RFC 8621 : The JSON Meta Application Protocol (JMAP) for Mail

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 14 août 2019

Date de publication du RFC : Août 2019

https://www.bortzmeyer.org/8621.html

Ce nouveau RFC décrit un remplaçant pour le traditionnel protocole IMAP, remplaçant fondé sur le cadre JMAP ("*JSON Meta Application Protocol*", RFC 8620 [1]).

Le protocole décrit dans ce RFC fournit les mêmes services qu'IMAP (RFC 9051) : accéder aux boîtes aux lettres de courrier, chercher dans ces boîtes, gérer les messages (détruire les inutiles, par exemple), etc. Par rapport à IMAP, outre l'utilisation du format JSON, l'accent est mis sur la synchronisation rapide, l'optimisation pour les clients mobiles, et sur la possibilité de notifications. JMAP est sans état (pas besoin de connexion permanente). Ce « JMAP pour le courrier » s'appuie sur JMAP, normalisé dans le RFC 8620. JMAP est un protocole générique, qui peut servir à synchroniser bien des choses entre un client et un serveur (par exemple un agenda, ou bien une liste de contacts). Par abus de langage, je vais souvent dire « JMAP » dans cet article alors que je devrais normalement préciser « JMAP pour le courrier », premier « utilisateur » du JMAP générique.

JMAP manipule différents types d'objets. Le plus important est sans doute `Email` (section 4 du RFC), qui modélise un message. Il s'agit d'une représentation de haut niveau, le client JMAP n'a pas à connaitre tous les détails de l'IMF ("*Internet Message Format*", RFC 5322), de MIME (RFC 2045), etc. Un objet de type `Email` a une liste d'en-têtes et un corps, et JMAP fournit des méthodes pour accéder aux différentes parties du corps. Il y a même plusieurs représentations d'un message, pour s'adapter aux différents clients. Par exemple, un message MIME est normalement un arbre, de profondeur quelconque, mais un client JMAP peut décider de demander une représentation aplatie, avec juste une liste d'attachements. (La plupart des MUA présentent à l'utilisateur une vue aplatie de l'objet MIME.) Voilà pourquoi l'objet `Email` a plusieurs propriétés, le client choisissant à laquelle il accède :

— `bodyStructure` : l'arbre MIME, c'est la représentation la plus « authentique »,

1. Pour voir le RFC de numéro NNN, `https://www.ietf.org/rfc/rfcNNN.txt`, par exemple `https://www.ietf.org/rfc/rfc8620.txt`

— `textBody` : une liste des parties MIME à afficher quand on préfère du texte,
— `htmlBody` : une liste des parties MIME à afficher quand on préfère de l'HTML,
— `attachments` : la liste des « pièces jointes » (rappelez-vous que le concept de « pièces jointes » a été créé pour l'interface avec l'utilisateur ; il n'a pas de sens en MIME, qui ne connait qu'un arbre avec des feuilles de différents types).

Les en-têtes doivent pouvoir être internationaux (RFC 6532).

Un message a évidemment des métadonnées, parmi lesquelles :
— `id`, un identifiant du message (ce n'est pas le `Message-ID:`, c'est attribué par le serveur JMAP), contrairement à IMAP, l'identificateur d'un message ne change pas, même quand le message change de boîte, et il peut apparaitre dans plusieurs boîtes à la fois
— `blobIf`, un identifiant du message représenté sous la forme d'une suite d'octets, à analyser par le client, par opposition à l'objet de haut niveau identifié par `id`,
— `size`, la taille du message,
— `keywords`, des mots-clés, parmi lesquels certains, commençant par un dollar, ont une signification spéciale.

En IMAP, les mots-clés spéciaux sont précédés d'une barre inverse. En JMAP, c'est le dollar. Parmi ces mots-clés, `$seen` indique que le message a été lu, `$answered`, qu'on y a répondu, `$junk`, que le serveur l'a classé comme spam, etc. Ces mots-clés sont dans un registre IANA `<https://www.iana.org/assignments/imap-jmap-keywords/imap-jmap-keywords.xml#imap-keywords-1>`.

