JMAP
Internet-Draft
Intended status: Standards Track
Expires: April 22, 2017

N. Jenkins
FastMail

October 19, 2016

JSON Meta Application Protocol
draft-jenkins-jmap-00

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

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 http://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 April 22, 2017.

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 authentication and 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].

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. Terminology

1.2.1. User

A user represents a set of permissions relating to what data can be seen. To access data in JMAP, you first authenticate as a particular user.

1.2.2. Accounts

An account is a collection of data.

All data, other than the Account objects themselves, belong to a single account. A single account may contain an arbitrary set of data, 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.

1.2.3. 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*.

1.3. Ids

All object ids are assigned by the server, and are immutable. They MUST be unique among all objects of the *same type* within the *same account*. Ids may clash across accounts, or for two objects 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 bytes in size, but MUST NOT start with the "#" character, as this is reserved for doing back references during object creation (see the _setFoos_ description).

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.

1.5. The JMAP API model

All data exchanges are authenticated using an access token. Authentication is covered in section 2.

An authenticated client may exchange data with the server using four different mechanisms:

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 section 3.

2. The client may download binary files from the server. This is detailed in section 4.

3. The client may upload binary files to the server. This is specified in section 5.

4. 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. Authentication

When connecting to any JMAP server, the client must first gain an access token. It cannot just use a username/password directly. This allows the server to know (and show the user) which clients currently have access to the account, and to be able to revoke access individually.

The server may support multiple different mechanisms for authenticating a user to gain the access token. It is expected that further types may be added in future extensions to the JMAP specification.

2.1. Service autodiscovery

To begin authentication, the client needs to know the authentication URL for the JMAP server.

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

o *DNS srv* See [RFC6186] and [RFC6764]

- *.well-known/servicename* See [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 authentication URL 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.

To support clients that are unable to do SRV lookups, the server SHOULD make the _hostname_ the same domain as the username if possible.

2.2. Getting an access token

Authorization always starts with the client making a POST request to the authentication URL (found either via service autodiscovery or manual entry). The request MUST be of type "application/json" and specify an "Accept: application/json" header. The body of the request MUST be a single JSON object, encoded in UTF-8, with the following properties:

- *username*: "String" The username the client wishes to authenticate. This is normally the primary email address of the user.

- *clientName*: "String" The name of the client software. e.g. "Mozilla Thunderbird".

- *clientVersion*: "String" Information to identify the version of the client. This MUST change for any changed client code (e.g. a version control tag or counter for development software) and SHOULD sort lexically later for newer versions.

- *deviceName*: "String" A human-friendly string to identify the device making the request, e.g. "Joe Blogg's iPhone".

The server may use the client/device information to help identify the login to the user in a login log or other security reporting. Although hopefully unnecessary, they may also be helpful for working around client bugs in the future.

The server will respond with one of the following HTTP status codes:

2.2.1. 200: Success, but more authorization required.

The response body will be a single JSON object with the following properties.

- *loginId*: "String" An id from the server to allow it to connect the next request with previous requests in the login process. This SHOULD be of limited time validity (e.g. 15 minutes from previous call).

- *methods*: "AuthMethod[]" A list of the supported authentication methods to continue with authentication. See below for definition of the *AuthMethod* object.

- *prompt*: "String|null" A message to display in the client to the user. The client MUST treat this as plain text, but SHOULD automatically hyperlink any URLs it finds if a system browser is available.

This is the standard response to an initial request. Note, a server may return this even if the username is not actually active, to prevent enumeration. The client should then pick one of the _methods_ from the list in the response to continue with authentication (if no methods supported by the client are in the list, it will not be able to get an access token).

An *AuthMethod* object MUST have a *type* property. This is a "String" representing the method of authentication. For some types, there may be other values required on the AuthMethod object in addition; see the description of types below. The following types are currently defined, but more may be added in the future. A client SHOULD offer the user the option to choose any of the method types returned that the client supports. The client MUST ignore any types it does not understand:

- "external": The user must do something out-of-band to authorize the app. The server SHOULD use the _prompt_ property of the response to tell the user what they need to do. A client that supports the _external_ authorisation type MUST offer a mechanism for the user to indicate to the client when they have completed the out-of-band authentication.

- "oauth": OAuth based authentication. For OAuth integration, see the docs of the service in question, since every service implements it slightly differently and the client must register with the service beforehand to use it. If using this method, an access token is obtained entirely through the OAuth mechanism from this point on, and requests will be authenticated as per the OAuth

spec. See the "Refetching URL endpoints" section below for how to obtain the URL endpoints after successfully authenticating using OAuth.

