JMAP   N. Jenkins
Internet-Draft   FastMail
Intended status: Standards Track   March 5, 2018
Expires: September 6, 2018

JSON Meta Application Protocol
draft-ietf-jmap-core-04

Abstract

This document specifies a protocol for synchronising JSON-based data objects efficiently, with support for push and out-of-band binary data upload/download.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

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

This specification 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 the separate end points for users after login, the separation of binary and structured data, and a shared data model 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", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

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 [RFC7159]. 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.

Types signatures are given for all JSON objects in this document. The following conventions are used:

- "Boolean|String" - The value is either a JSON "Boolean" value, or a JSON "String" value.

- "Foo" - Any name that is not a native JSON type means an object for which the properties (and their types) are defined elsewhere within this document.

- "Foo[]" - An array of objects of type "Foo".

- "String[Foo]" - A JSON "Object" being used as a map (associative array), where all the values are of type "Foo".

1.2. The Number datatype

The JSON datatypes are limited to those found in JavaScript. A "Number" in JavaScript is represented as a signed double (64-bit floating point). However, except where explicitly specified, all numbers used in this API are unsigned integers <= 2^53 (the maximum integer that may be reliably stored in a double).

1.3. The Date datatypes

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

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

1.4. JSON as the data encoding format

JSON is a text-based data interchange format as specified in [RFC7159]. 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 properties with the same key).

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]).

1.5. Terminology

1.5.1. User

A user represents a set of permissions relating to what data can be seen.

1.5.2. 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 operations in JMAP are isolated to a single account; there are a few explicit operations to copy data between them. Certain properties are guaranteed for data within the same account, for example uniqueness of ids within a type in that account.

An account is not the same as a user, although it is common for the 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. Users may also have access to accounts belonging to another user if that user is sharing some of their data. A single set of credentials may provide access to data in multiple accounts.

1.5.3. 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*.

1.6. Ids

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

Ids are always "String"s. An id MUST be a valid UTF-8 string of at least 1 character in length and maximum 256 octets in size, but MUST NOT start with the "#" character, as this is reserved for doing back references during object creation (see the _/set_ description).

1.7. The JMAP API model

JMAP uses HTTP [RFC7230] to expose API, Push, Upload and Download resources. Implementations MUST support HTTP/1.1, and MAY support later versions. Support for common HTTP mechanisms such as redirection and caching are assumed.

All HTTP requests MUST be authenticated. Servers MUST conform with the [RFC7235] HTTP Authentication framework to reject requests that fail authentication and inform the client of available authentication schemes.

Clients SHOULD understand and be able to handle standard HTTP status codes appropriately.

An authenticated client can fetch the 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 and 4.

2. The client may download or upload binary files from/to the server. This is detailed in section 5.

3. The client may connect to a push channel on the server, to be notified when data has changed. This is explained in section 6.

2. The JMAP Session resource

To communicate with a JMAP server you need two things to start:

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 Service Autodiscovery section below).

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

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:

- *username*: "String" The username associated with the given credentials.

- *accounts*: "String[Account]" A map of *account id* to Account object for each account the user has access to. A single set of credentials may provide access to multiple accounts, for example if another user is sharing their mail with the logged in user, or if there is an account that contains data for a group or business. All data belongs to a single account. With the exception of a few explicit operations to copy data between accounts, all JMAP methods take an _accountId_ argument that specifies on which account the operations are to take place. This argument is always optional; if not specified, the primary account is used. All ids (other than account ids of course) are only unique within their account. 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. 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. 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.

  - *isPrimary*: "Boolean" This MUST be true for *at most* one of the accounts returned. This is to be considered the user's main or default account by the client. If no account being returned belongs to the user, or in any other way there is no appropriate way to determine a default account, then this MAY be "false" for all accounts.

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

  - *hasDataFor*: "String[]" A list of the data profiles available in this account. Each future JMAP data types specification will define a profile name to encompass that set of types.

- *capabilities*: "String[Object]" An object specifying the capabilities of this server. Each key is a URI for a specification 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 specification. The client MUST ignore any properties it does not understand. The capabilities object MUST include a property called "ietf:jmap". The value of this property is an object which MUST contain the following information on server capabilities:

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

* *maxConcurrentUpload*: "Number" The maximum number of concurrent requests the server will accept to the upload endpoint.

