA stage reply SHOULD be used instead.

    Multi-line SMTP responses should be concatenated to a single string as follows:

    + The hyphen following the SMTP code on all but the last line is replaced with a space.

    + Any prefix in common with the first line is stripped from lines after the first.

    + CRLF is replaced by a space.

For example:

```
 550-5.7.1 Our system has detected that this message is
 550 5.7.1 likely spam.
```

would become:

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

For messages 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 message has been successfully delivered to the recipient. This MUST be one of the following values:

+ "queued": The message is in a local mail queue and the status will change once it exits the local mail queues. The "smtpReply" property may still change.
+ "yes": The message was successfully delivered to the mail store 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 that successful relaying to an external SMTP server SHOULD NOT be taken as an indication that the message has successfully reached the final mail store. 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 message 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 message content has been displayed to the recipient. Note that there is no guarantee that the recipient has noticed, read, or understood the content.

If a Message Disposition Notification (MDN) is received for this recipient with Disposition-Type (as per [RFC8098], Section 3.2.6.2) equal to "displayed", this property SHOULD be set to "yes".

The server MAY also set this property based on other feedback channels.

o dsnBlobIds: "Id[]" (server-set)

A list of blob ids for DSNs [RFC3464] received for this submission, in order of receipt, oldest first. The blob is the whole MIME message (with a top-level content-type of "multipart/report"), as received.

o mdnBlobIds: "Id[]" (server-set)

A list of blob ids for MDNs [RFC8098] received for this submission, in order of receipt, oldest first. The blob is the whole MIME message (with a top-level content-type of "multipart/report"), as received.

JMAP servers MAY choose not to expose DSN and MDN responses as Email objects if they correlate to an EmailSubmission object. It SHOULD only do this if it exposes them in the "dsnBlobIds" and "mdnblobIds" fields instead, and it 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 at any time after the message is successfully sent or after it has finished retrying to send the message. For very basic SMTP proxies, this MAY be immediately after creation, as it has no way to assign a real id and return the information again if fetched later.

The following JMAP methods are supported.

### 7.1. EmailSubmission/get

This is a standard "/get" method as described in [RFC8620], Section 5.1.

### 7.2. EmailSubmission/changes

This is a standard "/changes" method as described in [RFC8620], Section 5.2.

### 7.3. EmailSubmission/query

This is a standard "/query" method as described in [RFC8620], Section 5.5.

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

- identityIds: "Id[]"

  The EmailSubmission "identityId" property must be in this list to match the condition.

- emailIds: "Id[]"

  The EmailSubmission "emailId" property must be in this list to match the condition.

- threadIds: "Id[]"

  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-time to match the condition.

- after: "UTCDate"

  The "sendAt" property of the EmailSubmission object must be the same as or after this date-time to match the condition.

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

The following EmailSubmission properties MUST be supported for sorting:

- "emailId"
- "threadId"
- "sentAt"

### 7.4. EmailSubmission/queryChanges

This is a standard "/queryChanges" method as described in [RFC8620], Section 5.6.

### 7.5. EmailSubmission/set

This is a standard "/set" method as described in [RFC8620], Section 5.3 with the following two additional request arguments:

- onSuccessUpdateEmail: "Id[PatchObject]|null"

  A map of EmailSubmission id to an object containing properties to update on the Email object referenced by the EmailSubmission if the create/update/destroy succeeds. (For references to EmailSubmissions created in the same "/set" invocation, this is equivalent to a creation-reference, so the id will be the creation id prefixed with a "#".)

- onSuccessDestroyEmail: "Id[]|null"

  A list of EmailSubmission ids for which the Email with the corresponding "emailId" should be destroyed if the create/update/destroy succeeds. (For references to EmailSubmission creations, this is equivalent to a creation-reference, so the id will be the creation id prefixed with a "#".)

After all create/update/destroy items in the "EmailSubmission/set" invocation have been processed, a single implicit "Email/set" call MUST be made to perform any changes requested in these two arguments. The response to this MUST be returned after the "EmailSubmission/set" response.

An Email is sent by creating an EmailSubmission object. When processing each create, the server must check that the message is valid, and the user has sufficient authorisation to send it. If the creation succeeds, the message will be sent to the recipients given in the envelope "rcptTo" parameter. The server MUST remove any Bcc header field present on the message during delivery. The server MAY add or remove other header fields from the submitted message 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 submission (i.e., it does not cancel a future send). The "emailId" and "threadId" properties of the EmailSubmission object remain, but trying to fetch them (with a standard "Email/get" call) will return a "notFound" error if the corresponding objects have been destroyed.