- "password": The user must input their current password for the account.

- "totp": The user must input a TOTP [1] code from a device registered with the account.

- "yubikeyotp": The user must input a Yubico OTP [2] code from a Yubikey registered with the account.

- "u2f": The user must sign a challenge using a FIDO U2F [3] key registered with the account. The AuthMethod object for this type MUST also have the following properties:

  * *appId*: "String" The app id to use.

  * *signChallenge*: "String" The challenge to be signed by the U2F key.

  * *registeredKeys*: "RegisteredKey[]" The keys associated with this user. A *RegisteredKey* object has the following properties:

  * *version*: "String" The U2F protocol version.

  * *keyHandle*: "String" The key handle of the registered key.

- "sms": The user must input a one-time code sent via SMS to a phone number registered with the account. The AuthMethod object for this type MUST also have the following property:

  * *phoneNumbers*: "LoginPhone[]|null" An array of *LoginPhone* objects, each of which represents a phone registered with the account. A *LoginPhone* object has the following properties:

    + *id*: "String" The id of the phone. This is used when asking the server to send a code.

    + *number*: "String" The phone number to display to the user to remind them which number the SMS will be/was sent to. This MAY have some characters replaced with an "X" or other "blanked-out" character if the server does not wish to disclose the full phone number at this point. e.g. if the phone registered with the account is "+61 123 456 789", the server might return "+61 1XX XXX X89" as the number.

+ *isCodeSent*: "Boolean" Has the verification code been sent to this number yet? The server MAY choose to send the SMS before the first time this auth option is returned, or may wait for the user to explicitly request it.

If not using ""oauth"", the user will at some point indicate to the client to continue authentication (after inputing any required token/code/password dependent on the auth method chosed). At this point the client submits a POST request to the same URL as before, with the body being a single JSON object with the following properties:

o *loginId*: "String" The _loginId_ the server sent from the previous request.

o *type*: "String" The type of the method chosen to continue authentication.

o *value*: "*" The value as appropriate for the given type:

  * "external": "null"

  * "password"/"totp"/"yubikeyotp"/"sms": "String" - the password/ one-time code.

  * "u2f": "SignResponse" - an object with _keyHandle_, _signatureData_ and _clientData_ "String" properties, as defined in the U2F spec.

Note: The client SHOULD NOT store any password/code the user has entered beyond what is required to submit it to the server in this step.

The server will then return one of the same set of responses as before, which should be handled the same (for example, if two-factor authentication is required, a "200" response may be returned again and TOTP/U2F prompted for).

If the user chooses to authenticate using SMS, they may need to request the server to send the code to a particular number before they can submit a code. To do this, the client submits a POST request to the same URL as before, with the body being a single JSON object with the following properties:

o *loginId*: "String" The _loginId_ the server sent from the previous request.

- *sendCodeTo*: "String" The id of the phone number to send the code to.

The server SHOULD send the code to the given phone if the phone id is valid. If the code has already been sent, it is server-dependent whether it is sent again or ignored. The server MUST return one of the same set of responses as before, which should be handled the same (in most cases this will be a "200" response identical to before except that the _isCodeSent_ property for the phone will now be "true").

2.2.2. 201: Authentication is complete, access token created.

The response body will be a single JSON object with the following properties.

- *username*: "String" The username that was successfully authenticated.

- *accessToken*: "String" The secret token to be used by the client to authenticate all future JMAP requests. The client should keep this secure, preferably in an OS keychain or the like. Since tokens should not be reused across devices or clients, the client SHOULD NOT reveal this token to the user.

- *accounts*: "String[Account]" A map of *account id* to Account object for each account the user has access to. A single access token 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 exactly one of the accounts returned. This is to be considered the user's main or default account by the client.

  * *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. The keys are URIs, which specify the specifications supported by the server. The value for each of these keys is an object that MAY include further information about the server's capabilities in relation to that spec. The client MUST ignore any properties it does not understand. The capabilities object MUST include a property called "{TODO: URI for this spec}". The value of this property is an object which SHOULD contain the following information on server capabilities:

  * *maxSizeUpload*: "Number" The maximum file size, in bytes, 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 bytes, 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.

  * *maxObjectsInGet*: "Number" The maximum number of obje ts that the client may request in a single "getFoos" type method call.

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

  Future specifications will define their own properties on the capabilities object.

- *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 a variable called "blobId". The URL SHOULD contain a variable called "name".

