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

Abstract

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

Status of This Memo

   This Internet-Draft is submitted in full conformance with the
   provisions of BCP 78 and BCP 79.

   Internet-Drafts are working documents of the Internet Engineering
   Task Force (IETF). Note that other groups may also distribute
   working documents as Internet-Drafts. The list of current Internet-
   Drafts is at https://datatracker.ietf.org/drafts/current/.

   Internet-Drafts are draft documents valid for a maximum of six months
   and may be updated, replaced, or obsoleted by other documents at any
   time. It is inappropriate to use Internet-Drafts as reference
   material or to cite them other than as "work in progress."

   This Internet-Draft will expire on September 19, 2019.

Table of Contents

1. Introduction

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

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

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

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

1.1. Notational conventions

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

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

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

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

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

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

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

o "String" - The JSON string type.

   o "Number" - The JSON number type.

   o "Boolean" - The JSON boolean type.

   o "A[B]" - A JSON object where the keys are all of type "A", and the
      values are all of type "B".

   o "A[]" - An array of values of type "A".

   o "A|B" - The value is either of type "A" or of type "B".

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

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

   o *server-set*: Only the server can set the value for this property.
      The client MUST NOT send this property when creating a new object
      of this type.

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

   o *default*: (This is followed by a JSON value). The value that
      will be used for this property if it is omitted in an argument, or
      when creating a new object of this type.

1.2. The Id data type

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

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

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

   o Ids starting with a dash

   o Ids starting with digits

o Ids that contain only digits

o Ids that differ only by ASCII case (for example, A vs. a)

o the specific sequence of three characters "NIL" (because this
   sequence can be confused with the IMAP protocol expression of the
   null value)

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

1.3. The Int and UnsignedInt data types

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

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

1.4. The Date and UTCDate data types

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

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

1.5. JSON as the data encoding format

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

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

[1.6](). Terminology

[1.6.1](). User

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

[1.6.2](). Accounts

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

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

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

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

[1.6.3](). Data types and records

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

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

1.7. The JMAP API model

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

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

   1. The client may make an API request to the server to get or set
       structured data. This request consists of an ordered series of
       method calls. These are processed by the server, which then
       returns an ordered series of responses. This is described in
       sections 3 to 5.

   2. The client may download or upload binary files from/to the
       server. This is detailed in section 6.

   3. The client may connect to a push channel on the server, to be
       notified when data has changed. This is explained in section 7.

1.8. Vendor-specific extensions

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

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

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

2. The JMAP Session resource

   You need two things to connect to a JMAP server:

   1. The URL for the JMAP Session resource. This may be requested
       directly from the user, or discovered automatically based on a
       username domain (see section 2.2 below).

   2. Credentials to authenticate with. How to obtain credentials is
       out of scope for this document.

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

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

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

      The client MUST ignore any properties it does not understand.

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

      * *maxSizeUpload*: "UnsignedInt" The maximum file size, in
         octets, that the server will accept for a single file upload
         (for any purpose). Suggested minimum: 50,000,000.

      * *maxConcurrentUpload*: "UnsignedInt" The maximum number of
         concurrent requests the server will accept to the upload
         endpoint. Suggested minimum: 4.

      * *maxSizeRequest*: "UnsignedInt" The maximum size, in octets,
         that the server will accept for a single request to the API
         endpoint. Suggested minimum: 10,000,000.

      * *maxConcurrentRequests*: "UnsignedInt" The maximum number of
         concurrent requests the server will accept to the API endpoint.
         Suggested minimum: 4.

* *maxCallsInRequest*: "UnsignedInt" The maximum number of method
  calls the server will accept in a single request to the API
  endpoint. Suggested minimum: 16.

* *maxObjectsInGet*: "UnsignedInt" The maximum number of objects
  that the client may request in a single "/get" type method
  call. Suggested minimum: 500

* *maxObjectsInSet*: "UnsignedInt" The maximum number of objects
  the client may send to create, update or destroy in a single
  "/set" type method call. This is the combined total, e.g. if
  the maximum is 10 you could not create 7 objects and destroy 6,
  as this would be 13 actions, which exceeds the limit.
  Suggested minimum: 500.

* *collationAlgorithms*: "String[]" A list of identifiers for
  algorithms registered in the collation registry defined in
  [RFC4790] that the server supports for sorting when querying
  records.

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

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

o *accounts*: "Id[Account]" A map of *account id* to Account object
  for each account (see section 1.5.2) the user has access to. An
  Account object has the following properties:

  * *name*: "String" A user-friendly string to show when presenting
    content from this account, e.g. the email address representing
    the owner of the account.

  * *isPersonal*: "Boolean" This is "true" if the account belongs
    to the authenticated user, rather than a group account or a
    personal account of another user that has been shared with
    them.

  * *isReadOnly*: "Boolean" This is "true" if the entire account is
    read-only.

  * *accountCapabilities*: "String[Object]" The set of capability
    URIs for the methods supported in this account. Each key is a
    URI for a capability that has methods you can use with this

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

The client MUST ignore any properties it does not understand.