Similarly, destroying an EmailSubmission object MUST NOT affect the deliveries it represents. It purely removes the record of the submission. The server MAY automatically destroy EmailSubmission objects after some time or in response to other triggers, and MAY forbid the client from manually destroying EmailSubmission objects.

If the message to be sent is larger than the server supports sending, a standard "tooLarge" SetError MUST be returned. A "maxSize" "UnsignedInt" property MUST be present on the SetError specifying the maximum size of a message that may be sent, in octets.

If the Email or Identity id given cannot be found, the submission creation is rejected with a standard "invalidProperties" SetError.

The following extra SetError types are defined:

For "create":

- "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.

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

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

- "invalidRecipients" - The "rcptTo" property of the envelope (supplied or generated) contains at least one rcptTo value, which is not a valid email address for sending to. An "invalidRecipients" "String[]" property MUST also be present on the SetError, which is a list of the invalid addresses.

- "forbiddenMailFrom" - The server does not permit the user to send a message with the envelope From address [RFC5321].

- "forbiddenFrom" - The server does not permit the user to send a message with the From header field [RFC5322] of the message 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.

For "update":

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

#### 7.5.1. Example

The following example presumes a draft of the Email 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": "I64588216",
      "emailId": "M7f6ed5bcfd7e2604d1753f6c",
      "envelope": {
        "mailFrom": {
          "email": "john@example.com",
          "parameters": null
        },
        "rcptTo": [{
          "email": "jane@example.com",
          "parameters": null
        },
        ...
        ]
      }
    }
  },
  "onSuccessUpdateEmail": {
    "#k1490": {
      "mailboxIds/7cb4e8ee-df87-4757-b9c4-2ea1ca41b38e": null,
      "mailboxIds/73dbcb4b-bffc-48bd-8c2a-a2e91ca672f6": true,
      "keywords/$draft": null
    }
  }
}, "0" ]]
```

A successful response might look like this. Note that there are two responses due to the implicit "Email/set" call, but both have the same method call id as they are due to the same call in the request:

```
[[ "EmailSubmission/set", {
  "accountId": "ue411d190",
  "oldState": "012421s6-8nrq-4ps4-n0p4-9330r951ns21",
  "newState": "355421f6-8aed-4cf4-a0c4-7377e951af36",
  "created": {
    "k1490": {
      "id": "ES-3bab7f9a-623e-4acf-99a5-2e67facb02a0"
    }
  }
}, "0" ],
[ "Email/set", {
  "accountId": "ue411d190",
  "oldState": "778193",
  "newState": "778197",
  "updated": {
      "M7f6ed5bcfd7e2604d1753f6c": null
  }
}, "0" ]]
```

Suppose instead an admin has removed sending rights for the user, so the submission is rejected with a "forbiddenToSend" error. The description argument of the error is intended for display to the user, so it should be localised appropriately. Let’s suppose the request was sent with an Accept-Language header like this:

```
Accept-Language: de;q=0.9,en;q=0.8
```

The server should attempt to choose the best localisation from those it has available based on the Accept-Language header, as described in [RFC8620], Section 3.8. If the server has English, French, and German translations, it would choose German as the preferred language and return a response like this:

