Stream: Internet Engineering Task Force (IETF)
RFC: 9610
Category: Standards Track
Published: December 2024
ISSN: 2070-1721
Author: N. Jenkins, Ed.
Fastmail

RFC 9610
JSON Meta Application Protocol (JMAP) for Contacts

Abstract

This document specifies a data model for synchronising contact data with a server using the JSON Meta Application Protocol (JMAP).

Status of This Memo

This is an Internet Standards Track document.

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

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at https://www.rfc-editor.org/info/rfc9610.

Copyright Notice

Table of Contents

1. Introduction

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

This specification defines a data model for synchronising contacts between a client and a server using JMAP.

1.1. Notational Conventions

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

Type signatures, examples, and property descriptions in this document follow the conventions established in Section 1.1 of [RFC8620]. The Id, UnsignedInt, and UTCDate data types defined in Sections 1.2, 1.3, and 1.4 of [RFC8620] are also used in this document.

1.2. Terminology

The same terminology used in the core JMAP specification (see Section 1.6 of [RFC8620]) is also used in this document.

The terms AddressBook and ContactCard (with these specific capitalizations) are used to refer to the data types defined in this document and instances of those data types.

1.3. Data Model Overview

An Account (see Section 1.6.2 of [RFC8620]) with support for the contact data model contains zero or more AddressBook objects, which is a named collection of zero or more ContactCards. A ContactCard is a representation of a person, company, entity, or a group of such entities in JSContact Card format, as defined in Section 2 of [RFC9553]. Each ContactCard belongs to one or more AddressBooks.

In servers with support for JMAP Sharing [RFC9670], users may see and configure sharing of contact data with others. Sharing permissions are managed per AddressBook.

1.4. Addition to the Capabilities Object

The capabilities object is returned as part of the JMAP Session object; see Section 2 of [RFC8620]. This document defines one additional capability URI.

1.4.1. urn:ietf:params:jmap:contacts

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

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

maxAddressBooksPerCard: `UnsignedInt|null`
: The maximum number of AddressBooks (see Section 2) that can be assigned to a single ContactCard object (see Section 3). This **MUST** be an integer >= 1, or null for no limit (or rather, the limit is always the number of AddressBooks in the account).

mayCreateAddressBook: `Boolean`
: The user may create an AddressBook in this account if, and only if, this is true.

2. AddressBooks

An AddressBook is a named collection of ContactCards. All ContactCards are associated with one or more AddressBooks.

An **AddressBook** object has the following properties:

id: `Id` (immutable; server-set)
: The id of the AddressBook.

name: `String`
: The user-visible name of the AddressBook. This **MUST NOT** be the empty string and **MUST NOT** be greater than 255 octets in size when encoded as UTF-8.

description: `String|null` (default: null)
An optional long-form description of the AddressBook that provides context in shared environments where users need more than just the name.

sortOrder: `UnsignedInt` (default: 0)
Defines the sort order of AddressBooks when presented in the client's UI so it is consistent between devices. The number **MUST** be an integer in the range 0 <= sortOrder < 2^{31}.

An AddressBook with a lower order is to be displayed before a AddressBook with a higher order in any list of AddressBooks in the client's UI. AddressBooks with equal order should be sorted in alphabetical order by name. The sorting should take into account locale-specific character order convention.

isDefault: `Boolean` (server-set)
This **SHOULD** be true for exactly one AddressBook in any account and **MUST NOT** be true for more than one AddressBook within an account. The default AddressBook should be used by clients whenever they need to choose an AddressBook for the user within this account and they do not have any other information on which to make a choice. For example, if the user creates a new contact card, the client may automatically set the card as belonging to the default AddressBook from the user's primary account.

isSubscribed: `Boolean`
True if the user has indicated they wish to see this AddressBook in their client. This **SHOULD** default to false for AddressBooks in shared accounts that the user has access to and true for any new AddressBooks created by the user themself.

If false, the AddressBook and its contents **SHOULD** only be displayed when the user explicitly requests it. The UI may offer to the user the option of subscribing to it.

shareWith: `Id[AddressBookRights]|null` (default: null)
A map of the Principal id (Section 2 of [RFC9670]) to rights for Principals this AddressBook is shared with. The Principal to which this AddressBook belongs **MUST NOT** be in this set. This is null if the AddressBook is not shared with anyone or if the server does not support [RFC9670]. The value may be modified only if the user has the "mayShare" right. The account id for the Principals may be found in the `urn:ietf:params:jmap:principals:owner` capability of the Account to which the AddressBook belongs.

myRights: `AddressBookRights` (server-set)
The set of access rights the user has in relation to this AddressBook.