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

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

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

o *primaryAccounts*: "String[Id]" A map of capability URIs (as found
   in _accountCapabilities_) to the account id to be considered the
   user's main or default account for data pertaining to that
   capability. If no account being returned belongs to the user, or
   in any other way there is no appropriate way to determine a
   default account, there MAY be no entry for a particular URI, even
   though that capability is supported by the server (and in the
   capabilities object). "urn:ietf:params:jmap:core" SHOULD NOT be
   present.

o *username*: "String" The username associated with the given
   credentials, or the empty string if none.

o *apiUrl*: "String" The URL to use for JMAP API requests.

o *downloadUrl*: "String" The URL endpoint to use when downloading
   files, in [RFC6570] URI Template (level 1) format. The URL MUST
   contain variables called "accountId", "blobId", "type" and "name".
   The use of these variables is described in section 6.2. Due to

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

o *uploadUrl*: "String" The URL endpoint to use when uploading
   files, in [RFC6570] URI Template (level 1) format. The URL MUST
   contain a variable called "accountId". The use of this variable
   is described in section 6.1.

o *eventSourceUrl*: "String" The URL to connect to for push events,
   as described in section 7.3, in [RFC6570] URI Template (level 1)
   format. The URL MUST contain variables called "types",
   "closeafter" and "ping". The use of these variables is described
   in section 7.3.

o *state*: "String" A (preferably short) string representing the
   state of this object on the server. If the value of any other
   property on the session object changes, this string will change.
   The current value is also returned on the API Response object (see
   section 3.3), allowing clients to quickly determine if the session
   information has changed (e.g. an account has been added or
   removed) and so they need to refetch the object.

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

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

2.1. Example

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

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

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

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

2.2. Service autodiscovery

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

o *DNS SRV* ([RFC2782], [RFC6186] and [RFC6764])

o *.well-known/servicename* ([RFC5785])

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

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

3. Structured data exchange

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

3.1. Making an API request

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

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

3.1.1. The Invocation data type

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

   1. A "String" *name* of the method to call or of the response.

   2. A "String[*]" object containing _named_ *arguments* for that
       method or response.

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

3.2. The Request object

   A *Request* object has the following properties:

   o *using*: "String[]" The set of capabilities the client wishes to
      use. The client MAY include capability identifiers even if the
      method calls it makes do not utilise those capabilities. The
      server advertises the set of specifications it supports in the
      JMAP Session object, as keys on the _capabilities_ property.

   o *methodCalls*: "Invocation[]" An array of method calls to process
      on the server. The method calls MUST be processed sequentially,
      in order.

   o *createdIds*: "Id[Id]" (optional) A map of (client-specified)
      creation id to the id the server assigned when a record was
      successfully created.

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

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

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

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

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

3.2.1. Example request

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

3.3. The Response object

A *Response* object has the following properties:

o *methodResponses*: "Invocation[]" An array of responses, in the
   same format as the _methodCalls_ on the request object. The
   output of the methods MUST be added to the _methodResponses_ array
   in the same order as the methods are processed.

o *createdIds*: "Id[Id]" (optional; only returned if given in
   request) A map of (client-specified) creation id to the id the
   server assigned when a record was successfully created. This MUST

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

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

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

3.3.1. Example response:

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

3.4. Omitting arguments

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

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

3.5. Errors

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

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

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

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

3.5.1. Request-level errors

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

   The following problem types are defined:

   o "urn:ietf:params:jmap:error:unknownCapability" The client included
      a capability in the "using" property of the request that the
      server does not support.

   o "urn:ietf:params:jmap:error:notJSON" The content type of the
      request was not "application/json" or the request did not parse as
      I-JSON.

   o "urn:ietf:params:jmap:error:notRequest" The request parsed as JSON
      but did not match the type signature of the Request object.

   o "urn:ietf:params:jmap:error:limit" The request was not processed
      as it would have exceeded one of the *request* limits defined on
      the capability object, such as maxSizeRequest, maxCallsInRequest
      or maxConcurrentRequests. A "limit" property MUST also be present
      on the "problem details" object, containing the name of the limit
      being applied.

3.5.1.1. Example

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

Another example:

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

3.5.2. Method-level errors

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

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

An "error" response looks like this:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.6. **References to previous method results**

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

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

   A *ResultReference* object has the following properties:

   o *resultOf*: "String" The method call id of the method call to get
      the result from (the string given as the third item in the array
      for a method call).

   o *name*: "String" The expected name of the response.

   o *path*: "String" A pointer into the arguments. This is an
      [RFC6901] JSON Pointer, except it also allows the use of "*" to
      map through an array (see description below).

   To resolve:

   1. Find the first response with a method call id identical to the
       resultOf property of the _ResultReference_ in the
       methodResponses array from previously processed method calls in
       the same request. If none, evaluation fails.

   2. If the response name is not identical to the _name_ property of
       the _ResultReference_, evaluation fails.

   3. Apply the _path_ to the arguments object of the response (the
       second item in the response array) following the [RFC6901] JSON
       Pointer algorithm, except with the following addition in section
       4 (Evaluation):

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

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

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

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

After executing the first method call the _methodResponses_ array is:

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

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

1. Find the first response with method call id "t0". The Foo/
    changes response fulfils this criterion.

2. Check the response name is the same as in the result reference.
    It is, so this is fine.

3. Apply the _path_ as a JSON pointer to the arguments object. This
    simply selects the "created" property, so the result of
    evaluating is: "["f1", "f4"]"

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

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

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

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

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

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

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

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

   2. "Thread/get" is the name specified in the result reference, so
       this is fine.