```
[[ "EmailSubmission/set", {
  "accountId": "ue411d190",
  "oldState": "012421s6-8nrq-4ps4-n0p4-9330r951ns21",
  "newState": "012421s6-8nrq-4ps4-n0p4-9330r951ns21",
  "notCreated": {
    "k1490": {
      "type": "forbiddenToSend",
      "description": "Verzeihung, wegen verdaechtiger Aktivitaeten Ihres
       Benutzerkontos haben wir den Versand von Nachrichten gesperrt.
       Bitte wenden Sie sich fuer Hilfe an unser Support Team."
    }
  }
}, "0" ]]
```

## 8. Vacation Response

A vacation response sends an automatic reply when a message is delivered to the mail store, informing the original sender that their message may not be read for some time.

Automated message sending can produce undesirable behaviour. To avoid this, implementors MUST follow the recommendations set forth in [RFC3834].

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

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

  The id of the object. There is only ever one VacationResponse object, and its id is "singleton".

- isEnabled: "Boolean"

  Should a vacation response be sent if a message arrives between the "fromDate" and "toDate"?

- fromDate: "UTCDate|null"

  If "isEnabled" is true, messages that arrive on or after this date-time (but before the "toDate" if defined) should receive the user's vacation response. If null, the vacation response is effective immediately.

- toDate: "UTCDate|null"

  If "isEnabled" is true, messages that arrive before this date-time (but on or after the "fromDate" if defined) should receive the user's vacation response. If null, the vacation response is effective indefinitely.

- subject: "String|null"

  The subject that will be used by the message sent in response to messages when the vacation response is enabled. If null, an appropriate subject SHOULD be set by the server.

- textBody: "String|null"

  The plaintext body to send in response to messages when the vacation response is enabled. If this is null, the server SHOULD generate a plaintext body part from the "htmlBody" when sending vacation responses but MAY choose to send the response as HTML only. If both "textBody" and "htmlBody" are null, an appropriate default body SHOULD be generated for responses by the server.

- htmlBody: "String|null"

  The HTML body to send in response to messages when the vacation response is enabled. If this is null, the server MAY choose to generate an HTML body part from the "textBody" when sending vacation responses or MAY choose to send the response as plaintext only.

The following JMAP methods are supported.

### 8.1. VacationResponse/get

This is a standard "/get" method as described in [RFC8620], Section 5.1.

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

### 8.2. VacationResponse/set

This is a standard "/set" method as described in [RFC8620], Section 5.3.

## 9. Security Considerations

All security considerations of JMAP [RFC8620] apply to this specification. Additional considerations specific to the data types and functionality introduced by this document are described in the following subsections.

### 9.1. EmailBodyPart Value

Service providers typically perform security filtering on incoming messages, and it’s important that the detection of content-type and charset for the security filter aligns with the heuristics performed by JMAP servers. Servers that apply heuristics to determine the content-type or charset for an 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 mail 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 rescan the truncated value for malicious content before returning it.

### 9.2. HTML Email Display

HTML message bodies provide richer formatting for messages but present a number of security challenges, especially when embedded in a webmail context in combination with interface HTML. Clients that render HTML messages should carefully consider the potential risks, including:

- Embedded JavaScript can rewrite the message 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 messages 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 (however, this 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 messages 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 a message 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 message will alter their appearance, potentially breaking the layout of the message.

- 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 a message 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 message 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. Sanitising HTML correctly is tricky, and implementors are strongly recommended to use a well-tested library with a carefully vetted whitelist-only approach. New features with unexpected security characteristics may be added to HTML rendering engines in the future; a blacklist approach is likely to result in security issues.

