RFC 9205 : Building Protocols with HTTP

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 10 juin 2022

Date de publication du RFC : Juin 2022

https://www.bortzmeyer.org/9205.html

Aujourd'hui, la grande majorité des API accessibles via le réseau fonctionnent au-dessus de HTTP. Ce nouveau RFC, qui remplace le RFC 3205 [1], décrit les bonnes pratiques pour la conception de telles API, notamment pour les protocoles IETF bâtis sur HTTP, comme DoH ou RDAP.

Il y a plein d'applications qui fonctionnent au-dessus de HTTP. Ce nouveau RFC se concentre sur celles qui sont d'usage général et qui ont plusieurs mises en œuvre et déploiements. (Si vous faites un service centralisé qui n'a qu'un seul déploiement de son API spécifique, ce RFC ne va pas forcément être pertinent pour vous.) Si vous avez déjà lu le RFC 3205, il faudra tout recommencer, les changements sont nombreux. Ces applications utilisant HTTP sont parfois qualifiées de REST mais, en toute rigueur, toutes ne suivent pas rigoureusement les principes de REST. Notez aussi un sous-ensemble, CRUD, pour les applications dont l'essentiel du travail est de créer/supprimer/gérer des objets distants.

Normalement, HTTP avait été conçu pour le Web et ses usages. Mais on voit aujourd'hui de très nombreuses API réseau être fondées sur HTTP pour diverses raisons :

— Parce que le développeur ou la développeuse ne connait que ça,
— parce qu'il existe un très grand nombre de bibliothèques pour faire les clients (comme l'excellente Requests <https://requests.readthedocs.io/> pour le langage Python, qui sera utilisée ici dans plusieurs exemples), et de cadriciels pour les serveurs,
— parce que dans certains cas, il sera même possible d'utiliser un navigateur Web pour y accéder, et que tout le monde a un navigateur Web et est familier avec ce logiciel,
— parce que certains services comme l'authentification sont déjà disponibles dans les logiciels existants,
— parce que HTTP est le seul protocole dont on peut être sûr qu'il passera même dans les réseaux les plus coincés comme les "*hotspots*" des hôtels et des aéroports.

1. Pour voir le RFC de numéro NNN, https://www.ietf.org/rfc/rfcNNN.txt, par exemple https://www.ietf.org/rfc/rfc3205.txt

Mais tout n'est pas forcément rose et HTTP peut ne pas être bien adapté à ce qu'envisage le développeur de l'API. Et cette développeuse peut faire des erreurs lors de la conception de l'API, erreurs que ce RFC vise à éviter.

En effet, quand on développe une API sur HTTP, il y a plusieurs décisions à prendre :

— Définir un nouveau plan d'URL ? (C'est quand même rare, presque tout le monde utilise `http:` ou, aujourd'hui, `https:`.)
— Utiliser un port « non-standard » ?
— Comment coexister avec les autres utilisations de HTTP, comme la navigation sur le Web ?

La section 2 précise l'applicabilité de ce RFC. Il concerne les protocoles qui utilisent HTTP (ports 80 ou 443, plans d'URI `http:` ou `https:`). Ceux qui utiliseraient une version modifiée de HTTP ne comptent pas, et cette pratique est d'ailleurs déconseillée, puisque ces versions modifiées feraient probablement perdre les avantages d'utiliser HTTP, notamment la réutilisation des logiciels et infrastructures existants.

Section 3, maintenant. Quelles sont les caractéristiques importantes de HTTP, qui gouvernent ce que peuvent faire les applications qui l'utilisent ? D'abord, sa sémantique très générale : on peut tout faire avec HTTP. Notammment, HTTP est indépendant du type de ressources sur lesquelles il agit. Ainsi, des composants HTTP génériques (bibliothèques, serveurs, relais) peuvent être développés et déployés pour des applications très différentes, même des applcations qui n'existent pas encore. (Voilà d'alleurs pourquoi la section précédente insistait sur le fait q'il ne faut pas modifier HTTP.) Plus subtile serait l'erreur qui consisterait à spécifier un certain profil de HTTP, en restreignant ce que HTTP peut faire ou pas (« la réponse à un `POST` doit être 201 »). Une telle restriction, là encore, empêcherait d'utiliser cerrtains composants génériques, en faisant perdre à HTTP de sa généralité.