   3. Apply the _path_ as a JSON pointer to the arguments object.
       Token-by-token:

       1. "list": get the array of thread objects

       2. "*": for each of the items in the array:

           1. "emailIds": get the array of email ids

           2. Concatenate these into a single array of all the ids in
               the result.

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

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

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

3.7. Localisation of user-visible strings

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

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

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

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

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

```
Content-Language: de
```

3.8. Security

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

3.9. Concurrency

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

4. The Core/echo method

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

4.1. Example

Request:

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

Response:

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

5. Standard methods and naming convention

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

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

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

5.1. /get

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

It takes the following arguments:

o *accountId*: "Id" The id of the account to use.

o *ids*: "Id[]|null" The ids of the Foo objects to return. If
   "null" then *all* records of the data type are returned, if this
   is supported for that data type and the number of records does not
   exceed the _maxObjectsInGet_ limit.

o *properties*: "String[]|null" If supplied, only the properties
   listed in the array are returned for each Foo object. If "null",
   all properties of the object are returned. The id property of the
   object is *always* returned, even if not explicitly requested. If
   an invalid property is requested, the call MUST be rejected with
   an "invalidArguments" error.

The response has the following arguments:

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

o *state*: "String" A (preferably short) string representing the
   state on the server for *all* the data of this type in the account
   (not just the objects returned in this call). If the data
   changes, this string MUST change. If the Foo data is unchanged,
   servers SHOULD return the same state string on subsequent requests
   for this data type. When a client receives a response with a
   different state string to a previous call, it MUST either throw
   away all currently cached objects for the type, or call _Foo/
   changes_ to get the exact changes.

o *list*: "Foo[]" An array of the Foo objects requested. This is
   the *empty array* if no objects were found, or if the _ids_
   argument passed in was also the empty array. The results MAY be
   in a different order to the _ids_ in the request arguments. If an
   identical id is included more than once in the request, the server
   MUST only include it once in either the _list_ or _notFound_
   argument of the response.

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

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

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

5.2. /changes

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

   o *accountId*: "Id" The id of the account to use.

   o *sinceState*: "String" The current state of the client. This is
      the string that was returned as the _state_ argument in the _Foo/
      get_ response. The server will return the changes that have
      occurred since this state.

   o *maxChanges*: "UnsignedInt|null" The maximum number of ids to
      return in the response. The server MAY choose to return fewer
      than this value, but MUST NOT return more. If not given by the
      client, the server may choose how many to return. If supplied by
      the client, the value MUST be a positive integer greater than 0.
      If a value outside of this range is given, the server MUST reject
      the call with an "invalidArguments" error.

   The response has the following arguments:

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

   o *oldState*: "String" This is the _sinceState_ argument echoed
      back; the state from which the server is returning changes.

   o *newState*: "String" This is the state the client will be in after
      applying the set of changes to the old state.

o *hasMoreChanges*: "Boolean" If "true", the client may call _Foo/
   changes_ again with the _newState_ returned to get further
   updates. If "false", _newState_ is the current server state.

o *created*: "Id[]" An array of ids for records which have been
   created since the old state.

o *updated*: "Id[]" An array of ids for records which have been
   updated since the old state.

o *destroyed*: "Id[]" An array of ids for records which have been
   destroyed since the old state.

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

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

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

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

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

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

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

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

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

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

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

                        state: "E"
                        hasMoreChanges: false

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

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

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

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

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

5.3. /set

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

   The _Foo/set_ method takes the following arguments:

   o *accountId*: "Id" The id of the account to use.

   o *ifInState*: "String|null" This is a state string as returned by
      the _Foo/get_ method (representing the state of all objects of
      this type in the account). If supplied, the string must match the
      current state, otherwise the method will be aborted and a
      "stateMismatch" error returned. If "null", any changes will be
      applied to the current state.

   o *create*: "Id[Foo]|null" A map of _creation id_ (a temporary id
      set by the client) to Foo objects, or "null" if no objects are to
      be created.

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

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

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

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

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

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

* All parts prior to the last (i.e. the value after the final
   slash) MUST already exist on the object being patched.

* There MUST NOT be two patches in the PatchObject where the
   pointer of one is the prefix of the pointer of the other, e.g.
   "alerts/1/offset" and "alerts".

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

* If "null", set to the default value if specified for this
   property, otherwise remove the property from the patched
   object. If the key is not present in the parent, this a no-op.

* Anything else: The value to set for this property (this may be
   a replacement or addition to the object being patched).

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

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

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

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

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

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

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

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

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

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

The response has the following arguments:

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

o *oldState*: "String|null" The state string that would have been
   returned by _Foo/get_ before making the requested changes, or
   "null" if the server doesn't know what the previous state string
   was.

o *newState*: "String" The state string that will now be returned by
   Foo/get.