* *maxSizeRequest*: "Number" The maximum size, in octets, that the server will accept for a single request to the API endpoint.

* *maxConcurrentRequests*: "Number" The maximum number of concurrent requests the server will accept to the API endpoint.

* *maxCallsInRequest*: "Number" The maximum number of method calls the server will accept in a single request to the API endpoint. This MUST be greater than or equal to "32" to ensure clients can rely on the ability to make efficient network use.

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

* *maxObjectsInSet*: "Number" The maximum number of objects the client may send to create, update or destroy in a single "/set" type method call.

* *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.

Future specifications will define their own properties on the capabilities object. Servers MAY advertise vendor-specific JMAP extensions. To avoid conflict, the identifiers for these MUST be a URI beginning with a domain owned by the vendor. Clients MUST opt in to any specifications it wishes to use (see "Making an API request").

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

- *downloadUrl*: "String" The URL endpoint to use when downloading files (see the Download section of this spec), in [RFC6570] URI Template (level 1) format. The URL MUST contain variables called "blobId", MAY contain a variables called "accountId" and SHOULD contain a variable called "name".

- *uploadUrl*: "String" The URL endpoint to use when uploading files (see the Upload section of this spec), in [RFC6570] URI Template (level 1) format. The URL MAY contain a variable called "accountId".

- *eventSourceUrl*: "String" The URL to connect to for push events (see the Push section of this spec).

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

2.1. Service Autodiscovery

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

- *DNS srv* ([RFC6186] and [RFC6764])

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

A JMAP-supporting host for the domain "example.com" SHOULD publish a SRV record "_jmaps._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, the location of which may be found on the JMAP Session object.

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

3.2. The Request object

A *Request* object has the following properties:

- *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.

- *methodCalls*: "Array[]" An array of method calls to process on the server. The method calls MUST be processed sequentially, in order. A *method call* is represented by an array containing three elements:

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

  2. An "Object" containing _named_ *arguments* for that method.

  3. A *client id*: an arbitrary "String" 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 client id in the response).

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

3.2.1. Example request

```
{
  "using": [ "ietf.org/rfc/jmap-core", "ietf.org/rfc/jmap-mail" ],
  "methodCalls": [
    ["method1", {"arg1": "arg1data", "arg2": "arg2data"}, "#1"],
    ["method2", {"arg1": "arg1data"}, "#2"],
    ["method3", {}, "#3"]
  ]
}
```

3.3. Vendor-specific extensions

Individual services will have custom features they wish to expose over JMAP. This may take the form of extra datatypes 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. 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.

3.4. The Response object

A *Response* object has the following properties:

o *methodResponses*: "Array[]" 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.

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

3.4.1. Example response:

```
            {
              "methodResponses": [
                ["method1", {"arg1": 3, "arg2": "foo"}, "#1"],
                ["method2", {"isBlah": true}, "#2"],
                ["anotherResponseFromMethod2", {
                  "data": 10,
                  "yetmoredata": "Hello"
                }, "#2"],
                ["error", {"type":"unknownMethod"}, "#3"]
              ]
            }
```

3.5. 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.

3.6. Errors

3.6.1. Request-level errors

If the data sent as an API request is not valid JSON or does not match the structure above, or includes a capability that the server does not support in the "using" property of the request, a "400 Bad Request" error will be returned at the HTTP level. The body of the response SHOULD include a short description of the problem to help client developers debug the issue.

3.6.2. 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.

An "error" response looks like this:

```
                          ["error", {
                            type: "unknownMethod"
                          }, "client-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 "serverError", 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:

"serverError": An unexpected or unknown error occured during the processing of the call. The state of the server after such an error is undefined.

"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.

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

"timedOut": The method failed to execute because it timed out waiting for a lock, or was taking too much compute time.

"accountNotFound": An _accountId_ was included with the method call that does not correspond to a valid account.

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

"accountReadOnly": This method call would modify state in an account that has "isReadOnly == true".

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 "serverError" type.

3.7. 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.

To do this, the client prefixes the argument name with "#". 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 back reference should be resolved and the value used as the "real" argument. The method is then processed as normal. If any back reference fails to resolve, the whole method MUST be rejected with a "resultReference" 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 client 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 client 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", {
                       "sinceState": "abcdef"
                   }, "t0" ],
                   [ "Foo/get", {
                       "#ids": {
                           "resultOf": "t0",
                           "name": "Foo/changes",
                           "path": "/changed"
                       }
                   }, "t1" ]]
```