- *uploadUrl*: "String" The URL endpoint to use when uploading files (see the Upload section of this spec).

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

URLs are returned only after logging in. This allows different URLs to be used for users located in different geographic datacentres within the same service.

Note, if authentication is done via IP or mobile subscriber ID or some similar mechanism, a "201" response MAY be returned in response to the initial request (with just the username and client info).

2.2.3. 400: Malformed request

The request is of the wrong content type, or does not contain data in the expected format. The client MUST NOT retry the same request. There is no content in the response.

2.2.4. 403: Authentication step failed, but client may try again

Returned in response to a continuation request which failed (e.g. the password entered was not correct, or the out-of-band step was not completed successfully). The response body will be a single JSON object with the same properties as the "200" response, and the client may try again.

2.2.5. 404: Not found

The JMAP authentication server is not available at this address. The client needs to rediscover the authentication URL. There is no content in the response.

2.2.6. 410: Restart authentication

The login attempt has failed permanently. This may be due to a password being incorrect, the login id expiring, or any other reason. The client MUST restart authentication (go back to sending the username and client info to the server). There is no content in the response.

2.2.7. 429: Rate limited

Returned if the server is temporarily blocking this IP/client from authenticating. This may be due to too many failed password attempts, or detected username enumeration attempts, or any other reason. (Legitimate) clients should wait a while then try again. There is no content in the response.

2.2.8. 500: Internal server error

Something has gone wrong internally, and the server is in a broken state. Don't automatically retry. There is no content in the response.

2.2.9. 503: Service unavailable

The server is currently down. Try again later with exponential backoff. There is no content in the response.

2.3. Refetching URL endpoints

A server MAY (although SHOULD NOT) move end points for any services other than authentication at any time. If a request to the API/file upload/event source endpoint returns a "404", the client MUST refetch the URL endpoints. To do this, it should make an authenticated GET request to the authentication URL (see below for how to authenticate requests).

For OAuth logins, this is how the URLs may be fetched initially as well.

The server MUST respond with one of the following status codes:

2.3.1. 201: Authentication is complete, access token created.

The request was successful. The response will be of type "application/json" and consists of a single JSON object containing the following properties:

- *username*: "String" The username that was successfully authenticated.
- *accounts*: "String[Account]" An object representing the accounts the user has access to. See the full description above.
- *capabilities*: "String[Object]" An object specifying the capabilities of this server. See the full description above.
- *apiUrl*: "String" The URL to use for JMAP API requests.
- *downloadUrl*: "String" The URL endpoint to use when downloading files (see above).
- *uploadUrl*: "String" The URL endpoint to use when uploading files (see the Upload section of this spec).
- *eventSourceUrl*: "String" The URL to connect to for push events (see the Push section of this spec).

2.3.2. 403: Restart authentication

The "Authorization" header was missing or did not contain a valid token. Reauthenticate and then retry the request. There is no content in the response.

2.3.3. 404: Not found

The JMAP server is no longer here. There is no content in the response.

2.3.4. 500: Internal server error

Something has gone wrong internally, and the server is in a broken state. Don't automatically retry. There is no content in the response.

2.3.5. 503: Service unavailable

The server is currently down. Try again later with exponential backoff. There is no content in the response.

2.4. Revoking an access token

The validity of an access token is determined by the server. It may be valid for a limited time only, or expire after a certain time of inactivity, or be valid indefinitely etc. If an access token

expires, it MUST NOT be resurrected. The client MUST restart the authentication process to get a new access token.

For OAuth, see the provider's documentation on revoking access tokens.

Otherwise, a client may revoke an access token at any time by making an authenticated DELETE HTTP request to the authentication URL (the one used to get the token in the first place). The response from the server will be one of the following:

2.4.1. 204: Success

The access token has now been revoked. There is no content in the response.

2.4.2. 401: Unauthorized

Failed due to missing "Authorization" header, or the "Authorization" header did not contain a valid access token. As per the HTTP spec, the response MUST have a "WWW-Authenticate: Bearer" header. There is no content in the response.

2.5. Authenticating HTTP requests

All HTTP requests other than to the authentication URL must be authenticated. To do this, the client MUST add an "Authorization" header to each request.

Once authenticated, the client will have an access token. This is used with the "Bearer" scheme as specified in [RFC6750] to authenticate HTTP requests.