Une autre erreur courante est de s'attribuer tout ou partie de l'espace de nommage fourni par les liens hypertextes. C'est par exemple le cas lorsqu'une application estime qu'elle peut contrôler tout le chemin dans l'URI et décider que `/truc/machin` est forcément à elle (RFC 8820). Cela complique le déploiement, par exemple si on veut installer cette application sous `/chose` et excusivement sous ce chemin (cf. section 4.4). L'application devrait au contraire permettre de la souplesse et utiliser les possibilités qu'offre le système de liens (RFC 8288).

Enfin, HTTP dispose de nombreuses possibilités comme le multiplexage que permettent HTTP/2 (RFC 9113) et HTTP/3 (RFC 9114), l'intégration avec TLS, la possibilité de relayage, la négociation de contenu, la disponibilité de nombreux clients, et l'application qui utilise HTTP doit donc veiller à ne pas casser cet écosystème, et en tout cas à ne pas réinventer la roue, alors que HTTP offre déjà de nombreuses solutions éprouvées.

Bref, compte-tenu de tout cela, comment faire pour bien utiliser HTTP dans sa nouvelle application ? La section 4 est là pour répondre à cette question.

D'abord, bien définir la dépendance de l'application à HTTP, en donnant comme référence RFC 9110 (et surtout pas une version spécifique de HTTP, toujours afin de profiter au maximum de l'écosystème existant). On notera quand même que DoH (RFC 8484) impose (section 5.2 de son RFC) au moins HTTP/2, pour être sûr d'avoir du multiplexage. Notre RFC permet explicitement ce genre d'exceptions.

Le RFC recommande également, quand on montre un dialogue HTTP titre d'exemple, d'utiliser plutôt les conventions de HTTP/1 (RFC 9112), plus lisibles. Donc, par exemple, `GET /truc HTTP/1.1` plutôt que le `:method = GET :path = /resource` de HTTP/2. C'est ce que fait curl :

```
% curl -v http://www.example
> GET / HTTP/2
> Host: www.example
> user-Agent: curl/7.68.0
> accept: */*
>
< HTTP/2 200
< content-Type: text/plain
< content-Length: 13
```

On l'a dit, il ne faut pas modifier le comportement de base de HTTP. Ce qu'on peut faire, par contre :

— Spécifier le type des données (RFC 6838), par exemple JSON (qu'utilise RDAP),
— spécifier des champs dans l'en-tête,
— spécifier comment on trouve les ressources via des liens (RFC 8288).

Et le client ? L'application qui utilise HTTP ne devrait pas exiger de comportement trop spécifique du client ; idéalement, un navigateur Web normal devrait pouvoir interagir avec l'application. On peut par exemple s'appuyer sur les principes FETCH <`https://fetch.spec.whatwg.org/`>. Il est également préférable que l'application qui va utiliser HTTP soit claire sur le traitement attendu pour les redirections HTTP, ou pour les "*cookies*" (RFC 6265), et rappeler que la vérification des certificats doit se faire selon les principes de la section 4.3.4 du RFC 9110.

Le client doit, idéalement, pouvoir se configurer avec uniquement un URL. (Par exemple, un serveur DoH est annoncé ainsi, comme `https://doh.bortzmeyer.fr/`, la seule information dont vous avez besoin pour l'utiliser.) Et si on ne connait qu'un nom de domaine ? La solution du chemin d'URL fixe qu'on s'alloue (≪ obligatoirement `/app` ≫) étant interdite (RFC 8820), il y a le choix :

— Utiliser un chemin sous `/.well-known` (RFC 8615),
— Utiliser les gabarits du RFC 6570, pour générer les URI (un tel mécanisme de découverte est plus souple mais sans doute moins rapide).