After executing the first method call the _methodResponses_ array is:

```
                   [[ "Foo/changes", {
                       "accountId": "1",
                       "oldState": "abcdef",
                       "newState": "123456",
                       "hasMoreChanges": false,
                       "changed": [ "f1", "f4" ],
                       "destroyed": []
                   }, "t0" ]]
```

So 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 client 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 "changed" 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:

```
                          {
                              "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", {
       "filter": { inMailbox: "id_of_inbox" },
       "sort": [{ property: "receivedAt", isAscending: false }],
       "collapseThreads": true,
       "position": 0,
       "limit": 10
     }, "t0" ],
     [ "Email/get", {
       "#ids": {
         "resultOf": "t0",
         "name": "Email/query",
         "path": "/ids"
       },
       "properties": [ "threadId" ]
     }, "t1" ],
     [ "Thread/get", {
       "#ids": {
         "resultOf": "t1",
         "name": "Email/get",
         "path": "/list/*/threadId"
       }
     }, "t2" ],
     [ "Email/get", {
       "#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": "1",
    "filter": { inMailbox: "id_of_inbox" },
    "sort": [{ property: "receivedAt", isAscending: false }],
    "collapseThreads": true,
    "state": "abcdefg",
    "canCalculateChanges": true,
    "position": 0,
    "total": 101,
    "ids": [ "msg1023", "msg223", "msg110", "msg93", "msg91", "msg38", "msg36",
"msg33", "msg11", "msg1" ]
}, "t0" ],
[ "Email/get", {
    "accountId": "1",
    "state": "123456",
    "list": [{
        "id": "msg1023",
        "threadId": "trd194",
    }, {
        "id": "msg223",
        "threadId": "trd114"
    },
    ...
    ],
    "notFound": null
}, "t1" ],
[ "Thread/get", {
    "accountId": "1",
    "state": "123456",
    "list": [{
        "id: "trd194",
        "emailIds": [ "msg1020", "msg1021", "msg1023" ]
    }, {
        "id: "trd114",
        "emailIds": [ "msg201", "msg223" ]
    },
    ...
    ],
    "notFound": null
}, "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 client 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: a) "list": get the array of thread objects b) "*": for each of the items in the array:

i) `emailIds`: get the array of email ids
ii) 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:

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

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

Each individual method call within a request MUST be serializable; concurrent execution of methods MUST produce the same effect as running them one at a time in some order.

This means that the observable ordering may interleave method calls from different concurrent API requests, such that the data on the server may change between two method calls within a single API request.

4. 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.

4.1. /get

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

It takes the following arguments:

- o *accountId*: "String|null" The id of the Account to use. If "null", the primary account is used.

- o *ids*: "String[]|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.

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

- o *state*: "String" A 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*: "String[]|null" This array contains the ids passed to the method for records that do not exist. This property is "null" 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.

4.2. /changes

When the state of the set of Foo records 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*: "String|null" The id of the Account to use. If "null", the primary account is used.

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*: "Number|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*: "String" 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 *changed*: "String[]|null" An array of ids for records which have been created or modified but not destroyed since the oldState, or "null" if none.

o *destroyed*: "String[]|null" An array of ids for records which have been destroyed since the old state, or "null" if none.

If a _maxChanges_ is supplied, or set automatically by the server, the server MUST ensure the number of ids returned across _changed_ 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 update returned 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.

If a Foo record has been modified AND destroyed since the oldState, the server SHOULD just return the id in the _destroyed_ list, but MAY return it in the _changed_ list as well. If a Foo record has been created AND destroyed since the oldState, the server SHOULD remove the id from the response entirely, but MAY include it in the _destroyed_ list.

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).

4.3. /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*: "String|null" The id of the Account to use. If "null", the primary account is used.

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, otherwise the method will be aborted and a "stateMismatch" error returned. If "null", any changes will be applied to the current state.

o *create*: "String[Foo]|null" A map of _creation id_ (an arbitrary string 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*: "String[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*: "String[]|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 is not permissible to 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 record objects may hold references to others (foreign keys). When records are created or modified, they may reference other records being created _in the same API request_ by using the 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 (a series of method

calls), 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.

Creation ids are scoped by type; a separate "creation id -> id" map MUST be kept for each type for the duration of the request. Foreign key references are always for a particular record type, so use of the same creation key in two different types cannot cause any ambiguity. Creation ids sent by the client SHOULD be unique within the single API request for a particular data type. If a creation id is reused for the same type, the server MUST map the creation id to the most recently created item with that id.

The response has the following arguments:

o *accountId*: "String" 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*: "String[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*: "String[Foo|null]|null" The _keys_ in this map are the ids of all Foos that were successfully updated, or "null" if none successful. 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.

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

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

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

- *notDestroyed*: "String[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:

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

- *description*: "String|null" A description of the error to display to the user.

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