An **AddressBookRights** object has the following properties:

mayRead: `Boolean`
The user may fetch the ContactCards in this AddressBook.

mayWrite: `Boolean`
The user may create, modify, or destroy all ContactCards in this AddressBook, or move them to or from this AddressBook.

mayShare: `Boolean`
The user may modify the "shareWith" property for this AddressBook.

mayDelete: `Boolean`
The user may delete the AddressBook itself.

2.1. AddressBook/get

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

2.2. AddressBook/changes

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

2.3. AddressBook/set

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

onDestroyRemoveContents: `Boolean` (default: false)
If false, any attempt to destroy an AddressBook that still has a ContactCard in it will be rejected with an "addressBookHasContents" SetError. If true, any ContactCard that is in the AddressBook will be removed from it, and if such a ContactCard does not belong to any other AddressBook, it will be destroyed.

onSuccessSetIsDefault: `Id|null`
If an id is given, and all creates, updates, and destroys (if any) succeed without error, the server will try to set this AddressBook as the default. (For references to AddressBook creations, this is equivalent to a creation-reference, so the id will be the creation id prefixed with a "#".)

If the id is not found or if the change is not permitted by the server for policy reasons, it **MUST** be ignored and the current default AddressBook (if any) will remain as such. No error is returned to the client in this case.

As per Section 5.3 of [RFC8620], if the default AddressBook is successfully changed, any changed objects **MUST** be reported in either the "created" or "updated" argument in the response as appropriate, with the server-set value included.

The "shareWith" property may only be set by users that have the "mayShare" right. When modifying the "shareWith" property, the user cannot give a right to a Principal if the Principal did not already have that right and the user making the change also does not have that right. Any attempt to do so **MUST** be rejected with a "forbidden" SetError.

Users can subscribe or unsubscribe to an AddressBook by setting the "isSubscribed" property. The server **MAY** forbid users from subscribing to certain AddressBooks even though they have permission to see them, rejecting the update with a "forbidden" SetError.

The following extra SetError type is defined for "destroy":

addressBookHasContents: The AddressBook has at least one ContactCard assigned to it and the "onDestroyRemoveContents" argument was false.

3. ContactCards

A **ContactCard** object contains information about a person, company, or other entity, or represents a group of such entities. It is a JSContact Card object as defined in Section 2 of [RFC9553] with the following additional properties:

id: `Id` (immutable; server-set)
The id of the ContactCard. The "id" property **MAY** be different to the ContactCard's "uid" property (as defined in Section 2.1.9 of [RFC9553]). However, there **MUST NOT** be more than one ContactCard with the same uid in an Account.

addressBookIds: `Id[Boolean]`
The set of AddressBook ids that this ContactCard belongs to. A card **MUST** belong to at least one AddressBook at all times (until it is destroyed). The set is represented as an object, with each key being an AddressBook id. The value for each key in the object **MUST** be true.

For any Media object in the card (see Section 2.6.4 of [RFC9553]), a new property is defined:

blobId: `Id`
An id for the Blob representing the binary contents of the resource.

When returning ContactCards, any Media with a URI that uses the "data:" URL scheme [RFC2397] **SHOULD** return a "blobId" property and omit the "uri" property, as this lets clients load the (potentially large) image file only when needed and avoids the overhead of Base64 encoding. The "mediaType" property **MUST** also be set. Similarly, when creating or updating a ContactCard, clients **MAY** send a "blobId" instead of the "uri" property for a Media object.

A contact card with a "kind" property equal to "group" represents a group of contacts. Clients often present these separately from other contact cards. The "members" property, as defined in Section 2.1.6 of [RFC9553], contains a set of uids (as defined in Section 2.1.9 of [RFC9553]) for other contacts that are the members of this group. Clients should consider the group to contain any ContactCard with a matching uid from any account they have access to that has support for the `urn:ietf:params:jmap:contacts` capability. Any uid that cannot be found **SHOULD** be ignored but preserved. For example, suppose a user adds contacts from a shared address book to their private group, then temporarily loses access to this address book. The uids cannot be resolved, so the contacts will disappear from the group. However, if they are given permission to access the data again, the uids will be found and the contacts will reappear.