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

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

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

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

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

o *destroyed*: "Id[]|null" A list of Foo ids for records that were
   successfully destroyed, or "null" if none.

o *notCreated*: "Id[SetError]|null" A map of creation id to a
   SetError object for each record that failed to be created, or
   "null" if all successful.

o *notUpdated*: "Id[SetError]|null" A map of Foo id to a SetError
   object for each record that failed to be updated, or "null" if all
   successful.

o *notDestroyed*: "Id[SetError]|null" A map of Foo id to a SetError
   object for each record that failed to be destroyed, or "null" if
   all successful.

A *SetError* object has the following properties:

o *type*: "String" The type of error.

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

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

o "forbidden": (create; update; destroy) The create/update/destroy
   would violate an ACL or other permissions policy.

o "overQuota": (create; update) The create would exceed a server-
   defined limit on the number or total size of objects of this type.

o "tooLarge": (create; update) The create/update would result in an
   object that exceeds a server-defined limit for the maximum size of
   a single object of this type.

o "rateLimit": (create) Too many objects of this type have been
   created recently, and a server-defined rate limit has been
   reached. It may work if tried again later.

o "notFound": (update; destroy) The id given to update/destroy
   cannot be found.

o "invalidPatch": (update) The PatchObject given to update the
   record was not a valid patch (see the patch description).

o "willDestroy" (update) The client requested an object be both
   updated and destroyed in the same /set request, and the server has
   decided to therefore ignore the update.

o "invalidProperties": (create; update) The record given is invalid
   in some way. For example:

   * It contains properties which are invalid according to the type
      specification of this record type.

   * It contains a property that may only be set by the server (e.g.
      "id") and is different to the current value. Note, to allow
      clients to pass whole objects back, it is not an error to
      include a server-set property in an update so long as the value
      is identical to the current value on the server.

   * There is a reference to another record (foreign key) and the
      given id does not correspond to a valid record.

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

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

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

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

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

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

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

5.4. /copy

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

The copy is conceptually in three phases:

1. Reading the current values from the "from" account.

2. Writing the new copies to the other account.

3. Destroying the originals in the "from" account, if requested.

Data may change in between phases due to concurrent requests.

The _Foo/copy_ method takes the following arguments:

o *fromAccountId*: "Id" The id of the account to copy records from.

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

o *accountId*: "Id" The id of the account to copy records to. This
   MUST be different to the "fromAccountId".

o *ifInState*: "String|null" This is a state string as returned by
   the _Foo/get_ method. If supplied, the string must match the
   current state of the account referenced by the accountId,
   otherwise the method will be aborted and a "stateMismatch" error
   returned. If "null", any changes will be applied to the current
   state.

o *create*: "Id[Foo]" A map of _creation id_ to a Foo object. The
   object MUST contain an id property: the id (in the fromAccount) of
   the record to be copied. Any other properties included are used
   instead of the current value for that property on the original
   when creating the copy.

o *onSuccessDestroyOriginal*: "Boolean" (default: false) If "true",
   an attempt will be made to destroy the original records that were
   successfully copied: after emitting the _Foo/copy_ response, but
   before processing the next method, the server MUST make a single
   call to _Foo/set_ to destroy the original of each successfully
   copied record; the output of this is added to the responses as
   normal to be returned to the client.

o *destroyFromIfInState*: "String|null" This argument is passed on
   as the "ifInState" argument to the implicit _Foo/set_ call, if
   made at the end of this request to destroy the originals that were
   successfully copied.

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

The response has the following arguments:

o *fromAccountId*: "Id" The id of the account records were copied
   from.

o *accountId*: "Id" The id of the account records were copied to.

o *oldState*: "String|null" The state string that would have been
   returned by _Foo/get_ on the account records were copied to before
   making the requested changes, or "null" if the server doesn't know
   what the previous state string was.

o *newState*: "String" The state string that will now be returned by
   Foo/get on the account records were copied to.

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

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

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

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

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

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

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

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

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

5.5. /query

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

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

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

A call to _Foo/query_ takes the following arguments:

o *accountId*: "Id" The id of the account to use.

o *filter*: "FilterOperator|FilterCondition|null" Determines the set
   of Foos returned in the results. If "null", all objects in the
   account of this type are included in the results. A
   FilterOperator object has the following properties:

   * *operator*: "String" This MUST be one of the following strings:
      "AND" / "OR" / "NOT":

      + *AND*: all of the conditions must match for the filter to
         match.

      + *OR*: at least one of the conditions must match for the
         filter to match.

      + *NOT*: none of the conditions must match for the filter to
         match.

   * *conditions*: "(FilterOperator|FilterCondition)[]" The
      conditions to evaluate against each record.

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

o *sort*: "Comparator[]|null" Lists the names of properties to
   compare between two Foo records, and how to compare them, to
   determine which comes first in the sort. If two Foo records have
   an identical value for the first comparator, the next comparator
   will be considered and so on. If all comparators are the same
   (this includes the case where an empty array or "null" is given as
   the _sort_ argument), the sort order is server-dependent, but MUST
   be stable between calls to Foo/query. A *Comparator* has the
   following properties:

   * *property*: "String" The name of the property on the Foo
      objects to compare.

   * *isAscending*: "Boolean" (optional; default: true) If "true",
      sort in ascending order. If "false", reverse the comparator's
      results to sort in descending order.