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

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

- "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.

- "notFound": (update; destroy) The id given cannot be found.

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

- "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.

- "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 are 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 so long as the value is identical

to the current value on the server (or the value that will be set by the server if a create).

   * 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.

- "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.

4.4. /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:

- *accountId*: "String|null" The id of the Account to use. If "null", the primary account is used.

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 email.

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.

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 SHOULD have reasonable default behavior for many languages when the user's language is unknown.

3. It MAY be selected based on out-of-band information about the user's language/locale.

4. It SHOULD be case-insensitive where such a concept makes sense for a language/locale.

The "i;unicode-casemap" collation ([RFC5051]) and the Unicode Collation Algorithm (<http://www.unicode.org/reports/tr10/>) are two examples that fulfil these criterion. 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.

- *position*: "Number" (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.

- *anchor*: "String|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).

- *anchorOffset*: "Number|null" The index of the anchor object relative to the index of the first result to return. This MAY be negative. For example, "-1" means the first Foo after the anchor Foo should be the first result in the results returned (see below for more details).

- *limit*: "Number|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 should be clamped to the maximum; since the total number of results in the search is returned, the client can determine if it has received all the results. If a negative value is given, the call MUST be rejected with an "invalidArguments" error.

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 subtracted from 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 _anchorOffset_ is "null", it defaults to "0". 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*: "String" The id of the account used for the call.

o *filter*: "FilterOperator|FilterCondition|null" The filter to apply to the search. Echoed back from the call.

o *sort*: "Comparator[]|null" The sort options used. Echoed back from the call.

o *state*: "String" A string encoding the current state on the server. This string MUST change if the results of the search _may_ have changed (for example, there has been a change to the state of the set of Foos; it does not _guarantee_ that anything in the search has changed). It may be passed to _Foo/queryChanges_ to efficiently get the set of changes from the client's current state. Should a client receive back a response with a different state string to a previous call, it MUST either throw away the currently cached search 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 delta 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.

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

- *total*: "Number" The total number of foos in the results (given the _filter_).

- *ids*: "String[]" The list of ids for each foo in the search 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.

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 search.

"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.

4.5. /queryChanges

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

- *accountId*: "String|null" The id of the account to use for this call. If "null", the primary account will be used.

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

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

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

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

o *uptoId*: "String|null" The last (highest-index) id the client currently has cached from the search 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.

The response has the following arguments:

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

o *filter*: "FilterOperator|FilterCondition|null" The filter to apply to the search. Echoed back from the call.

o *sort*: "Comparator[]|null" The sort options used. Echoed back from 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 *uptoId*: "String|null" Echoed back from the call.

o *total*: "Number" The total number of foos in the results (given the _filter_).

o *removed*: "String[]" The _id_ for every foo that was in the results in the old state and is not in the results in the new state. 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 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 _filter_ or _sort_ includes a mutable property, the server MUST include all foos in the current results for which this property MAY have changed.

o *added*: "AddedItem[]" The id and index in the search 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*: "String"

* *index*: "Number"

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

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

then if it *splices out* all foos in the removed array:

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

and *splices in* (in order) all of the foos 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.

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

"tooManyChanges": There are more changes the 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 search results.

"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. The client MUST invalidate its cache of the search results.

4.6. Examples

Suppose we have a type _Todo_ with the following properties:

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

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

- *keywords*: "String[Boolean]" (mutable; 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".

- *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.

and the server supports querying by keyword using the syntax "{ hasKeyword: "foo" }" in the _filter_ argument to _/query_.

Now, a client might want to display the list of todos with a particular query, so it makes the following method call:

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

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

```
       [["Todo/query", {
         "accountId": "x",
         "filter": { "hasKeyword": "music" },
         "sort": [{ "property": "title" }],
         "state": "y13213",
         "canCalculateChanges": true,
         "position": 0,
         "total": 26,
         "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": "Listen to Daft Punk",
           "keywords": {
             "music": 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", {
                   "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", {
                        "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.