Et le plan d'URI (le premier composant de l'URI) ? `http:` et surtout `https:` sont évidemment recommandés mais on peut aussi choisir un plan spécifique. Cela va évidemment rendre l'application inutilisable par un navigateur Web ordinaire. Certains navigateurs permettent d'enregistrer un mécanisme de gestion de ces plans non standards (comme le `registerProtocolHandler()` du WHATWG <`https://html.spec.whatwg.org/`>) mais cela ne marche pas partout. Et on aura le même problème avec tout l'écosystème logiciel de HTTP. Bref, utiliser un plan autre que `http:` ou `https:` fera perdre une bonne partie des avantages qu'il y avait à utiliser le protocole HTTP. D'autres problèmes se poseront comme l'impossibilité d'utiliser le concept d'origine (RFC 6454) par exemple dans la "*Same Origin Policy*", ou comme d'autres fonctions utiles de HTTP ("*cookies*", authentification, mémorisation - RFC 9111, HSTS - RFC 6797, etc). Si vous tenez encore, après tout ça, à créer un plan à vous, consultez le RFC 7595.

Et les ports ? HTTP utilise par défaut les ports 80 pour le trafic en clair et 443 pour le traffic chiffré. Utiliser un autre port est possible (`https://machin.example:666`) mais rend le trafic de l'application distinguable, ce qui peut être gênant pour la vie privée. (C'est un des choix de conception de DoH que d'utiliser HTTPS sur le port 443, pour ne pas être distinguable, et donc être plus difficile à filtrer.) Le RFC 7605 donne des détails sur ce choix des ports.

Maintenant, quelles méthodes HTTP utiliser ? Le RFC exige que les applications utilisant HTTP se servent uniquement des méthodes enregistrées <`https://www.iana.org/assignments/http-methods/http-methods.xml#methods`>, comme `GET` ou `PUT`. Certes, une procédure existe pour enregistrer de nouvelles méthodes mais l'IETF insiste que ces nouvelles méthodes doivent être génériques, et non pas limitées aux besoins d'une seule application. (Le RFC 4791 avait créé des méthodes spécifiques, mais c'était avant. C'est maintenant interdit.)

Donc, pas de méthodes nouvelles. Mais quelle(s) méthode(s) utiliser ? `GET` est le choix le plus évident. Cette méthode est idempotente (et permet donc, entre autres, la mémorisation), et a une sémantique simple et compréhensible. Elle a quelques limites (comme le fait que tous les éventuels paramètres doivent être dans l'URL, ce qui peut nécessiter un encodage spécial, et peut empêcher des paramètres de grande taille) mais rien de bien grave. Si c'est trop gênant pour une application donnée, il ne reste plus qu'à utiliser `POST`.

Et pour récupérer des métadonnées sur le service ? Le RFC note que la méthode `OPTIONS` n'est pas très pratique, par exemple parce qu'elle ne permet pas de donner comme documentation un simple URL (la méthode par défaut étant `GET`). Il recommande plutôt un URL dans `/.well-known` (RFC 8615), en créant un nouveau nom, ou bien avec les URL `host-meta` (RFC 6415). Pour des métadonnées sur une ressource particulière, il est recommandé d'utiliser les liens (RFC 8288). Le RFC note que, dans ce dernier cas, l'en-tête `Link:` marche même avec la méthode `HEAD` donc pas besoin de récupérer la ressource pour avoir des informations sur ses métadonnées.

Et les codes de retour HTTP comme 403 ou 404, comment les utiliser ? D'abord, une application qui utilise HTTP n'a pas forcément un complet contrôle sur ces codes de retour, qui peuvent être générés par des composants logiciels différents. Donc, le client HTTP doit se méfier, le code reçu n'est pas forcément significatif de l'application. Ensuite, une application peut avoir davantage de messages différents qu'il n'existe de codes de retour HTTP, ce qui peut pousser à de mauvaises pratiques, comme l'utilisation de codes de retour non standard, ou commme l'utilisation de codes certes standard mais utilisés d'une manière très éloignée de ce qui était prévu. Bref, le RFC conseille de découpler les erreurs applicatives des erreurs HTTP, de ne pas chercher à tout prix un code de retour HTTP pour chaque erreur applicative, et de ne pas hésiter à utiliser les codes de retour génériques (comme 500, pour « quelque chose ne va pas dans le serveur »). Pour envoyer des informations plus détaillées sur l'erreur, il est préférable d'utiliser les techniques du RFC 7807. Autre avertissement du RFC, les raisons envoyées par le serveur après un code de retour (”*404 File not found*”) ne sont pas significatives. Dans certains cas (message HTTP dans un message HTTP), elles ne sont pas transmises du tout, contrairement au code de retour, la seule information sur laquelle on peut compter. L'application ne doit donc pas espérer que le client recevra ces raisons.