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

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

  1. It MUST be unicode-aware.

  2. It MAY be selected based on an Accept-Language header in
      the request (as defined in [RFC7231] section 5.3.5), or
      out-of-band information about the user's language/locale.

  3. It SHOULD be case-insensitive where such a concept makes
      sense for a language/locale. Where the user's language is
      unknown, it is RECOMMENDED to follow the advice in section
      5.2.3 of [RFC8264].

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

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

  + "Boolean": "false" comes before "true".

  + "Number": A lower number comes before a higher number.

  + "Date"/"UTCDate": The earlier date comes first.

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

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

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

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

o *anchor*: "Id|null" A Foo id. If supplied the _position_ argument
   is ignored. The index of this id in the results will be used in
   combination with the "anchorOffset" argument to determine the
   index of the first result to return (see below for more details).

o *anchorOffset*: "Int" (default: 0) The index of the first result
   to return relative to the index of the anchor, if an anchor is
   given. This MAY be negative. For example, "-1" means the Foo
   immediately preceding the anchor is the first result in the list
   returned (see below for more details).

o *limit*: "UnsignedInt|null" The maximum number of results to
   return. If "null", no limit presumed. The server MAY choose to
   enforce a maximum "limit" argument. In this case, if a greater
   value is given (or if it is "null"), the limit is clamped to the
   maximum; the new limit is returned with the response so the client
   is aware. If a negative value is given, the call MUST be rejected
   with an "invalidArguments" error.

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

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

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

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

The response has the following arguments:

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

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

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

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

o *canCalculateChanges*: "Boolean" This is "true" if the server
   supports calling _Foo/queryChanges_ with these "filter"/"sort"
   parameters. Note, this does not guarantee that the _Foo/
   queryChanges_ call will succeed, as it may only be possible for a
   limited time afterwards due to server internal implementation
   details.

o *position*: "UnsignedInt" The 0-based index of the first result in
   the "ids" array within the complete list of query results.

o *ids*: "Id[]" The list of ids for each foo in the query results,
   starting at the index given by the _position_ argument of this
   response, and continuing until it hits the end of the results or
   reaches the "limit" number of ids. If _position_ is >= _total_,
   this MUST be the empty list.

o *total*: "UnsignedInt" (only if requested) The total number of
   foos in the results (given the _filter_). This argument MUST be
   omitted if the _calculateTotal_ request argument is not "true".

o *limit*: "UnsignedInt" (if set by the server) The limit enforced
   by the server on the maximum number of results to return. This is
   only returned if the server set a limit, or used a different limit
   to that given in the request.

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

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

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

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

5.6. /queryChanges

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

o *accountId*: "Id" The id of the account to use.

o *filter*: "FilterOperator|FilterCondition|null" The filter argument that was used with _Foo/query_.

o *sort*: "Comparator[]|null" The sort argument that was used with _Foo/query_.

o *sinceQueryState*: "String" The current state of the query in the client. This is the string that was returned as the _queryState_ argument in the _Foo/query_ response with the same sort/filter. The server will return the changes made to the query since this state.

o *maxChanges*: "UnsignedInt|null" The maximum number of changes to return in the response. See error descriptions below for more details.

o *upToId*: "Id|null" The last (highest-index) id the client currently has cached from the query results. When there are a large number of results, in a common case the client may have only downloaded and cached a small subset from the beginning of the results. If the sort and filter are both only on immutable properties, this allows the server to omit changes after this point in the results, which can significantly increase efficiency. If they are not immutable, this argument is ignored.

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

The response has the following arguments:

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

o *oldQueryState*: "String" This is the "sinceQueryState" argument
   echoed back; the state from which the server is returning changes.

o *newQueryState*: "String" This is the state the query will be in
   after applying the set of changes to the old state.

o *total*: "UnsignedInt" (only if requested) The total number of
   foos in the results (given the _filter_). This argument MUST be
   omitted if the _calculateTotal_ request argument is not "true".

o *removed*: "Id[]" The _id_ for every foo that was in the query
   results in the old state and is not in the results in the new
   state.

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

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

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

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

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

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

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

     An *AddedItem* object has the following properties:

     * *id*: "Id"

     * *index*: "UnsignedInt"

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

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

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

      removed = ["id2", "id31", ...];
      fooIds => ["id1", null, null, "id3", "id4", null, null, null]

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

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

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

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

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

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

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

5.7. Examples

   Suppose we have a type _Todo_ with the following properties:

   o *id*: "Id" (immutable; server-set) The id of the object.

   o *title*: "String" A brief summary of what is to be done.

   o *keywords*: "String[Boolean]" (default: {}) A set of keywords that
      apply to the todo. The set is represented as an object, with the
      keys being the _keywords_. The value for each key in the object
      MUST be "true". (This format allows you to update an individual
      key using patch syntax rather than having to update the whole set
      of keywords as one, which an "String[]" representation would
      require.)

   o *neuralNetworkTimeEstimation*: "Number" (server-set) The title and
      keywords are fed into the server's state-of-the-art neural network
      to get an estimation of how long this todo will take, in seconds.

   o *subTodoIds*: "Id[]|null" The ids of a list of subtodos to
      complete as part of this todo.