3.1. ContactCard/get

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

3.2. ContactCard/changes

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

3.3. ContactCard/query

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

3.3.1. Filtering

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

inAddressBook: `Id`
: An AddressBook id. A card must be in this address book to match the condition.

uid: `String`
: A card must have this string exactly as its uid (as defined in Section 2.1.9 of [RFC9553]) to match.

hasMember: `String`
: A card must have a "members" property (as defined in Section 2.1.6 of [RFC9553]) that contains this string as one of the uids in the set to match.

kind: `String`
: A card must have a "kind" property (as defined in Section 2.1.4 of [RFC9553]) that equals this string exactly to match.

createdBefore: `UTCDate`
: The "created" date-time of the ContactCard (as defined in Section 2.1.3 of [RFC9553]) must be before this date-time to match the condition.

createdAfter: `UTCDate`
: The "created" date-time of the ContactCard (as defined in Section 2.1.3 of [RFC9553]) must be the same or after this date-time to match the condition.

updatedBefore: `UTCDate`
: The "updated" date-time of the ContactCard (as defined in Section 2.1.10 of [RFC9553]) must be before this date-time to match the condition.

updatedAfter: `UTCDate`
: The "updated" date-time of the ContactCard (as defined in Section 2.1.10 of [RFC9553]) must be the same or after this date-time to match the condition.

text: `String`
: A card matches this condition if the text matches with text in the card.

name: `String`
: A card matches this condition if the value of any NameComponent in the "name" property or the "full" property in the "name" property of the card (as defined in Section 2.2.1.2 of [RFC9553]) matches the value.

name/given: `String`
: A card matches this condition if the value of a NameComponent with kind "given" inside the "name" property of the card (as defined in Section 2.2.1.2 of [RFC9553]) matches the value.

name/surname: `String`
: A card matches this condition if the value of a NameComponent with kind "surname" inside the "name" property of the card (as defined in Section 2.2.1.2 of [RFC9553]) matches the value.

name/surname2: `String`
: A card matches this condition if the value of a NameComponent with kind "surname2" inside the "name" property of the card (as defined in Section 2.2.1.2 of [RFC9553]) matches the value.

nickname: `String`
: A card matches this condition if the "name" of any Nickname in the "nicknames" property of the card (as defined in Section 2.2.2 of [RFC9553]) matches the value.

organization: `String`
: A card matches this condition if the "name" of any Organization in the "organizations" property of the card (as defined in Section 2.2.3 of [RFC9553]) matches the value.

email: `String`
: A card matches this condition if the "address" or "label" of any EmailAddress in the "emails" property of the card (as defined in Section 2.3.1 of [RFC9553]) matches the value.

phone: `String`
: A card matches this condition if the "number" or "label" of any Phone in the "phones" property of the card (as defined in Section 2.3.3 of [RFC9553]) matches the value.

onlineService: `String`
: A card matches this condition if the "service", "uri", "user", or "label" of any OnlineService in the "onlineServices" property of the card (as defined in Section 2.3.2 of [RFC9553]) matches the value.

address: `String`
: A card matches this condition if the value of any AddressComponent in the "addresses" property or the "full" property in the "addresses" property of the card (as defined in Section 2.5.1 of [RFC9553]) matches the value.

note: `String`
: A card matches this condition if the "note" of any Note in the "notes" property of the card (as defined in Section 2.8.3 of [RFC9553]) matches the value.

If zero properties are specified on the FilterCondition, the condition **MUST** always evaluate to true. If multiple properties are specified, ALL must apply for the condition to be true (it is equivalent to splitting the object into one-property conditions and making them all the child of an AND filter operator).

The exact semantics for matching `String` fields is deliberately not defined to allow for flexibility in indexing implementation, subject to the following:

- Text **SHOULD** be matched in a case-insensitive manner.
- Text contained in either (but matched) single or double quotes **SHOULD** be treated as a phrase search. That is, a match is required for that exact sequence of words, excluding the surrounding quotation marks. Use `\"`, `\'`, and `\\` to match a literal `"`, `'`, and `\` respectively in a phrase.
- Outside of a phrase, whitespace **SHOULD** be treated as dividing separate tokens that may be searched for separately in the contact, but **MUST** all be present for the contact to match the filter.
- Tokens **MAY** be matched on a whole-word basis using stemming (e.g., a text search for `bus` would match "buses", but not "business").

3.3.2. Sorting

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

- "created" - The "created" date on the ContactCard.
- "updated" - The "updated" date on the ContactCard.