Autre difficulté pour le concepteur ou la conceptrice d'applications utilisant HTTP, les redirections. Il y a quatre redirections différentes en HTTP, chacune pouvant être temporaire ou définitive, et permettant de changer de méthode ou pas (une requête `POST` indiquant une redirection suivie d'une requête `GET` par le client). On a donc :

— 301, définitive et permettant de changer de méthode,
— 302, temporaire et permettant de changer de méthode,
— 307, temporaire et ne permettant pas de changer de méthode,
— 308, définitive et ne permettant pas de changer de méthode.

Et il faut donc réfléchir un peu avant de choisir un code de redirection (le RFC privilégie 301 et 302, plus souples).

Et les champs dans l'en-tête ? Une application a souvent envie d'en ajouter, que ce soit dans la requête ou dans la réponse. Mais le RFC n'est pas très chaud, demandant qu'on mette les informations plutôt dans l'URL ou dans le corps du message HTTP. Si on crée de nouveaux en-têtes, en théorie (c'est très théorique…), il faut les enregistrer à l'IANA (RFC 9110, section 16.3). Si ces en-têtes ont une structure, il est très recommandé qu'elles suivent les règles du RFC 8941.

Et le corps du message, justement ? L'application doit spécifier quel format est attendu. C'est souvent JSON (RFC 8259) mais cela peut être aussi XML, CBOR (RFC 8949), etc.

Une des grandes forces d'HTTP est la possibilité de mémorisation, décrite en détail dans le RFC 9111. La mémorisation améliore les performances, rend le service moins sensible aux perturbations, et

permet le passage à l'échelle. Les applications qui utilisent HTTP ont donc tout intérêt à permettre et à utiliser cette mémorisation. Il est donc recommandé d'indiquer dans la réponse une durée de vie, de préférence avec `Cache-Control: max-age=...` ou bien, si c'est nécessaire, d'indiquer explicitement que la réponse ne doit pas être mémorisée (`Cache-Control: no-store`).

Un autre avantage pour une application d'utiliser HTTP est l'existence d'un cadre général pour l'authentification (RFC 9110, section 11). Attention, certains mécanismes ne doivent être utilisés qu'au-dessus d'HTTPS, comme la "*basic authentication*" du RFC 7617. HTTPS permet également d'utiliser les certificats client pour l'authentification. (Attention, avec TLS [Caractère Unicode non montré [2]] 1.2, ces certificats, qui contiennent des données personnelles, sont transmis en clair.)

Conséquence de l'utilisation de HTTP, l'application est utilisable via un navigateur Web. Cela peut être vu comme un avantage (tout le monde a un navigateur Web sous la main) ou comme un inconvénient (si la sémantique de l'application ne permet pas réellement un usage pratique depuis un navigateur). Mais quoi qu'on en pense, l'application sera accessible aux navigateurs, et il est donc important de s'assurer que cela ne provoquera pas de problème. Par exemple, si on peut changer un état avec une requête `POST`, l'application pourrait être attaquée assez facilement par CSRF. Si l'application tire une partie des données qu'elle renvoie en réponse d'une source que l'attaquant peut contrôler, on risque le XSS. Il est donc recommandé, même si l'application n'est pas prévue pour être utilisée par un navigateur, de suivre les mêmes règles de développement sécurisé que si elle devait être accédée depuis un navigateur, notamment :

— utiliser des champs dans l'en-tête comme `X-Content-Type-Options: nosniff`,
— utiliser CSP,
— utiliser `Referrer-Policy:`,
— utiliser l'option `HttpOnly` sur les "*cookies*" (RFC 6265, section 5.2.6).