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

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

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

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

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

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

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

or it may send a minimal patch:

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

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

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

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

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

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

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

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

and receive in response:

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

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

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

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

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

5.8. Proxy considerations

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

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

   1. It must pass a "createdIds" property with each subrequest. If
       this is not given by the client, an empty object should be used
       for the first subrequest. The "createdIds" property of each
       subresponse should be passed on in the next subrequest.

   2. It must resolve back-references to previous method results that
       were processed on a different server. This is a relatively
       simple syntactic substitution, described in section 3.6.

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

6. Binary data

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

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

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

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

   o The server SHOULD use a separate quota for unreferenced blobs to
      the accounts's usual quota. This quota SHOULD be separate per
      user in the case of shared accounts.

   o This quota SHOULD be at least the maximum total size that a single
      object can reference on this server. For example, if supporting
      JMAP Mail, this should be at least the maximum total attachments
      size for a message.

   o When an upload would take the user over quota, the server MUST
      delete unreferenced blobs in date order, oldest first, until there
      is room for the new blob.

   o Except where quota restrictions force early deletion, an
      unreferenced blob MUST NOT be deleted for at least 1 hour from the
      time of upload; if reuploaded, the same blobId MAY be returned,
      but this SHOULD reset the expiry time.

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

6.1. Uploading binary data

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

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

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

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

o *blobId*: "Id", The id representing the binary data uploaded. The
   data for this id is immutable. The id _only_ refers to the binary
   data, not any metadata.

o *type*: "String" The media type of the file (as specified in
   [RFC6838], section 4.2) as set in the Content-Type header of the
   upload HTTP request.

o *size*: "UnsignedInt" The size of the file in octets.

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

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

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

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

6.2. Downloading binary data

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

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

o "accountId": The id of the account to which the record with the
   blobId belongs.

o "blobId": The blobId representing the data of the file to
   download.

o "type": The type for the server to set in the "Content-Type"
   header of the response; the blobId only represents the binary data
   and does not have a content-type innately associated with it.

o "name": The name for the file; the server MUST return this as the
   filename if it sets a "Content-Disposition" header.

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

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

6.3. Blob/copy

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

The _Blob/copy_ method takes the following arguments:

o *fromAccountId*: "Id" The id of the account to copy blobs from.

o *accountId*: "Id" The id of the account to copy blobs to.

o *blobIds*: "Id[]" A list of ids of blobs to copy to the other
   account.

The response has the following arguments:

   o *fromAccountId*: "Id" The id of the account blobs were copied
      from.

   o *accountId*: "Id" The id of the account blobs were copied to.

   o *copied*: "Id[Id]|null" A map of the blobId in the _fromAccount_
      to the id for the blob in the account it was copied to, or "null"
      if none were successfully copied.

   o *notCopied*: "Id[SetError]|null" A map of blobId to a SetError
      object for each blob that failed to be copied, "null" if none.

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

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

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

7. **Push**

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

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

7.1. The StateChange object

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

   o *@type*: "String" This MUST be the string "StateChange".

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

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

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

7.1.1. Example

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

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

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

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

7.2. PushSubscription

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

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

A *PushSubscription* object has the following properties:

o *id*: "Id" (immutable; server-set) The id of the push
   subscription.

o *deviceClientId*: "String" (immutable) An id that uniquely
   identifies the client + device it is running on. The purpose of
   this is to allow clients to identify which PushSubscription
   objects they created even if they lose their local state, so they
   can revoke or update them. This string MUST be different on
   different devices, and be different from apps from other vendors.
   It SHOULD be easy to re-generate, not depend on persisted state.
   It is RECOMMENDED to use a secure hash of a string that contains:

   1. A unique identifier associated with the device where the JMAP
       client is running, normally supplied by the device's operating
       system.

   2. A custom vendor/app id, including a domain controlled by the
       vendor of the JMAP client.

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

o *url*: "String" (immutable) An absolute URL where the JMAP server
   will POST the data for the push message. This MUST begin with
   "https://".

o *keys*: "Object|null" (immutable) Client-generated encryption
   keys. If supplied the server MUST use them as specified in
   [RFC8291] to encrypt all data sent to the push subscription. The
   object MUST have the following properties:

   * *p256dh*: the P-256 ECDH Diffie-Hellman public key as described
      in [RFC8291], encoded in URL-safe Base64 representation as
      defined in [RFC4648].

   * *auth*: the authentication secret as described in [RFC8291],
      encoded in URL-safe Base64 representation as defined in
      [RFC4648].

o *verificationCode*: "String|null" This MUST be "null" (or omitted)
   when the subscription is created. The JMAP server then generates
   a verification code and sends it in a push message, and the client
   updates the PushSubscription object with the code; see section
   7.2.2 for details.

o *expires*: "UTCDate|null" The time this push subscription expires.
   If specified, the JMAP server MUST NOT make further requests to
   this resource after this time. It MAY automatically destroy the
   push subscription at or after this time.