Et quelles opérations sont possibles avec les objets de type `Email` ? Ce sont les opérations génériques de JMAP (RFC 8620, section 5). Ainsi, on peut récupérer un message avec `Email/get`. Cette requête :

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

peut récupérer, par exemple, cette valeur :

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

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

Notez que le client a demandé deux messages, mais qu'un seul, le `f123u457`, a été trouvé.

Tout aussi indispensable, `Email/query` permet de demander au serveur une recherche, selon de nombreux critères comme la date, les mots-clés, ou bien le contenu du corps du message.

`Email/set` permet de modifier un message, ou d'en créer un (qu'on pourra ensuite envoyer avec `EmailSubmission`, décrit plus loin). Notez qu'il n'y a pas de `Email/delete`. Pour détruire un message, on utilise `Email/set` en changeant la propriété indiquant la boîte aux lettres, pour mettre la boîte aux lettres spéciale qui sert de poubelle (rôle = `trash`).

Comme IMAP, JMAP pour le courrier a la notion de **boîte aux lettres** (section 2 du RFC). Une boîte (vous pouvez appeler ça un dossier ou un "*label*" si vous voulez) est un ensemble de messages. Tout message est dans au moins une boîte. Les attributs importants d'une boîte :

— Un nom unique (par exemple Vacances ou Personnel), en Unicode (RFC 5198),
— Un identificateur attribué par le serveur (et a priori moins lisible par des humaines que ne l'est le nom),
— Un rôle, optionnel, qui indique à quoi sert la boîte, ce qui est utile notamment si le serveur peut être utilisé en JMAP et en IMAP. Ainsi, le rôle `inbox` identifie la boîte où le courrier arrive par défaut. (Les rôles figurent dans un registre IANA `<https://www.iana.org/assignments/imap-mailbox-name-attributes/imap-mailbox-name-attributes.xml#imap-mailbox-name-attr` créé par le RFC 8457.)
— Certains attributs ne sont pas fixes, par exemple le nombre total de messages contenus dans la boîte, ou bien le nombre de messages non lus.
— Les droits d'accès (ACL, cf. RFC 4314.) Les permissions sont par boîte, pas par message.

Ensuite, on utilise les méthodes JMAP pour accéder aux boîtes (révisez donc le RFC 8620, qui décrit le JMAP générique). Ainsi, pour accéder à une boîte,, on utilise la méthode JMAP `Mailbox/get`, qui utilise le `/get` JMAP (RFC 8620, section 5.1). Le paramètre `ids` peut être nul, cela indique alors qu'on veut récupérer tous les messages (c'est ce qu'on fait dans l'exemple ci-dessous).

De même, pour effectuer une recherche sur le serveur, JMAP normalise la méthode `/query` (RFC 8620, section 5.5) et JMAP pour le courrier peut utiliser `Mailbox/query`.

Par exemple, si on veut voir toutes les boîtes existantes, le client JMAP envoie le JSON :

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

et reçoit une réponse du genre (on n'affiche que les deux premières boîtes) :

```
[[ "Mailbox/get", {
     "accountId": "u33084183","state": "78540",
     "state": "78540",
     "list": [{
         "id": "MB23cfa8094c0f41e6",
         "name": "Boîte par défaut",
         "role": "inbox",
         "totalEmails": 1607,
         "unreadEmails": 15,
         "myRights": {
               "mayAddItems": true,
               ...},
       {
         "id": "MB674cc24095db49ce",
         "name": "Personnel",
 ...
```

Notez que `state` est l'identificateur d'un état de la boîte. Si on veut ensuite récupérer les changements, on pourra utiliser `Mailbox/changes` avec comme paramètre `"sinceState": "88540"`.

Dans JMAP, les messages peuvent être regroupés en fils de discussion (”*threads*”, section 3 du RFC). Tout message est membre d'un fil (parfois membre unique). Le RFC n'impose pas de méthode unique pour constituer les fils mais suggère :

— D'utiliser les en-têtes du RFC 5322 (`In-Reply-To:` ou `References:` indiquant le `Message-Id:` d'un autre message).
— Et de vérifier que les messages ont le même sujet (après avoir supprimé des préfixes comme « Re : »), pour tenir compte des gens qui volent les fils <https://www.bortzmeyer.org/ne-pas-voler-les-fils.html>. Cette heuristique est imparfaite (le sujet peut avoir changé sans pour autant que le message soit sans rapport avec le reste du fil).