Now, suppose another user deleted the "Listen to Daft Punk" todo. The first user will receive a push notification (see later in the spec) 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", {
                 "filter": { "hasKeyword": "music" },
                 "sort": [{ "property": "title" }],
                 "sinceState": "y13213"
                 "maxChanges": 50,
               }, "1"]]
```

and receive in response:

```
               [["Todo/changes", {
                 "accountId": "x",
                 "oldState": "10324",
                 "newState": "871903",
                 "hasMoreChanges": false,
                 "changed": null,
                 "destroyed": ["b"]
               }, "0"],
               ["Todo/queryChanges", {
                 "filter": { "hasKeyword": "music" },
                 "sort": [{ "property": "title" }],
                 "oldState": "y13213"
                 "newState": "y13218"
                 "total": 25,
                 "removed": ["b"],
                 "added": null
               }, "1"]]
```

5. Binary data

Binary data is referenced by a _blobId_ in JMAP, and uploaded/ downloaded separately to the core API. A blobId does not have a name inherent to it, but this is normally given in the same object that contains the blobId. 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 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 from 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 user's usual quota.

- 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 SHOULD NOT be deleted for at least 24h 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.

5.1. 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 MAY contain a variable called "accountId". The client may use this template in combination with an _accountId_

(if required in the template) 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:

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

- *blobId*: "String", 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.

- *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, with CFWS collapsed to SP and [RFC2231]/[RFC2047] encoding removed.

- *size*: "Number" 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.

5.2. Downloading binary data

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

The client may use this template in combination with an _accountId_ (if required in the URL template) and _blobId_ to download any binary data (files) referenced by other objects. Since a blob is not associated with a particular name, the template SHOULD allow a name to be substituted in as well; the server will return this as the filename if it sets a "Content-Disposition" header.

To download the data the client makes an authenticated GET request to the download URL with the appropriate variables substituted in. The client SHOULD send an "Accept" header with the content type they would like the server to return for the file. The "Content-Type" header of a successful response SHOULD be set to the type as requested in the "Accept" header by the client, or "application/octet-stream" if unknown and no "Accept" header given.

5.3. Blob/copy

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

The _Blob/copy_ method takes the following arguments:

- *fromAccountId*: "String|null" The id of the account to copy blobs from. If "null", defaults to the primary account.

- *toAccountId*: "String|null" The id of the account to copy blobs to. If "null", defaults to the primary account.

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

The response has the following arguments:

- *fromAccountId*: "String" The id of the account emails were copied from.

- *toAccountId*: "String" The id of the account emails were copied to.

- *copied*: "String[String]|null" A map of the blob id in the _fromAccount_ to the id for the blob in the _toAccount_, or "null" if none were successfully copied.

- *notCopied*: "String[SetError]|null" A map of blob id 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_.

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

"fromAccountNotFound": A _fromAccountId_ was explicitly included with the request, but it does not correspond to a valid account.

"toAccountNotFound": A _toAccountId_ was explicitly included with the request, but it does not correspond to a valid account.

6. Push

Push notifications allow clients to efficiently update (almost) instantly to stay in sync with data changes on the server. In JMAP, push notifications occur out-of-band (i.e. not over the same connection as API exchanges), so that they can make use of efficient native push mechanisms on different platforms.

The general model for push is simple and sends minimal data over the push channel. 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; it will still get all changes next time it syncs.

6.1. 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 *changed*: "String[TypeState]" A map of _account id_ to an object encoding the state of data types that have changed for that account since the last push event, 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).

o *trigger*: "String" What caused this change. The following causes are defined:

  * "delivery": The arrival of a new message caused the change.

  * "user": An action by the user caused the change.

  * "unknown": The cause of the change is unknown.

  Future specifications may define further values. Clients MUST treat an unrecognised value the same as "unknown". Clients in battery constrained environments may use this information to decide whether to immediately fetch the changes.

6.2. PushSubscription

A push subscription is a message delivery context established between the client and a push service. A *PushSubscription* object has the following properties:

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

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 remove the push subscription at or after this time.