For example, if _user@example.com_ successfully logged in and the client received an _accessToken_ of "abcdef1234567890", to authenticate requests you would add the following header:

Authorization: Bearer abcdef1234567890

2.5.1. Signed GET requests

Sometimes, particularly in the browser context, authenticating a GET request using the usual "Authorization" header is not easily implemented. In such situations, a client may use a signed request instead. A signed request is an unauthenticated GET request containing a special query string parameter (a so-called "token"). The process below describes how a client can obtain such a token, then use it to download the file:

1. The client makes an authenticated POST request to the URL for which it is unable to make an authenticated GET request. If authorization is granted, the server MUST send the token back as the body of the response, in one of the following forms:

   * A JSON Web Token [4], in this case the response MUST have a "Content-Type" of "application/jwt".

   * A plain "String" or any other data structure, in this case the response SHOULD have a "Content-Type" of "text/plain" unless another standardised MIME type is applicable.

2. The client makes an unauthenticated GET request to the same URL but with "access_token=<response to the previous request>" appended to the query part (see [RFC3986] section 3.4 for details). If no query part was present in the URL before this operation, a new query part is created. The server MUST send back the contents of the requested file, as if it were requested through a standard authenticated GET request to the URL.

The server SHOULD expire any given signing "token" quickly for obvious security reasons, but the actual expiration policy is up to the server implementation.

Sample HTTP exchange demonstrating the use of a signed request to download a file:

o Obtain a token

```
POST /jmap/download/act1/blob2/mydocument.pdf HTTP/1.1
Host: server.example.com
Authorization: Bearer abcdef1234567890

HTTP/1.1 200 OK
Content-Type: text/plain
Content-Length: 10

abcDEfGhIJ
```

o Download the file

```
GET /jmap/download/act1/blob2/mydocument.pdf?access_token=abcDEfGhIJ HTTP/1.1
Host: server.example.com

HTTP/1.1 200 OK
Content-Type: application/pdf
Content-Length: 987654

<< binary content >>
```

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 URL; see the Authentication section of the spec for how to discover this URL and how to authenticate requests.

The request MUST have a content type of "application/json" and be encoded in UTF-8.

The server SHOULD respond with one of the following HTTP response codes:

3.1.1. 200: OK

The API request was successful. The response will be of type "application/json" and consists of the response to the API calls, as described below.

3.1.2. 400: Bad Request

The request was malformed. For example, it may have had the wrong content type, or have had a JSON object that did not conform to the API calling structure (see _The structure of an API request_ below). The client SHOULD NOT retry the same request. There is no content in the response.

3.1.3. 401: Unauthorized

The "Authorization" header was missing or did not contain a valid token. Reauthenticate and then retry the request. As per the HTTP spec, the response MUST have a "WWW-Authenticate" header listing the

available authentication schemes. There is no content in the response.

3.1.4. 404: Not Found

The API endpoint has moved. See the Authentication section of the spec for how to rediscover the current URL to use. There is no content in the response.

3.1.5. 413: Payload Too Large

Returned if the client makes a request with more method calls than the server is willing to accept in a single request, or if the total bytes of the request is larger than the max size the server is willing to accept.

3.1.6. 429: Rate limited

Returned if the client has made too many requests recently, or has too many concurrent requests currently in progress. Clients SHOULD wait a while then try again. The response MAY include a "Retry-After" header indicating how long to wait before making a new request.

3.1.7. 500: Internal Server Error

Something has gone wrong internally, and the server is in a broken state. Don't automatically retry. There is no content in the response.

3.1.8. 503: Service Unavailable

The server is currently down. Try again later with exponential backoff. There is no content in the response.

3.2. The structure of an API request

The client initiates an API request by sending the server a JSON array. Each element in this array is another array representing a method invocation on the server. The server will process the method calls and return a response consisting of an array in the same format. Each method call always contains three elements:

1. The *name* of the method to call, or the name of the response from the server. This is a "String".

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

3. A *client id*: an arbitrary "String" to be echoed back with the responses emitted by that method call (as we'll see lower down, a method may return 1 or more responses, as some methods make implicit calls to other ones).

Example query:

```
[
  ["method1", {"arg1": "arg1data", "arg2": "arg2data"}, "#1"],
  ["method2", {"arg1": "arg1data"}, "#2"],
  ["method3", {}, "#3"]
]
```

The method calls MUST be processed sequentially, in order. Each API request (which, as shown, may contain multiple method calls) receives a JSON response in exactly the same format. The output of the methods MUST be added to the array in the same order as the methods are processed.