  Subtle differences in parsing of HTML can introduce security flaws: to filter with 100% accuracy, you need to use the same parser that the HTML rendering engine will use.

- Encapsulating the message in an "<iframe sandbox>", as defined in [HTML], Section 4.7.6, can help mitigate a number of risks. This will:

  * Disable JavaScript.

  * Disable form submission.

  * Prevent drawing outside of its bounds or conflicts between message CSS and interface CSS.

  * Establish a unique anonymous origin, separate to the containing origin.

- A strong Content Security Policy (see <https://www.w3.org/TR/CSP3/>) can, among other things, block JavaScript and the 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 (see <https://www.w3.org/TR/referrer-policy/>).

- 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. Multiple Part Display

Messages may consist of multiple parts to be displayed sequentially as a body. Clients MUST render each part in isolation and MUST NOT concatenate the raw text values to render. Doing so may change the overall semantics of the message. If the client or server is decrypting a Pretty Good Privacy (PGP) or S/MIME encrypted part, concatenating with other parts may leak the decrypted text to an attacker, as described in [EFAIL].

### 9.4. Email Submission

SMTP submission servers [RFC6409] use a number of mechanisms to mitigate damage caused by compromised user accounts and end-user systems including rate limiting, anti-virus/anti-spam milters (mail filters), and other technologies. The technologies work better when they have more information about the client connection. If JMAP email submission is implemented as a proxy to an SMTP submission server, it is useful to communicate this information from the JMAP proxy to the submission server. The de facto XCLIENT extension to SMTP [XCLIENT] can be used to do this, but use of an authenticated channel is recommended to limit use of that extension to explicitly authorised proxies.

JMAP servers that proxy to an SMTP submission server SHOULD allow use of the submissions port [RFC8314]. Implementation of a mechanism similar to SMTP XCLIENT is strongly encouraged. While Simple Authentication and Security Layer (SASL) PLAIN over TLS [RFC4616] is presently the mandatory-to-implement mechanism for interoperability with SMTP submission servers [RFC4954], a JMAP submission proxy SHOULD implement and prefer a stronger mechanism for this use case such as TLS client certificate authentication with SASL EXTERNAL ([RFC4422], Appendix A) or Salted Challenge Response Authentication Mechanism (SCRAM) [RFC7677].

In the event the JMAP server directly relays mail to SMTP servers in other administrative domains, implementation of the de facto [milter] protocol is strongly encouraged to integrate with third-party products that address security issues including anti-virus/anti-spam, reputation protection, compliance archiving, and data loss prevention. Proxying to a local SMTP submission server may be a simpler way to provide such security services.

### 9.5. Partial Account Access

A user may only have permission to access a subset of the data that exists in an account. To avoid leaking unauthorised information, in such a situation, the server MUST treat any data the user does not have permission to access the same as if it did not exist.

For example, suppose user A has an account with two Mailboxes, inbox and sent, but only shares the inbox with user B. In this case, when user B fetches Mailboxes for this account, the server MUST behave as though the sent Mailbox did not exist. Similarly, when querying or fetching Email objects, it MUST treat any messages that just belong to the sent Mailbox as though they did not exist. Fetching Thread objects MUST only return ids for Email objects the user has permission to access; if none, the Thread again MUST be treated the same as if it did not exist.

If the server forbids a single account from having two identical messages, or two messages with the same Message-Id header field, a user with write access can use the error returned by trying to create/import such a message to detect whether it already exists in an inaccessible portion of the account.

### 9.6. Permission to Send from an Address

In recent years, the email ecosystem has moved towards associating trust with the From address in the message [RFC5322], particularly with schemes such as Domain-based Message Authentication, Reporting, and Conformance (DMARC) [RFC7489].