The following values for the "property" field on the Comparator object **SHOULD** be supported for sorting:

- "name/given" - The value of the first NameComponent in the "name" property whose "kind" is "given".
- "name/surname" - The value of the first NameComponent in the "name" property whose "kind" is "surname".
- "name/surname2" - The value of the first NameComponent in the "name" property whose "kind" is "surname2".

3.4. ContactCard/queryChanges

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

3.5. ContactCard/set

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

To set a new photo, the file must first be uploaded using the upload mechanism as described in Section 6.1 of [RFC8620]. This will give the client a valid blobId, size, and type to use. The server **MUST** reject attempts to set a file that is not a recognised image type as the photo for a card.

3.6. ContactCard/copy

This is a standard "/copy" method as described in Section 5.4 of [RFC8620].

4. Examples

For brevity, only the "methodCalls" property of the Request object and the "methodResponses" property of the Response object is shown in the following examples.

4.1. Fetching Initial Data

A user has authenticated and the client has fetched the JMAP Session object. It finds a single Account with the "urn:ietf:params:jmap:contacts" capability with id "a0x9" and wants to fetch all the address books and contacts. It might make the following request:

```
[
  ["AddressBook/get", {
    "accountId": "a0x9"
  }, "0"],
  ["ContactCard/get", {
    "accountId": "a0x9"
  }, "1"]
]
```

Figure 1: "methodCalls" Property of a JMAP Request

The server might respond with something like:

```
[
  ["AddressBook/get", {
    "accountId": "a0x9",
    "list": [{
      "id": "062adcfa-105d-455c-bc60-6db68b69c3f3",
      "name": "Personal",
      "description": null,
      "sortOrder": 0,
      "isDefault": true,
      "isSubscribed": true,
      "shareWith": {
        "3f1502e0-63fe-4335-9ff3-e739c188f5dd": {
          "mayRead": true,
          "mayWrite": false,
          "mayShare": false,
          "mayDelete": false
        }
      },
      "myRights": {
        "mayRead": true,
        "mayWrite": true,
        "mayShare": true,
        "mayDelete": false
      }
    }, {
      "id": "cd40089d-35f9-4fd7-980b-ba3a9f1d74fe",
      "name": "Autosaved",
      "description": null,
      "sortOrder": 1,
      "isDefault": false,
      "isSubscribed": true,
      "shareWith": null,
      "myRights": {
        "mayRead": true,
        "mayWrite": true,
        "mayShare": true,
        "mayDelete": false
      }
    }],
    "notFound": [],
    "state": "~4144"
  }, "0"],
  ["ContactCard/get", {
    "accountId": "a0x9",
    "list": [{
      "id": "3",
      "addressBookIds": {
        "062adcfa-105d-455c-bc60-6db68b69c3f3": true
      },
      "name": {
        "components": [
          { "kind": "given", "value": "Joe" },
          { "kind": "surname", "value": "Bloggs" }
        ],
        "isOrdered": true
      },
      "emails": {
```

```
        "0": {
          "contexts": {
            "private": true
          },
          "address": "joe.bloggs@example.com"
        }
      }
    }],
    "notFound": [],
    "state": "ewarbckaqJ::112"
  }, "1"]
]
```

Figure 2: "methodResponses" Property of a JMAP Response

4.2. Changing the Default Address Book

The client tries to change the default address book from "Personal" to "Autosaved" (and makes no other change):

```
[
  ["AddressBook/set", {
    "accountId": "a0x9",
    "onSuccessSetIsDefault": "cd40089d-35f9-4fd7-980b-ba3a9f1d74fe"
  }, "0"]
]
```

Figure 3: "methodCalls" Property of a JMAP Request

The server allows the change, returning the following response:

```
[
  ["AddressBook/set", {
    "accountId": "a0x9",
    "updated": {
      "cd40089d-35f9-4fd7-980b-ba3a9f1d74fe": {
        "isDefault": true
      },
      "062adcfa-105d-455c-bc60-6db68b69c3f3": {
        "isDefault": false
      },
      "oldState": "~4144",
      "newState": "~4148"
    }
  }, "0"]
]
```

Figure 4: "methodResponses" Property of a JMAP Response

5. Internationalisation Considerations

Experience has shown that unrestricted use of Unicode can lead to problems such as inconsistent rendering, users reading text and interpreting it differently than intended, and unexpected results when copying text from one location to another. Servers **MAY** choose to mitigate this by restricting the set of characters allowed in otherwise unconstrained `String` fields. The FreeformClass, as documented in Section 4.3 of [RFC8264], might be a good starting point for this.