Example response:

```
[
  ["responseFromMethod1", {"arg1": 3, "arg2": "foo"}, "#1"],
  ["responseFromMethod2", {"isBlah": true}, "#2"],
  ["anotherResponseFromMethod2", {
    "data": 10,
    "yetmoredata": "Hello"
  }, "#2"],
  ["aResponseFromMethod3", {}, "#3"]
]
```

3.3. Errors

If the data sent as an API request is not valid JSON or does not match the structure above, an error will be returned at the transport level. For example, when using JMAP over HTTP, a "400 Bad Request" error will be returned at the HTTP level.

Possible errors for each method are specified in the method descriptions. If a method encounters an error, the appropriate "error" response MUST be inserted at the current point in the output 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 has a type property as specified in the method description. Other properties may be present with further information; these are detailed in the method descriptions where appropriate.

Any method MAY return an error of type "serverError" if an unexpected or unknown error occurs during the processing of that call. The state of the server after such an error is undefined.

If an unknown method is called, an "unknownMethod" error (this is the type shown in the example above) MUST be inserted and then the next method call MUST be processed as normal.

If an unknown argument or invalid arguments (wrong type, or in violation of other specified constraints) are supplied to a method, an "invalidArguments" error MUST be inserted and then the next method call MUST be processed as normal.

3.4. 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). To ensure compatibility with clients that don't know about a specific custom extension, and for compatibility with future versions of JMAP, the server MUST ONLY expose these extensions if the client explicitly opts in. Without opt-in, the server MUST follow the spec and reject anything that does not conform to it as specified.

3.5. 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.

3.6. Concurrency

To ensure the client always sees a consistent view of the data, the state accessed by a method call MUST NOT change during the execution of the method, except due to actions by the method call itself. The

state MAY change in-between method calls (even within a single API request).

3.7. 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). This implicitly limits the maximum length of message lists in queries and the like.

3.8. The Date datatypes

Where a JMAP API specifies "Date" as a type, it means a string in [RFC3339] _date-time_ format, with the _time-offset_ component always "Z" (i.e. the date-time MUST be in UTC time) and _time-secfrac_ always omitted. The "T" and "Z" MUST always be upper-case. For example, ""2014-10-30T14:12:00Z"".

3.9. Use of null

Unless otherwise specified, a missing property in the arguments object of a request (from the client), or a response (from the server) MUST be intepreted exactly the same as that property having the value "null".

Unless otherwise specified, a missing property in a data object MUST be interpreted in the following ways: - In the response to a _getFoos_ style call, or when *creating* an object in a _setFoos_ style call, a missing property MUST be interpreted as though it had the default value for that type, or "null" if no default is specified. - When *updating* an object in a _setFoos_ style call, a missing property MUST be interpreted as the existing value for that property (i.e. don't update it).

For network efficiency, when fetching the server and client may make use of the above and omit properties which have the default value for the data type.

3.10. CRUD methods

JMAP provides a uniform interface for creating, retrieving, updating and deleting various types of objects. For a "Foo" data type, records of that type would be fetched via a "getFoos" call and modified via a "setFoos" call. Delta updates may be fetched via a "getFooUpdates" call. These methods all follow a standard format as described below.

3.10.1. getFoos

Objects of type *Foo* are fetched via a call to _getFoos_. Methods with a name starting with "get" MUST NOT alter state on the server.

This method may take some or all of the following arguments; see the definition of the data type in question. However, if one of the following arguments is available, it will behave exactly as specified below.

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

The response to "getFoos" is called "foos". It has the following arguments:

- *accountId*: "String" The id of the account used for the call.
- *state*: "String" A string representing the state on the server for *all* the data of this type. If the data changes, this string will change. It is used to get delta updates, if supported for the type.
- *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.
- *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 error may be returned instead of the "foos" response:

"accountNotFound": Returned if an _accountId_ was explicitly included with the request, but it does not correspond to a valid account.

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

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

"invalidArguments": Returned if one of the arguments is of the wrong type, or otherwise invalid. A "description" property MAY be present on the response object to help debug with an explanation of what the problem was.

3.10.2. getFooUpdates

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 _foos_ response will change. The _getFooUpdates_ call allows a client to efficiently update the state of any its Foo cache to match the new state on the server. It takes the following arguments:

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

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

- *maxChanges*: "Number|null" The maximum number of Foo 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.

- *fetchRecords*: "Boolean|null" If "true", immediately after outputting the _fooUpdates_ response, the server will make an implicit call to _getFoos_ with the _changed_ property of the response as the _ids_ argument. If "false" or "null", no implicit call will be made.

- *fetchRecordProperties*: "String[]|null" If the _getFoos_ method takes a _properties_ argument, this argument is passed through on implicit calls (see the _fetchRecords_ argument).