The set of Identity objects (see Section 6) in an account lets the client know which email addresses the user has permission to send from. Each email submission is associated with an Identity, and servers SHOULD reject submissions where the From header field of the message does not correspond to the associated Identity.

The server MAY allow an exception to send an exact copy of an existing message received into the mail store to another address (otherwise known as "redirecting" or "bouncing"), although it is RECOMMENDED the server limit this to destinations the user has verified they also control.

If the user attempts to create a new Identity object, the server MUST reject it with the appropriate error if the user does not have permission to use that email address to send from.

The SMTP MAIL FROM address [RFC5321] is often confused with the From message header field [RFC5322]. The user generally only ever sees the address in the message header field, and this is the primary one to enforce. However, the server MUST also enforce appropriate restrictions on the MAIL FROM address [RFC5321] to stop the user from flooding a third-party address with bounces and non-delivery notices.

The JMAP submission model provides separate errors for impermissible addresses in either context.

# 10. IANA Considerations

## 10.1. JMAP Capability Registration for "mail"

IANA has registered the "mail" JMAP Capability as follows:

Capability Name: urn:ietf:params:jmap:mail

Specification document: this document

Intended use: common

Change Controller: IETF

Security and privacy considerations: this document, Section 9

## 10.2. JMAP Capability Registration for "submission"

IANA has registered the "submission" JMAP Capability as follows:

Capability Name: urn:ietf:params:jmap:submission

Specification document: this document

Intended use: common

Change Controller: IETF

Security and privacy considerations: this document, Section 9

### 10.3. JMAP Capability Registration for "vacationresponse"

IANA has registered the "vacationresponse" JMAP Capability as follows:

Capability Name: urn:ietf:params:jmap:vacationresponse

Specification document: this document

Intended use: common

Change Controller: IETF

Security and privacy considerations: this document, Section 9

### 10.4. IMAP and JMAP Keywords Registry

This document makes two changes to the IMAP keywords registry as defined in [RFC5788].

First, the name of the registry is changed to the "IMAP and JMAP Keywords" registry.

Second, a scope column is added to the template and registry indicating whether a keyword applies to "IMAP-only", "JMAP-only", "both", or "reserved". All keywords already in the IMAP keyword registry have been marked with a scope of "both". The "reserved" status can be used to prevent future registration of a name that would be confusing if registered. Registration of keywords with scope "reserved" omit most fields in the registration template (see registration of "$recent" below for an example); such registrations are intended to be infrequent.

IMAP clients MAY silently ignore any keywords marked "JMAP-only" or "reserved" in the event they appear in protocol. JMAP clients MAY silently ignore any keywords marked "IMAP-only" or "reserved" in the event they appear in protocol.

New "JMAP-only" keywords are registered in the following subsections. These keywords correspond to IMAP system keywords and are thus not appropriate for use in IMAP. These keywords cannot be subsequently registered for use in IMAP except via standards action.

#### 10.4.1. Registration of JMAP Keyword "$draft"

This registers the "JMAP-only" keyword "$draft" in the "IMAP and JMAP Keywords" registry.

Keyword name: $draft

Scope: JMAP-only

Purpose (description): This is set when the user wants to treat the message as a draft the user is composing. This is the JMAP equivalent of the IMAP \Draft flag.

Private or Shared on a server: BOTH

Is it an advisory keyword or may it cause an automatic action: Automatic. If the account has an IMAP mailbox marked with the \Drafts special use attribute [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 will typically 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" argument. In a mail store 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: this document

Person & email address to contact for further information: JMAP mailing list <jmap@ietf.org>

Intended usage: COMMON

Owner/Change controller: IESG

#### 10.4.2. Registration of JMAP Keyword "$seen"

This registers the "JMAP-only" keyword "$seen" in the "IMAP and JMAP Keywords" registry.

Keyword name: $seen

Scope: JMAP-only

Purpose (description): This is set when the user wants to treat the message as read. This is the JMAP equivalent of the IMAP \Seen flag.

Private or Shared on a server: BOTH