On peut ensuite accéder aux fils. Le client envoie :

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

Et récupère les fils `f123u4` et `f41u44` :

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

Un client qui vient de se connecter à un serveur JMAP va typiquement faire un `Email/query` sans conditions particulières, pour recevoir la liste des messages (ou alors en se limitant aux N messages les plus récents), puis récupérer les fils de discussion correspondants avec `Thread/get`, récupérer les messages eux-mêmes. Pour diminuer la latence, JMAP permet au client d'envoyer toutes ces requêtes en une seule fois ("*batching*"), en disant pour chaque requête qu'elle doit utiliser le résultat de la précédente ("*backreference*", membre JSON `resultOf`).

JMAP permet également d'envoyer des messages. Un client JMAP n'a donc besoin que d'un seul protocole, contrairement au cas courant aujourd'hui où il faut IMAP **et** SMTP, configurés séparement, avec, trop souvent, l'un qui marche et l'autre pas. Cela simplifie nettement les choses pour l'utilisateur. Cela se fait avec le type `EmailSubmission` (section 7 du RFC). Deux importantes propriétés d'un objet de type `EmailSubmission` sont `mailFrom`, l'expéditeur, et `rcptTo`, les destinataires. Rappel important sur le courrier électronique : il y a les adresses indiquées dans le message (champs `To:`, `Cc:`, etc, cf. RFC 5322), et les adresses indiquées dans l'enveloppe (commandes SMTP comme `MAIL FROM` et `RCPT TO`, cf. RFC 5321). Ces adresses ne sont pas forcément identiques. Lorsqu'on apprend le fonctionnement du courrier électronique, la distinction entre ces deux catégories d'adresses est vraiment cruciale.

Un `EmailSubmission/set` va créer l'objet `EmailSubmission`, **et** envoyer le message. Ici, on envoie à `john@example.com` et `jane@example.com` un message (qui avait été créé par `Email/set` et qui avait l'identificateur `M7f6ed5bcfd7e2604d1753f6c`) :

```
[[ "EmailSubmission/set", {
        "accountId": "ue411d190",
        "create": {
          "k1490": {
            "identityId": "I64588216",
            "emailId": "M7f6ed5bcfd7e2604d1753f6c",
            "envelope": {
              "mailFrom": {
                "email": "john@example.com",
                "parameters": null
              },
              "rcptTo": [{
                "email": "jane@example.com",
                "parameters": null
              },
              ...
              ]
            }
          }
        },
        "onSuccessUpdateEmail": {
          "#k1490": {
            "mailboxIds/7cb4e8ee-df87-4757-b9c4-2ea1ca41b38e": null,
            "mailboxIds/73dbcb4b-bffc-48bd-8c2a-a2e91ca672f6": true,
            "keywords/$draft": null
          }
        }
      }, "0" ]]
```

Anecdote sur l'envoi de courrier : les premières versions de « JMAP pour le courrier » utilisaient une boîte aux lettres spéciale, nommée `Outbox`, où on mettait les messages à envoyer (comme dans ActivityPub).

JMAP a d'autres types d'objets amusants, comme `VacationResponse` (section 8), qui permet de faire envoyer un message automatiquement lorsqu'on est absent (l'auto-répondeur du serveur doit

évidemment suivre le RFC 3834, pour éviter de faire des bêtises comme de répondre à une liste de diffusion). On crée un objet avec `VacationResponse/set` et hop, l'auto-répondeur est amorcé.

Et je n'ai pas parlé de tout, par exemple JMAP permet de pousser des changements depuis le serveur vers le client, si la boîte aux lettres est modifiée par un autre processus (RFC 8620, section 7).