The response to _getFooUpdates_ is called _fooUpdates_. It has the following arguments:

- *accountId*: "String" The id of the account used for the call.
- *oldState*: "String" This is the _sinceState_ argument echoed back; the state from which the server is returning changes.
- *newState*: "String" This is the state the client will be in after applying the set of changes to the old state.
- *hasMoreUpdates*: "Boolean" If "true", the client may call _getFooUpdates_ again with the _newState_ returned to get further updates. If "false", _newState_ is the current server state.
- *changed*: "String[]" An array of Foo ids for records which have been created or changed but not destroyed since the oldState.
- *removed*: "String[]" An array of Foo ids for records which have been destroyed since the old state.

The _maxChanges_ argument (and _hasMoreUpdates_ response argument) is available for data types with potentially large amounts of data (i.e. those for which there is a _getFooList_ method available for loading the data in pages). If a _maxChanges_ is supplied, or set automatically by the server, the server must try to limit the number of ids across _changed_ and _removed_ to the number given. If there are more changes than this between the client's state and the current server state, the update returned MUST take the client to an intermediate state, from which the client can continue to call _getFooUpdates_ until it is fully up to date. The server MUST NOT return more ids than the _maxChanges_ total. If the server is unable to calculate a suitable intermediate state, it MUST return a "cannotCalculateChanges" error.

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

The following errors may be returned instead of the _fooUpdates_ response:

"accountNotFound": Returned if an _accountId_ was explicitly included with the request, but it does not correspond to a valid account.

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

"invalidArguments": Returned if the request does not include one of the required arguments, or one of the arguments is of the wrong type, or otherwise invalid. A _description_ property MAY be present on the response object to help debug with an explanation of what the problem was.

"cannotCalculateChanges": Returned if 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.

3.10.3. setFoos

Modifying the state of Foo objects on the server is done via the _setFoos_ method. This encompasses creating, updating and destroying Foo records. This has two benefits:

1. It 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).

2. A single call can make all the changes you want to a particular type. If the client wants to use _ifInState_ to guard its changes, it can only make one call that modifies a particular type per request, since it will need the new state following that call to make the next modification.

The _setFoos_ method takes the following arguments:

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

- *ifInState*: "String|null" This is a state string as returned by the _getFoos_ 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.

- *create*: "String[Foo]|null" A map of _creation id_ (an arbitrary string set by the client) to Foo objects (containing all properties except the id, unless otherwise stated in the specific

documentation of the data type). If "null", no objects will be created.

- *update*: "String[Foo]|null" A map of id to Foo objects. The object may omit any property; only properties that have changed need be included. If "null", no objects will be updated.

- *destroy*: "String[]|null" A list of ids for Foo objects to permanently delete. If "null", no objects will be deleted.

Each create, update or destroy is considered an atomic unit. It is permissible for the server to commit some of the changes 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).

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.

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 sent by the client SHOULD be unique within the single API request for a particular data type. If a creation id is reused, the server MUST map the creation id to the most recently created item with that id.

The response to _setFoos_ is called _foosSet_. It has the following arguments:

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

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

- *newState*: "String" The state string that will now be returned by _getFoos_.

- *created*: "String[Foo]" A map of the creation id to an object containing any *server-assigned* properties of the Foo object (including the id) for all successfully created records.

- *updated*: "String[]" A list of Foo ids for records that were successfully updated.

- *destroyed*: "String[]" A list of Foo ids for records that were successfully destroyed.

- *notCreated*: "String[SetError]" A map of creation id to a SetError object for each record that failed to be created. The possible errors are defined in the description of the method for specific data types.

- *notUpdated*: "String[SetError]" A map of Foo id to a SetError object for each record that failed to be updated. The possible errors are defined in the description of the method for specific data types.

- *notDestroyed*: "String[SetError]" A map of Foo id to a SetError object for each record that failed to be destroyed. The possible errors are defined in the description of the method for specific data types.

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.

Other properties may also be present on the object, as described in the relevant methods.

The following errors may be returned instead of the "foosSet" response:

"accountNotFound": Returned if an _accountId_ was explicitly included with the request, but it does not correspond to a valid account.

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

"accountReadOnly": Returned if the account has isReadOnly == true.

"requestTooLarge": Returned if 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.

"invalidArguments": Returned if one of the arguments is of the wrong type, or otherwise invalid. A "description" property MAY be present on the response object to help debug with an explanation of what the problem was.