Is it an advisory keyword or may it cause an automatic action: Advisory. However, certain JMAP computed values such as "unreadEmails" will change as a result of changing this flag.

When/by whom the keyword is set/cleared: This is set by a JMAP client when it presents the message content to the user; clients often offer an option to clear this flag. In a mail store 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: this document

Person & email address to contact for further information: JMAP mailing list <jmap@ietf.org>

Intended usage: COMMON

Owner/Change controller: IESG

#### 10.4.3. Registration of JMAP Keyword "$flagged"

This registers the "JMAP-only" keyword "$flagged" in the "IMAP and JMAP Keywords" registry.

Keyword name: $flagged

Scope: JMAP-only

Purpose (description): This is set when the user wants to treat the message as flagged for urgent/special attention. This is the JMAP equivalent of the IMAP \Flagged flag.

Private or Shared on a server: BOTH

Is it an advisory keyword or may it cause an automatic action: Automatic. If the account has an IMAP mailbox marked with the \Flagged special use attribute [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 mail store 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: this document

Person & email address to contact for further information: JMAP mailing list <jmap@ietf.org>

Intended usage: COMMON

Owner/Change controller: IESG

#### 10.4.4. Registration of JMAP Keyword "$answered"

This registers the "JMAP-only" keyword "$answered" in the "IMAP and JMAP Keywords" registry.

Keyword name: $answered

Scope: JMAP-only

Purpose (description): This is set when the message has been answered.

Private or Shared on a server: BOTH

Is it an advisory keyword or may it cause an automatic action: Advisory.

When/by whom the keyword is set/cleared: JMAP clients typically set this when submitting a reply or answer to the message. It may be set by the "EmailSubmission/set" operation with an "onSuccessUpdateEmail" argument. In a mail store shared by JMAP and IMAP, this is also set and cleared as necessary so it matches the IMAP \Answered flag.

Related keywords: None

Related IMAP/JMAP Capabilities: None

Security Considerations: A server implementing this keyword as a shared keyword may disclose that a user has replied to a message. This information would be exposed to other users with read permission for the Mailbox keywords.

Published specification: this document

Person & email address to contact for further information: JMAP mailing list <jmap@ietf.org>

Intended usage: COMMON

Owner/Change controller: IESG

#### 10.4.5. Registration of "$recent" Keyword

This registers the keyword "$recent" in the "IMAP and JMAP Keywords" registry.

Keyword name: $recent

Scope: reserved

Purpose (description): This keyword is not used to avoid confusion with the IMAP \Recent system flag.

Published specification: this document

Person & email address to contact for further information:
JMAP mailing list <jmap@ietf.org>

Owner/Change controller: IESG

### 10.5. IMAP Mailbox Name Attributes Registry

#### 10.5.1. Registration of "inbox" Role

This registers the "JMAP-only" "inbox" attribute in the "IMAP Mailbox Name Attributes" registry, as established in [RFC8457].

Attribute Name: Inbox

Description: New mail is delivered here by default.

Reference: This document, Section 10.5.1

Usage Notes: JMAP only

### 10.6. JMAP Error Codes Registry

The following subsections register several new error codes in the "JMAP Error Codes" registry, as defined in [RFC8620].

#### 10.6.1. mailboxHasChild

JMAP Error Code: mailboxHasChild

Intended use: common

Change controller: IETF

Reference: This document, Section 2.5

Description: The Mailbox still has at least one child Mailbox. The client MUST remove these before it can delete the parent Mailbox.

#### 10.6.2. mailboxHasEmail

JMAP Error Code: mailboxHasEmail

Intended use: common

Change controller: IETF

Reference: This document, Section 2.5

Description: The Mailbox has at least one message assigned to it, and the onDestroyRemoveEmails argument was false.

#### 10.6.3. blobNotFound

JMAP Error Code: blobNotFound

Intended use: common

Change controller: IETF

Reference: This document, Section 4.6

Description: At least one blob id referenced in the object doesn’t exist.

#### 10.6.4. tooManyKeywords

JMAP Error Code: tooManyKeywords

Intended use: common

Change controller: IETF

Reference: This document, Section 4.6

Description: The change to the Email's keywords would exceed a server-defined maximum.

#### 10.6.5. tooManyMailboxes

JMAP Error Code: tooManyMailboxes

Intended use: common

Change controller: IETF

Reference: This document, Section 4.6

Description: The change to the set of Mailboxes that this Email is in would exceed a server-defined maximum.