Voici un exemple d'une réponse suivant ces principes :

```
HTTP/1.1 200 OK
Content-Type: application/example+json
X-Content-Type-Options: nosniff
Content-Security-Policy: default-src 'none'
Cache-Control: max-age=3600
Referrer-Policy: no-referrer
```

Il reste à régler la question des frontières de l'application. Le plus simple pour l'application est d'avoir son propre nom de domaine et donc une origine (RFC 6454) unique. Cela simplifie par exemple des problèmes comme les "*cookies*". Mais cela complique le déploiement, empêchant de mettre plusieurs applications derrière le même nom. Le RFC conseille donc plutôt de concevoir des applications pouvant coexister avec d'autres applications sous le même nom (RFC 8820).

Un mot sur la sécurité pour finir (section 6 du RFC). D'abord, une application qui utilise HTTP va évidemment hériter des questions de sécurité générale de HTTP, comme détaillées dans la section 17 du RFC 9110. Vu le caractère sensible des données traitées par beaucoup d'applications, le RFC recommande l'utilisation de HTTPS. Mais il développe aussi la question de la vie privée. HTTP est très bavard et le serveur en apprend beaucoup, souvent beaucoup trop, sur son client. Ainsi, les "*cookies*", l'adresse IP source, les `ETags`, les tickets de session TLS sont très utiles au serveur qui voudrait suivre un client. Et le RFC rappelle que HTTP donne assez d'informations « auxiliaires » pour pouvoir reconnaitre un client (ce qu'on nomme le "*fingerprinting*"). Bref, le maintien de son intimité va être aussi difficile que sur le Web.

2. Car trop difficile à faire afficher par LaTeX

Ce RFC remplace l'ancien RFC 3205. Comme le note l'annexe A de notre RFC, il y a trop de changements pour les lister ; ce document est très différent de son prédécesseur (qui date de 2002 !).

Voyons maintenant quelques exemples d'application utilisant HTTP. Dans le monde IETF, il y a évidemment RDAP (RFC 7480, RFC 9082 et RFC 9083). RDAP suit bien les principes de ce RFC. Par exemple, les chemins d'URL comme `/domain` ou `/ip` ne sont pas forcément à la « racine » du serveur HTTP. Autre exemple, DoH (RFC 8484), également fidéle (heureusement !) aux recommandations de l'IETF. Notez que ces recommandations laissent des choix. Ainsi, lorsque le nom de domaine cherché n'est pas trouvé, RDAP renvoie le code 404 (RFC 7480, section 5.3) alors que DoH préfère renvoyer un 200 (le serveur HTTP a bien été joignable et a bien répondu), gardant le signal de non-existence uniquement dans la réponse DNS (RFC 8484, section 4.2.1) transportée sur HTTP (l'argument est que DoH ne fait que transporter les requêtes d'un autre protocole, contrairement à RDAP).

J'ai parlé plus haut de la possibilité d'utilisation d'un navigateur Web ordinaire pour accéder aux applications utilisant HTTP. Mais comme ces applications envoient souvent des données structurées en JSON, il faut un navigateur qui gère bien le JSON. Et c'est justement ce que fait Firefox, qui sait l'afficher de manière pratique :

Terminons avec quelques exemples d'API « finales » (donc pas le sujet principal du RFC, qui parle de protocoles IETF). Commençons modestement par l'API du "*DNS looking glass*" <`https://www.bortzmeyer.org/dns-lg-usage.html`>. A priori, elle suit tous les principes de ce RFC. En tout cas, elle essaie. Mais si vous constatez des différences avec le RFC, n'hésitez pas à faire un rapport <`https://framagit.org/bortzmeyer/dns-lg/-/issues`>. Autre API intéressante, celle des sondes RIPE Atlas <`https://atlas.ripe.net/`>. Elle utilise toutes les possibilités de HTTP, notamment les multiples méthodes (`DELETE` pour supprimer une mesure en cours, par exemple). J'aurais juste trouvé plus logique d'utiliser `PUT` au lieu de `POST` pour créer une mesure. L'API de Mastodon (cf. sa documentation <`https://docs.joinmastodon.org/client/intro/`>) est encore plus incohérente, utilisant `POST` pour créer un pouète, mais `PUT` pour le mettre à jour.