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

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

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

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

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

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

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

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

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

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

7.2.1. PushSubscription/get

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

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

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

7.2.2. PushSubscription/set

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

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

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

o *@type*: "String" This MUST be the string "PushVerification".

o *pushSubscriptionId*: "String" The id of the push subscription
   that was created.

o *verificationCode*: "String" The verification code to add to the
   push subscription. This MUST contain sufficient entropy to avoid
   the client being able to brute force guess the code.

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

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

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

7.2.3. Example

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

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

Which returns:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

7.3. Event Source

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

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

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

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

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

   o "types": This MUST be either:

   *   A comma-separated list of type names, e.g.
       "Email,CalendarEvent". The server MUST only push changes for
       the types in this list.

   *   The single character: "*". Changes to all types are pushed.

   o   "closeafter": This MUST be one of the following values:

   *   "state": The server MUST end the HTTP response after pushing a
       state event. This can be used by clients in environments where
       buffering proxies prevent the pushed data from arriving
       immediately, or indeed at all, when operating in the usual
       mode.

   *   "no": The connection is persisted by the server as a standard
       event-source resource.

   o   "ping": A positive integer value representing a length of time in
       seconds, e.g. "300". If non-zero, the server MUST send an event
       called "ping" whenever this time elapses since the previous event
       was sent. This MUST NOT set a new event id. If the value is "0"
       the server MUST NOT send ping events.

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

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

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

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

8. Security considerations

8.1. Transport confidentiality

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

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

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

8.2. Authentication scheme

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

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

8.3. Service autodiscovery

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

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

8.4. JSON parsing

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

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

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

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

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

8.5. Denial of service

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

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

8.6. Connection to unknown push server

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

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

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

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

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

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

8.7. Push encryption

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

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

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

8.8. Traffic analysis

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

9. IANA considerations

9.1. Assignment of jmap service name

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

Service Name: jmap

Transport Protocol(s): tcp

Assignee: IESG

Contact: IETF Chair

Description: JSON Meta Application Protocol

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

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

9.2. Registration of well-known URI suffix for JMAP

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

URI Suffix: jmap

Change Controller: IETF

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

9.3. Registration of the jmap URN sub-namespace

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

Registered Parameter Identifier: jmap

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

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

9.4. Creation of "JMAP Capabilities" registry

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

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

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

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

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

9.4.1. Preliminary community review

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

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

9.4.2. Submit request to IANA

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

9.4.3. Designated expert review

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

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

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

9.4.4. Change procedures

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

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

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

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

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

9.4.5. JMAP Capabilities registry template:

Capability name: (see capability property in section 2)

Specification document:

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

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

Security and privacy considerations:

9.4.6. Initial registration for JMAP core

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

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

Intended use: common

Change Controller: IETF

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

9.4.7. Registration for JMAP error placeholder in JMAP capabilities registry

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

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

Intended use: placeholder

Change Controller: IETF

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

9.5. Creation of "JMAP Error Codes" registry

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

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

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

9.5.1. Designated expert review

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

1. Verify the error code does not conflict with existing names.

2. Verify the error code follows the syntax limitations (does not
    require URI encoding).

3. Encourage the error code to follow the naming convention of
    previously registered errors.

4. Encourage description of client behaviors that are recommended in
    response to the error code. These may distinguish the error code
    from other error codes.

5. Encourage description of when the server should issue the error
    as opposed to some other error code.

6. Encourage the submitter to note any security considerations
    associated with the error, if any. For example, an error code
    that might disclose existence of data the authenticated user does
    not have permission to know about.

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

9.5.2. JMAP Error Codes registry template:

JMAP Error Code:

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

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

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

Description:

9.5.3. Initial JMAP Error Codes registry

JMAP Error Code	Intended Use	Change Controller	Reference	Description
accountNotFound	common	IETF	[I-D.ietf-jmap-core] section 3.5.2	The accountId does not correspond to a valid account.
accountNotSupportedByMethod	common	IETF	[I-D.ietf-jmap-core] section 3.5.2	The accountId given corresponds to a valid account, but the account does not support this method or data type.
accountReadOnly	common	IETF	[I-D.ietf-jmap-core] section 3.5.2	This method call would modify state in an account that is read-only (as returned on the corresponding Account object in the JMAP Session

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

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

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

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

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

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

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

10. References

10.1. Normative References

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

[I-D.ietf-jmap-core]
          Jenkins, N. and C. Newman, "JSON Meta Application
          Protocol", draft-ietf-jmap-core-16 (work in progress),
          March 2019.

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

[RFC2782] Gulbrandsen, A., Vixie, P., and L. Esibov, "A DNS RR for
          specifying the location of services (DNS SRV)", RFC 2782,
          DOI 10.17487/RFC2782, February 2000,
          <https://www.rfc-editor.org/info/rfc2782>.

[RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818,
          DOI 10.17487/RFC2818, May 2000,
          <https://www.rfc-editor.org/info/rfc2818>.

[RFC3339] Klyne, G. and C. Newman, "Date and Time on the Internet:
          Timestamps", RFC 3339, DOI 10.17487/RFC3339, July 2002,
          <https://www.rfc-editor.org/info/rfc3339>.