#### 10.6.6. invalidEmail

JMAP Error Code: invalidEmail

Intended use: common

Change controller: IETF

Reference: This document, Section 7.5

Description: The Email to be sent is invalid in some way.

#### 10.6.7. tooManyRecipients

JMAP Error Code: tooManyRecipients

Intended use: common

Change controller: IETF

Reference: This document, Section 7.5

Description: The envelope [RFC5321] (supplied or generated) has more recipients than the server allows.

#### 10.6.8. noRecipients

JMAP Error Code: noRecipients

Intended use: common

Change controller: IETF

Reference: This document, Section 7.5

Description: The envelope [RFC5321] (supplied or generated) does not have any rcptTo email addresses.

#### 10.6.9. invalidRecipients

JMAP Error Code: invalidRecipients

Intended use: common

Change controller: IETF

Reference: This document, Section 7.5

Description: The rcptTo property of the envelope [RFC5321] (supplied or generated) contains at least one rcptTo value that is not a valid email address for sending to.

#### 10.6.10. forbiddenMailFrom

JMAP Error Code: forbiddenMailFrom

Intended use: common

Change controller: IETF

Reference: This document, Section 7.5

Description: The server does not permit the user to send a message with this envelope From address [RFC5321].

#### 10.6.11. forbiddenFrom

JMAP Error Code: forbiddenFrom

Intended use: common

Change controller: IETF

Reference: This document, Sections 6.3 and 7.5

Description: The server does not permit the user to send a message with the From header field [RFC5322] of the message to be sent.

#### 10.6.12. forbiddenToSend

JMAP Error Code: forbiddenToSend

Intended use: common

Change controller: IETF

Reference: This document, Section 7.5

Description: The user does not have permission to send at all right now.

# 11. References

## 11.1. Normative References

[HTML] Faulkner, S., Eicholz, A., Leithead, T., Danilo, A., and S. Moon, "HTML 5.2", World Wide Web Consortium Recommendation REC-html52-20171214, December 2017, <https://www.w3.org/TR/html52/>.

[RFC1870] Klensin, J., Freed, N., and K. Moore, "SMTP Service Extension for Message Size Declaration", STD 10, RFC 1870, DOI 10.17487/RFC1870, November 1995, <https://www.rfc-editor.org/info/rfc1870>.

[RFC2045] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies", RFC 2045, DOI 10.17487/RFC2045, November 1996, <https://www.rfc-editor.org/info/rfc2045>.

[RFC2047] Moore, K., "MIME (Multipurpose Internet Mail Extensions) Part Three: Message Header Extensions for Non-ASCII Text", RFC 2047, DOI 10.17487/RFC2047, November 1996, <https://www.rfc-editor.org/info/rfc2047>.

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

[RFC2231] Freed, N. and K. Moore, "MIME Parameter Value and Encoded Word Extensions: Character Sets, Languages, and Continuations", RFC 2231, DOI 10.17487/RFC2231, November 1997, <https://www.rfc-editor.org/info/rfc2231>.

[RFC2369] Neufeld, G. and J. Baer, "The Use of URLs as Meta-Syntax for Core Mail List Commands and their Transport through Message Header Fields", RFC 2369, DOI 10.17487/RFC2369, July 1998, <https://www.rfc-editor.org/info/rfc2369>.

[RFC2392] Levinson, E., "Content-ID and Message-ID Uniform Resource Locators", RFC 2392, DOI 10.17487/RFC2392, August 1998, <https://www.rfc-editor.org/info/rfc2392>.

[RFC2557] Palme, J., Hopmann, A., and N. Shelness, "MIME Encapsulation of Aggregate Documents, such as HTML (MHTML)", RFC 2557, DOI 10.17487/RFC2557, March 1999, <https://www.rfc-editor.org/info/rfc2557>.

[RFC2852] Newman, D., "Deliver By SMTP Service Extension", RFC 2852, DOI 10.17487/RFC2852, June 2000, <https://www.rfc-editor.org/info/rfc2852>.

[RFC3282] Alvestrand, H., "Content Language Headers", RFC 3282, DOI 10.17487/RFC3282, May 2002, <https://www.rfc-editor.org/info/rfc3282>.

[RFC3461] Moore, K., "Simple Mail Transfer Protocol (SMTP) Service Extension for Delivery Status Notifications (DSNs)", RFC 3461, DOI 10.17487/RFC3461, January 2003, <https://www.rfc-editor.org/info/rfc3461>.

[RFC3463] Vaudreuil, G., "Enhanced Mail System Status Codes", RFC 3463, DOI 10.17487/RFC3463, January 2003, <https://www.rfc-editor.org/info/rfc3463>.

[RFC3464] Moore, K. and G. Vaudreuil, "An Extensible Message Format for Delivery Status Notifications", RFC 3464, DOI 10.17487/RFC3464, January 2003, <https://www.rfc-editor.org/info/rfc3464>.

[RFC3834] Moore, K., "Recommendations for Automatic Responses to Electronic Mail", RFC 3834, DOI 10.17487/RFC3834, August 2004, <https://www.rfc-editor.org/info/rfc3834>.

[RFC4314] Melnikov, A., "IMAP4 Access Control List (ACL) Extension", RFC 4314, DOI 10.17487/RFC4314, December 2005, <https://www.rfc-editor.org/info/rfc4314>.