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

4. Downloading binary data

Binary data is referenced by a _blobId_ in JMAP. A blob id does not have a name inherent to it, but this is normally given in the same object that contains the blob id.

After completing authentication, the client will receive a _downloadUrl_ as part of the response. This is in [RFC6570] URI Template (level 1) format. The URL MUST contain variables called "accountId" and "blobId". The URL SHOULD contain a variable called "name".

The client may use this template in combination with an accountId 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 MUST make an authenticated GET request to the download URL with the appropriate variables substituted in, and then follow any redirects. In situations where it's not easy to authenticate the download request (e.g.: when downloading a file through a link in a HTML document), the client MAY use a signed GET request (see below for how to issue a signed request).

After following redirects, the server MUST return one of the following responses to a request to the download URL:

4.1. 200: OK

Request successful. The binary data is returned. The "Content-Type" header SHOULD be set to the correct content type for the content.

4.2. 401: Unauthorized

The "Authorization" header was missing or did not contain a valid token and there was no "access_token" query parameter, or it did not contain a valid token. Reauthenticate and then retry the request. As per the HTTP spec, the response MUST have a "WWW-Authenticate" header listing the available authentication schemes.

The server MAY return an HTML page response, which clients MAY show to the user. This is primarily for when the URL is passed off to the browser, and the JMAP client may not see the actual response.

4.3. 404: Not Found

The file was not found at this address.

4.4. 503: Service Unavailable

The server is currently down. The client should try again later with exponential backoff. There is no content in the response.

5. Uploading binary data

There is a single endpoint which handles all file uploads, regardless of what they are to be used for. To upload a file, the client submits a POST request to the file upload endpoint (see the authentication section for information on how to obtain this URL). The Content-Type MUST be correctly set for the type of the file being uploaded. The request MUST be authenticated as per any HTTP request. The request MAY include an "X-JMAP-AccountId" header, with the value being the account to use for the request. Otherwise, the default account will be used.

The server will respond with one of the following HTTP response codes:

5.1. 201: File uploaded successfully

The content of the response is a single JSON object with the following properties:

o *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 content type of the file.

- *size*: "Number" The size of the file in bytes.

- *expires*: "Date" The date the file will be deleted from tempoary storage if not referenced by another object, e.g. used in a draft.

Once the file has been used, for example attached to a draft message, the file will no longer expire, and is instead guaranteed to exist while at least one other object references it. Once no other object references it, the server MAY immediately delete the file at any time. It MUST NOT delete the file during the method call which removed the last reference, so that if there is a create and a delete within the same call that both reference the file, this always works.

If uploading a file would take the user over quota, the server SHOULD delete previously uploaded (but unused) files before their expiry time. This means a client does not have to explicitly delete unused temporary files (indeed, there is no way for it to do so).

If identical binary content is uploaded, the same _blobId_ SHOULD be returned.

The server MUST return one of the following responses to a request to the upload URL:

5.2. 400: Bad request

The request was malformed (this includes the case where an "X-JMAP-AccountId" header is sent with a value that does not exist). The client SHOULD NOT retry the same request. There is no content in the response.

5.3. 401: Unauthorized

The "Authorization" header was missing or did not contain a valid token. Reauthenticate and then retry the request. As per the HTTP spec, the response MUST have a "WWW-Authenticate" header listing the available authentication schemes. There is no content in the response.

5.4. 404: Not Found

The upload endpoint has moved. See the Authentication section of the spec for how to rediscover the current URL to use. There is no content in the response.

5.5. 413: Request Entity Too Large

The file is larger than the maximum size the server is willing to accept for a single file. The client SHOULD NOT retry uploading the same file. There is no content in the response. The client may discover the maximum size the server is prepared to accept by inspecting the server _capabilities_ object, returned with the successful authentication response.

5.6. 415: Unsupported Media Type

The server MAY choose to not allow certain content types to be uploaded, such as executable files. This error response is returned if an unacceptable type is uploaded. The client SHOULD NOT retry uploading the same file. There is no content in the response.

5.6.1. 429: Rate limited

Returned if the client has made too many upload requests recently, or has too many concurrent uploads currently in progress. Clients SHOULD wait a while then try again. The response MAY include a "Retry-After" header indicating how long to wait before making a new request.

5.7. 503: Service Unavailable

The server is currently down. The client should try again later with exponential backoff. There is no content in the response.

6. Push

Any modern email client should be able to update instantly whenever the data on the server is changed by another client or message delivery. Push notifications in JMAP 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 does not send any sensitive data over the push channel, making it suitable for use with less trusted 3rd party intermediaries. 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.