Attempts to set a value containing code points outside of the permissible set can be handled in a few ways by the server. The server could choose to strip the forbidden characters or replace them with U+FFFD (the Unicode replacement character) and store the resulting string. This is likely to be appropriate for non-printable characters -- such as the "Control Codes" defined in Section 23.1 of [UNICODE], excluding newline (U+000A), carriage return (U+000D), and tab (U+0009) -- that can end up in data accidentally due to copy-and-paste issues but are invisible to the end user. JMAP allows the server to transform data on create/update as long as any changed properties are returned to the client in the "/set" response so it knows what has changed, as per Section 5.3 of [RFC8620]. Alternatively, the server **MAY** just reject the create/update with an "invalidProperties" SetError.

6. Security Considerations

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

Contacts consist almost entirely of private, personally identifiable information, and represent the social connections of users. Privacy leaks can have real world consequences, and contact servers and clients **MUST** be mindful of the need to keep all data secure.

Servers **MUST** enforce the Access Control Lists (ACLs) set on address books to ensure only authorised data is shared.

7. IANA Considerations

7.1. JMAP Capability Registration for "contacts"

IANA has registered "contacts" in the "JMAP Capabilities" registry as follows:

Capability Name: `urn:ietf:params:jmap:contacts`
Intended Use: common
Change Controller: IETF
Security and Privacy Considerations: this document, Section 6
Reference: this document

7.2. JMAP Data Type Registration for "AddressBook"

IANA has registered "AddressBook" in the "JMAP Data Types" registry as follows:

Type Name: `AddressBook`
Can Reference Blobs: No
Can Use for State Change: Yes
Capability: `urn:ietf:params:jmap:contacts`
Reference: this document

7.3. JMAP Data Type Registration for "ContactCard"

IANA has registered "ContactCard" in the "JMAP Data Types" registry as follows:

Type Name: `ContactCard`
Can Reference Blobs: Yes
Can Use for State Change: Yes
Capability: `urn:ietf:params:jmap:contacts`
Reference: this document

7.4. JMAP Error Codes Registry

The following subsection has registered a new error code in the "JMAP Error Codes" registry, as defined in Section 9 of [RFC8620].

7.4.1. addressBookHasContents

JMAP Error Code: addressBookHasContents
Intended Use: common
Change Controller: IETF
Description: The AddressBook has at least one ContactCard assigned to it, and the "onDestroyRemoveContents" argument was false.
Reference: This document, Section 2.3

7.5. JSContact Property Registrations

IANA has registered the following additional properties in the "JSContact Properties" registry, as defined in Section 3 of [RFC9553].

7.5.1. id

Property Name: id
Property Type: not applicable
Property Context: Card
Intended Usage: reserved

Since Version: 1.0
Change Controller: IETF
Reference: this document

7.5.2. addressBookIds

Property Name: addressBookIds
Property Type: not applicable
Property Context: Card
Intended Usage: reserved
Since Version: 1.0
Change Controller: IETF
Reference: this document

7.5.3. blobId

Property Name: blobId
Property Type: not applicable
Property Context: Media
Intended Usage: reserved
Since Version: 1.0
Change Controller: IETF
Reference: this document

8. References

8.1. 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, <https://www.rfc-editor.org/info/rfc2119>.

[RFC2397] Masinter, L., "The "data" URL scheme", RFC 2397, DOI 10.17487/RFC2397, August 1998, <https://www.rfc-editor.org/info/rfc2397>.

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

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

[RFC9553] Stepanek, R. and M. Loffredo, "JSContact: A JSON Representation of Contact Data", RFC 9553, DOI 10.17487/RFC9553, May 2024, <https://www.rfc-editor.org/info/rfc9553>.

[RFC9670] Jenkins, N., Ed., "JSON Meta Application Protocol (JMAP) Sharing", RFC 9670, DOI 10.17487/RFC9670, November 2024, <https://www.rfc-editor.org/info/rfc9670>.

8.2. Informative References

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

[UNICODE] The Unicode Consortium, "The Unicode Standard", <https://www.unicode.org/versions/latest/>.

Author's Address

Neil Jenkins (EDITOR)
Fastmail
PO Box 234, Collins St West
Melbourne VIC 8007
Australia
Email: neilj@fastmailteam.com
URI: https://www.fastmail.com