[RFC4422] Melnikov, A., Ed. and K. Zeilenga, Ed., "Simple Authentication and Security Layer (SASL)", RFC 4422, DOI 10.17487/RFC4422, June 2006, <https://www.rfc-editor.org/info/rfc4422>.

[RFC4616] Zeilenga, K., Ed., "The PLAIN Simple Authentication and Security Layer (SASL) Mechanism", RFC 4616, DOI 10.17487/RFC4616, August 2006, <https://www.rfc-editor.org/info/rfc4616>.

[RFC4865] White, G. and G. Vaudreuil, "SMTP Submission Service Extension for Future Message Release", RFC 4865, DOI 10.17487/RFC4865, May 2007, <https://www.rfc-editor.org/info/rfc4865>.

[RFC4954] Siemborski, R., Ed. and A. Melnikov, Ed., "SMTP Service Extension for Authentication", RFC 4954, DOI 10.17487/RFC4954, July 2007, <https://www.rfc-editor.org/info/rfc4954>.

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

[RFC5248] Hansen, T. and J. Klensin, "A Registry for SMTP Enhanced Mail System Status Codes", BCP 138, RFC 5248, DOI 10.17487/RFC5248, June 2008, <https://www.rfc-editor.org/info/rfc5248>.

[RFC5256] Crispin, M. and K. Murchison, "Internet Message Access Protocol - SORT and THREAD Extensions", RFC 5256, DOI 10.17487/RFC5256, June 2008, <https://www.rfc-editor.org/info/rfc5256>.

[RFC5321] Klensin, J., "Simple Mail Transfer Protocol", RFC 5321, DOI 10.17487/RFC5321, October 2008, <https://www.rfc-editor.org/info/rfc5321>.

[RFC5322] Resnick, P., Ed., "Internet Message Format", RFC 5322, DOI 10.17487/RFC5322, October 2008, <https://www.rfc-editor.org/info/rfc5322>.

[RFC5788] Melnikov, A. and D. Cridland, "IMAP4 Keyword Registry", RFC 5788, DOI 10.17487/RFC5788, March 2010, <https://www.rfc-editor.org/info/rfc5788>.

[RFC6154] Leiba, B. and J. Nicolson, "IMAP LIST Extension for Special-Use Mailboxes", RFC 6154, DOI 10.17487/RFC6154, March 2011, <https://www.rfc-editor.org/info/rfc6154>.

[RFC6409] Gellens, R. and J. Klensin, "Message Submission for Mail", STD 72, RFC 6409, DOI 10.17487/RFC6409, November 2011, <https://www.rfc-editor.org/info/rfc6409>.

[RFC6532] Yang, A., Steele, S., and N. Freed, "Internationalized Email Headers", RFC 6532, DOI 10.17487/RFC6532, February 2012, <https://www.rfc-editor.org/info/rfc6532>.

[RFC6533] Hansen, T., Ed., Newman, C., and A. Melnikov, "Internationalized Delivery Status and Disposition Notifications", RFC 6533, DOI 10.17487/RFC6533, February 2012, <https://www.rfc-editor.org/info/rfc6533>.

[RFC6710] Melnikov, A. and K. Carlberg, "Simple Mail Transfer Protocol Extension for Message Transfer Priorities", RFC 6710, DOI 10.17487/RFC6710, August 2012, <https://www.rfc-editor.org/info/rfc6710>.

[RFC7677] Hansen, T., "SCRAM-SHA-256 and SCRAM-SHA-256-PLUS Simple Authentication and Security Layer (SASL) Mechanisms", RFC 7677, DOI 10.17487/RFC7677, November 2015, <https://www.rfc-editor.org/info/rfc7677>.

[RFC8098] Hansen, T., Ed. and A. Melnikov, Ed., "Message Disposition Notification", STD 85, RFC 8098, DOI 10.17487/RFC8098, February 2017, <https://www.rfc-editor.org/info/rfc8098>.

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

[RFC8314] Moore, K. and C. Newman, "Cleartext Considered Obsolete: Use of Transport Layer Security (TLS) for Email Submission and Access", RFC 8314, DOI 10.17487/RFC8314, January 2018, <https://www.rfc-editor.org/info/rfc8314>.

[RFC8457] Leiba, B., Ed., "IMAP "$Important" Keyword and "\Important" Special-Use Attribute", RFC 8457, DOI 10.17487/RFC8457, September 2018, <https://www.rfc-editor.org/info/rfc8457>.

[RFC8474] Gondwana, B., Ed., "IMAP Extension for Object Identifiers", RFC 8474, DOI 10.17487/RFC8474, September 2018, <https://www.rfc-editor.org/info/rfc8474>.

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

## 11.2. Informative References

[EFAIL] Poddebniak, D., Dresen, C., Mueller, J., Ising, F., Schinzel, S., Friedberger, S., Somorovsky, J., and J. Schwenk, "Efail: Breaking S/MIME and OpenPGP Email Encryption using Exfiltration Channels", August 2018, <https://www.usenix.org/system/files/conference/usenixsecurity18/sec18-poddebniak.pdf>.

[milter] Postfix, "Postfix before-queue Milter support", 2019, <http://www.postfix.org/MILTER_README.html>.

[RFC3501] Crispin, M., "INTERNET MESSAGE ACCESS PROTOCOL - VERSION 4rev1", RFC 3501, DOI 10.17487/RFC3501, March 2003, <https://www.rfc-editor.org/info/rfc3501>.

[RFC7489] Kucherawy, M., Ed. and E. Zwicky, Ed., "Domain-based Message Authentication, Reporting, and Conformance (DMARC)", RFC 7489, DOI 10.17487/RFC7489, March 2015, <https://www.rfc-editor.org/info/rfc7489>.

[XCLIENT] Postfix, "Postfix XCLIENT Howto", 2019, <http://www.postfix.org/XCLIENT_README.html>.

## Authors' Addresses

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

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

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

Email: chris.newman@oracle.com