When something changes on the server, the server pushes a small JSON object to the client with the following property:

- *changed*: "String[ChangedStates]" A map of _account id_ to an object encoding the state of data types which 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 *ChangedStates* object is a map of the type name (e.g. "Mailbox" or "Message") to the current state token for that type (i.e. the "state" property that would currently be returned by a call to "getMailboxes" or "getMessages", as appropriate). The types in JMAP are "Mailbox", "Thread", "Message", "ContactGroup", "Contact", "Calendar", "CalendarEvent".

Upon receiving this data, the client can compare the new state strings with its current values to see whether it has the current data for these types. The actual changes can then be efficiently fetched in a single standard API request (using the _getFooUpdates_ type methods).

6.1. Event Source

There are two mechanisms by which the client can receive the push events. The first is directly via a "text/event-stream" resource, as described in <http://www.w3.org/TR/eventsource/>. This is essentially a long running HTTP request down which the server can push data. When a change occurs, the server MUST push an event called *state* to any connected clients.

The server MAY also set a new "Last-Event-Id" that encodes the entire server state visible to the user. When a new connection is made to the event-source endpoint, the server can then work out whether the client has missed some changes which it should send immediately.

The server MUST also send an event called *ping* with an empty object as the data if a maximum of 5 minutes has elapsed since the previous event. This MUST NOT set a new "Last-Event-Id". A client may detect the absence of these to determine that the HTTP connection has been dropped somewhere along the route and so it needs to re-establish the connection.

Refer to the Authentication section of this spec for details on how to get the URL for the event-source endpoint. The request must be authenticated using an "Authorization" header like any HTTP request.

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

6.2. Web hook

The second push mechanism is to register a callback URL to which the JMAP server will make an HTTPS POST request whenever the event occurs. The request MUST have a content type of "application/json" and contain the same UTF-8 JSON encoded object as described above as the body.

The JMAP server MUST also set the following headers in the POST request: - "X-JMAP-EventType: state" - "X-JMAP-User: ${username}" where "${username}" is the username of the authenticated user for which the push event occurred.

The JMAP server MUST follow any redirects. If the final response code from the server is "2xx", the callback is considered a success. 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 response code MUST be considered a *permanent failure* and the callback should be deregistered (not tried again even for future events unless explicitly re-registered by the client).

The URL set by the client MUST use the HTTPS protocol and SHOULD encode within it a unique token that can be verified by the server to know that the request comes from the JMAP server the authenticated client connected to.

The callback is tied to the access token used to create it. Should the access token expire or be revoked, the callback MUST be removed by the JMAP server. The client MUST re-register the callback after reauthenticating to resume callbacks.

Each session may only have a single callback URL registered. It can be set or retrieved using the following API calls.

6.2.1. setPushCallback

To set the web hook, make a call to _setPushCallback_. It takes the following argument:

- *callback*: "String|null" The (HTTPS) URL the JMAP server should POST events to. This will replace any previously set URL. Set to "null" to just remove any previously set callback URL.

The response to _setPushCallback_ is called _pushCallbackSet_. It has the following argument:

o *callback*: "String|null" Echoed back from the call.

The following error may be returned instead of the _mailboxesSet_ response:

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

6.2.2. getPushCallback

To check the currently set callback URL (if any), make a call to _getPushCallback_. It does not take any arguments. The response to _getPushCallback_ is called "pushCallback". It has a single argument:

o *callback*: "String|null" The URL the JMAP server is currently posting push events to, or "null" if none.

7. References

7.1. Normative References

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

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

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <http://www.rfc-editor.org/info/rfc3986>.

[RFC5785] Nottingham, M. and E. Hammer-Lahav, "Defining Well-Known Uniform Resource Identifiers (URIs)", RFC 5785, DOI 10.17487/RFC5785, April 2010, <http://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, <http://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, <http://www.rfc-editor.org/info/rfc6570>.

[RFC6750] Jones, M. and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage", RFC 6750, DOI 10.17487/RFC6750, October 2012, <http://www.rfc-editor.org/info/rfc6750>.

[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, <http://www.rfc-editor.org/info/rfc6764>.

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

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

7.2. URIs

[1] https://tools.ietf.org/html/rfc6238

[2] https://developers.yubico.com/OTP/

[3] https://fidoalliance.org/specifications/download/

[4] https://jwt.io/

Author's Address

Neil Jenkins
FastMail
Level 1, 91 William St
Melbourne VIC 3000
Australia

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