JMAP a le concept de **capacités** ("*capabilities*"), que le serveur annonce au client, dans un objet JSON (rappel : JSON nomme « objets » les dictionnaires), et sous la forme d'un URI. JMAP pour le courrier ajoute trois capacités au registre des capacités JMAP <https://www.iana.org/assignments/jmap/jmap.xml#jmap-capabilities>, `urn:ietf:params:jmap:mail` pour dire qu'on sait gérer le courrier, `urn:ietf:params:jmap:submission`, pour dire qu'on sait en envoyer (cf. RFC 6409, sur ce concept de soumission d'un message), et `urn:ietf:params:jmap:vacationresponse` pour dire qu'on sait gérer un auto-répondeur.

Le courrier électronique pose plein de problèmes de sécurité intéressants. La section 9 de notre RFC les détaille. Par exemple, les messages en HTML sont particulièrement dangereux. (Il est toujours amusant de voir des entreprises de sécurité informatique envoyer leur "*newsletter*" en HTML, malgré les risques associés, qui sont aujourd'hui bien connus.) Le RFC rappelle donc aux clients JMAP (mais c'est valable pour tous les MUA) que du JavaScript dans le message peut changer son contenu, qu'un message en HTML peut récupérer du contenu sur l'Internet (via par exemple un `<img src=...`), ce qui trahit le lecteur et fait fuiter des données privées, que CSS, quoique moins dangereux que JavaScript, permet également des trucs assez limites, que les liens en HTML ne pointent pas toujours vers ce qui semble (`<a href="http://evil.example/">cliquez ici pour aller sur le site de votre banque https://good-bank.example</a>`), etc. Pour faire face à tous ces dangers du courrier en HTML, le RFC suggère de nettoyer le HTML avant de l'envoyer au client. Attention, outre que c'est une modification du contenu, ce qui est toujours délicat politiquement, le faire proprement est difficile, et le RFC recommande fortement d'utiliser une bibliothèque bien testée, de ne pas le faire soi-même à la main (il y a trop de pièges). Par exemple, en Python, on peut utiliser lxml <http://lxml.de/>, et son module `Cleaner`, ici en mode extrémiste qui retire tout ce qui peut être dangereux :

```
from lxml.html.clean import Cleaner
...
cleaner = Cleaner(scripts=True, javascript=True, embedded=True, meta=True, page_structure=True,
                                    links=True, remove_unknown_tags=True,
                                    style=True)
```

Mais il est probablement impossible de complètement sécuriser HTML dans le courrier. Le RFC explique à juste titre que HTML augmente beaucoup la surface d'attaque. Une organisation soucieuse de sécurité ne devrait pas traiter le HTML dans le courrier.

La soumission du courrier (cf. RFC 6409) pose également des problèmes de sécurité. Imaginez un client JMAP piraté et qui serve ensuite à envoyer du spam de manière massive, utilisant le compte de l'utilisateur ignorant de ce piratage. Les MTA qui acceptent du courrier ont des mécanismes de défense (maximum N messages par heure, avec au plus M destinataires par message…) mais ces mécanismes marchent d'autant mieux que le serveur a davantage d'information. Si la soumission via JMAP est mise en œuvre par un simple relais vers un serveur SMTP de soumission, certaines informations sur le client peuvent être perdues. De tels relais doivent donc veiller à transmettre au serveur SMTP toute l'information disponible, par exemple via le mécanisme XCLIENT <http://www.postfix.org/XCLIENT_README.html>.

JMAP a été développé essentiellement au sein de FastMail, qui le met en œuvre sur ses serveurs. Il existe une page « officielle » présentant le protocole <https://jmap.io/>, qui explique entre autres

les avantages de JMAP par rapport à IMAP. Vous y trouverez également des conseils pour les auteurs de clients, très bien faits et qui donnent une bonne idée de comment le protocole marche. Ce site Web est un passage recommandé.

On y trouve également une liste de mises en œuvre de JMAP <https://jmap.io/software.html>. Ainsi, le serveur IMAP bien connu Cyrus a déjà JMAP en expérimental <https://cyrusimap.org/imap/download/release-notes/3.0/x/3.0.3.html>. Le MUA K-9 Mail a, quant à lui, commencé le travail <https://github.com/k9mail/k-9/issues/3272#issuecomment-528326161>.