[RFC3553] Mealling, M., Masinter, L., Hardie, T., and G. Klyne, "An
          IETF URN Sub-namespace for Registered Protocol
          Parameters", BCP 73, RFC 3553, DOI 10.17487/RFC3553, June
          2003, <https://www.rfc-editor.org/info/rfc3553>.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO
          10646", STD 63, RFC 3629, DOI 10.17487/RFC3629, November
          2003, <https://www.rfc-editor.org/info/rfc3629>.

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

[RFC4790] Newman, C., Duerst, M., and A. Gulbrandsen, "Internet
          Application Protocol Collation Registry", RFC 4790,
          DOI 10.17487/RFC4790, March 2007,
          <https://www.rfc-editor.org/info/rfc4790>.

[RFC5051] Crispin, M., "i;unicode-casemap - Simple Unicode Collation
          Algorithm", RFC 5051, DOI 10.17487/RFC5051, October 2007,
          <https://www.rfc-editor.org/info/rfc5051>.

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

   [RFC5785] Nottingham, M. and E. Hammer-Lahav, "Defining Well-Known
              Uniform Resource Identifiers (URIs)", RFC 5785,
              DOI 10.17487/RFC5785, April 2010,
              <https://www.rfc-editor.org/info/rfc5785>.

   [RFC6186] Daboo, C., "Use of SRV Records for Locating Email
              Submission/Access Services", RFC 6186,
              DOI 10.17487/RFC6186, March 2011,
              <https://www.rfc-editor.org/info/rfc6186>.

   [RFC6335] Cotton, M., Eggert, L., Touch, J., Westerlund, M., and S.
              Cheshire, "Internet Assigned Numbers Authority (IANA)
              Procedures for the Management of the Service Name and
              Transport Protocol Port Number Registry", BCP 165,
              RFC 6335, DOI 10.17487/RFC6335, August 2011,
              <https://www.rfc-editor.org/info/rfc6335>.

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

   [RFC6749] Hardt, D., Ed., "The OAuth 2.0 Authorization Framework",
              RFC 6749, DOI 10.17487/RFC6749, October 2012,
              <https://www.rfc-editor.org/info/rfc6749>.

   [RFC6764] Daboo, C., "Locating Services for Calendaring Extensions
              to WebDAV (CalDAV) and vCard Extensions to WebDAV
              (CardDAV)", RFC 6764, DOI 10.17487/RFC6764, February 2013,
              <https://www.rfc-editor.org/info/rfc6764>.

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

   [RFC6901] Bryan, P., Ed., Zyp, K., and M. Nottingham, Ed.,
              "JavaScript Object Notation (JSON) Pointer", RFC 6901,
              DOI 10.17487/RFC6901, April 2013,
              <https://www.rfc-editor.org/info/rfc6901>.

   [RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer
              Protocol (HTTP/1.1): Message Syntax and Routing",
              RFC 7230, DOI 10.17487/RFC7230, June 2014,
              <https://www.rfc-editor.org/info/rfc7230>.

   [RFC7231] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer
              Protocol (HTTP/1.1): Semantics and Content", RFC 7231,
              DOI 10.17487/RFC7231, June 2014,
              <https://www.rfc-editor.org/info/rfc7231>.

   [RFC7493] Bray, T., Ed., "The I-JSON Message Format", RFC 7493,
              DOI 10.17487/RFC7493, March 2015,
              <https://www.rfc-editor.org/info/rfc7493>.

   [RFC7525] Sheffer, Y., Holz, R., and P. Saint-Andre,
              "Recommendations for Secure Use of Transport Layer
              Security (TLS) and Datagram Transport Layer Security
              (DTLS)", BCP 195, RFC 7525, DOI 10.17487/RFC7525, May
              2015, <https://www.rfc-editor.org/info/rfc7525>.

   [RFC7617] Reschke, J., "The 'Basic' HTTP Authentication Scheme",
              RFC 7617, DOI 10.17487/RFC7617, September 2015,
              <https://www.rfc-editor.org/info/rfc7617>.

   [RFC7807] Nottingham, M. and E. Wilde, "Problem Details for HTTP
              APIs", RFC 7807, DOI 10.17487/RFC7807, March 2016,
              <https://www.rfc-editor.org/info/rfc7807>.

   [RFC8030] Thomson, M., Damaggio, E., and B. Raymor, Ed., "Generic
              Event Delivery Using HTTP Push", RFC 8030,
              DOI 10.17487/RFC8030, December 2016,
              <https://www.rfc-editor.org/info/rfc8030>.

   [RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for
              Writing an IANA Considerations Section in RFCs", BCP 26,
              RFC 8126, DOI 10.17487/RFC8126, June 2017,
              <https://www.rfc-editor.org/info/rfc8126>.

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

   [RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data
              Interchange Format", STD 90, RFC 8259,
              DOI 10.17487/RFC8259, December 2017,
              <https://www.rfc-editor.org/info/rfc8259>.

   [RFC8264] Saint-Andre, P. and M. Blanchet, "PRECIS Framework:
              Preparation, Enforcement, and Comparison of
              Internationalized Strings in Application Protocols",
              RFC 8264, DOI 10.17487/RFC8264, October 2017,
              <https://www.rfc-editor.org/info/rfc8264>.

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

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

10.2. Informative References

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

Authors' Addresses

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

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

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

   Email: chris.newman@oracle.com