o *keys*: "Object|null" 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].

Clients may register the push subscription with the JMAP server, which will then make a POST request to the associated push endpoint whenever an event occurs.

The POST request MUST have a content type of "application/json" and contain the utf-8 JSON encoded _StateChange_ 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].

If the response code is "503" (Service Unavailable), the JMAP server MAY try again later, but may also just drop the event. If the response code is "429" (Too Many Requests) the JMAP server SHOULD attempt to reduce the frequency of pushes to that URL. Any other "4xx" or "5xx" response code MUST be considered a *permanent failure* and the push subscription should be deregistered (not tried again even for future events unless explicitly re-registered by the client).

The use of this push endpoint conforms with the use of a push endpoint by an Application Server as defined in [RFC8030]. A client MAY use the rest of [RFC8030] in combination with its own Push Server

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.

6.2.1. PushSubscription/set

Each session may only have a single push subscription registered. The push subscription is tied to the access token used to create it. Should the access token expire or be revoked, the push subscription MUST be removed by the JMAP server. The client MUST re-register the push subscription after reauthenticating to resume callbacks.

To set the push subscription, make a call to _PushSubscription/set_. It takes the following argument:

o *pushSubscription*: "PushSubscription|null" The PushSubscription object representing the endpoint the JMAP server will POST events to. This will replace any previously set subscription. Set to "null" to remove any previously registered subscription.

The response has no arguments.

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

"invalidUrl": Returned if the URL does not begin with "https://", or is otherwise syntactically invalid or does not resolve.

"forbidden": Returned if the URL is valid, but for policy reasons the server is not willing to connect to it.

6.2.2. PushSubscription/get

To check the currently set push subscription (if any), make a call to _PushSubscription/set_. It does not take any arguments. The response has a single argument:

o *pushSubscription*: "PushSubscription|null" The PushSubscription object the JMAP server is currently posting push events to, or "null" if none.

6.3. Event Source

Clients that can hold open TCP connections can connect directly to the JMAP server to receive push notifications via a "text/event-stream" resource, as described in <http://www.w3.org/TR/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 [1] will send a Last-Event-ID HTTP header 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 client MAY add a query parameter called "closeafter" with value "state" to the event-source resource URL when requesting the event-source resource. If set, 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.

The client MAY add a query parameter called "ping", with a positive integer value representing a length of time in seconds, e.g. "ping=300". If set, 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.

The server MAY modify the interval given as a query parameter 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 "Number") 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.

Refer to the JMAP Session resource section of this spec for details on how to get the URL for the event-source resource. Requests to the resource MUST be authenticated.

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

7. Security considerations

7.1. Transport confidentiality

All HTTP requests MUST use [RFC5246] TLS (https) transport to ensure the confidentiality of data sent and received via JMAP. Clients MUST validate TLS certificate chains to protect against man-in-the-middle attacks.

7.2. 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.

If offering the Basic authentication scheme, services are strongly recommended to not allow a user's regular password but 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.

7.3. Service autodiscovery

Unless secured by something like DNSSEC, 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.

7.4. JSON parsing

The security considerations of [RFC7159] apply to the use of JSON as the data interchange format.

7.5. 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.

7.6. 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, 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.

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

8. References

8.1. Normative References

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

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

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

[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>.

[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>.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <https://www.rfc-editor.org/info/rfc5246>.

[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>.

[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>.

[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>.

[RFC7159] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 7159, DOI 10.17487/RFC7159, March 2014, <https://www.rfc-editor.org/info/rfc7159>.

[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>.

[RFC7235] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Authentication", RFC 7235, DOI 10.17487/RFC7235, June 2014, <https://www.rfc-editor.org/info/rfc7235>.

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

[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>.

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

8.2. URIs

[1] https://html.spec.whatwg.org/multipage/server-sent-events.html

Author's Address

Neil Jenkins
FastMail
Level 2, 114 William St
Melbourne VIC 3000
Australia

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