RFC 7231 : Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 14 juin 2014

Date de publication du RFC : Juin 2014

https://www.bortzmeyer.org/7231.html

Ce RFC est le deuxième plus important de la longue série des nouveaux RFC décrivant le protocole HTTP 1.1 <https://www.bortzmeyer.org/http-11-reecrit.html>. Le premier, le RFC 7230[1] décrivait les principes généraux, les URI et la syntaxe des messages. Ce second RFC fournit la sémantique desdits messages. Il est donc assez long, mais facile à comprendre car il consiste surtout en une liste détaillée de champs d'en-têtes, de codes de retour, etc. Il a depuis été remplacé par le RFC 9110.

Si on veut comprendre HTTP 1.1 en détail, il faut donc commencer par le RFC 7230. Ensuite, on peut lire ce RFC 7231 mais la plupart des gens l'utiliseront sans doute uniquement comme référence, pour vérifier un point particulier de la norme. Rappelons juste qu'un message HTTP est soit une requête, soit une réponse, et que requête ou réponse sont composées d'une première ligne, puis d'une série de champs (formant l'en-tête de la requête ou de la réponse) et éventuellement d'un corps. La première ligne d'une requête est une méthode (comme `GET`), qui donne le sens principal de la requête (l'en-tête pouvant légèrement modifier cette sémantique) et ses paramètres, la première ligne d'une réponse est surtout composée d'un code de retour, les fameux trois chiffres.

Les méthodes des requêtes (comme `GET` ou `POST`) agissent sur des **ressources** (section 2 de notre RFC). Les ressources peuvent être n'importe quoi. Au début du Web, c'étaient forcément des fichiers mais cela a évolué par la suite et c'est désormais un concept bien plus abstrait. Une ressource est identifiée par un URI (RFC 3986 et section 2.7 du RFC 7230). Notez donc qu'on spécifie indépendemment méthode et ressource (contrairement à d'autres systèmes hypertextes où c'était l'identificateur qui indiquait l'action souhaitée).

1. Pour voir le RFC de numéro NNN, `https://www.ietf.org/rfc/rfcNNN.txt`, par exemple `https://www.ietf.org/rfc/rfc7230.txt`

La ressource, vous l'avez vu, est une notion assez abstraite. On ne peut interagir avec elle que via l'étroite interface de HTTP, sans savoir comment le serveur à l'autre bout gère les ressources (fichier ? extraction dynamique d'une base de données ? autre processus ?) Cette abstraction est à la base du principe « REST <http://roy.gbiv.com/pubs/dissertation/top.htm> ». Mais la ressource a une **représentation** (section 3 de notre RFC), qui est une suite d'octets, quelque chose de concret, donc. Une même ressource peut avoir plusieurs représentations. Un exemple simple est celui où la ressource est une image et où il y a une représentation en JPEG, une en PNG, etc. Les différentes représentations seront des suites d'octets complètement différentes les unes des autres alors qu'elles représenteront « la même » image.

Le choix de la représentation est fait par le mécanisme dit de « négociation du contenu <https://www.bortzmeyer.org/negociation-contenu-http.html> ».

Les représentations sont étiquetées avec un type de média (dit aussi type MIME) à la syntaxe bien connue « type/sous-type » comme `image/png`. En plus du type et du sous-type, ils peuvent contenir des paramètres comme le "*charset*" (terme impropre car c'est en fait un encodage), "*charsets*" qui sont enregistrés à l'IANA <https://www.iana.org/assignments/character-sets/character-sets.xml>, suivant le RFC 2978. Le tout est mis dans le champ `Content-type:` comme, par exemple :

```
Content-Type: text/html; charset=UTF-8
```

Malheureusement, les serveurs HTTP ne sont pas toujours correctement configurés et les étiquettes de type/sous-type peuvent être incorrectes. Certains navigateurs Web tentent de résoudre le problème en analysant la représentation (ce qu'on nomme le « "*content sniffing*" ») mais cette pratique, peu fiable, est déconseillée par notre RFC, notamment pour des raisons de sécurité (il existe des logiciels malveillants encodés de façon à sembler une image GIF pour certains logiciels et un exécutable Windows pour d'autres).

Outre ce type/sous-type, la représentation a d'autres métadonnées. Par exemple, on peut indiquer une langue, soit dans la requête (la langue qu'on veut), soit dans la réponse (la langue obtenue). La langue est codée par une étiquette de langue (RFC 5646) comme `fr`, `az-Arab` ou `en-AU`. En pratique, demander des langues spécifiques n'a guère d'intérêt car la qualité de la traduction n'est pas prise en compte <https://www.bortzmeyer.org/web-et-version-originale.html>. Si je préfère le français, mais que je peux lire l'anglais, une demande dans cet ordre me donnera surtout des pages Web mal traduites en français.

Les méthodes de HTTP font l'objet de la section 4 de notre RFC. Certaines méthodes sont sûres, c'est-à-dire qu'elles sont en lecture seule : elles ne modifient pas les ressources sur le serveur. On peut donc les utiliser sans modération. Les méthodes peuvent être idempotentes, c'est-à-dire que leur application répétée produit un résultat identique à une application unique. Toute méthode sûre est idempotente (puisqu'elle ne change pas la ressource) mais l'inverse n'est pas vrai. Enfin, certaines méthodes sont qualifiées de « cachables » (désolé pour l'affreux terme, et qui est faux en plus car il ne s'agit pas de dissimuler quoi que ce soit, c'est une allusion aux caches dans les réseaux). Les réponses peuvent potentiellement être gardées en mémoire pour resservir. Toutes les méthodes sûres sont cachables.

La reine des méthodes, la première définie, la plus courante est évidemment `GET`. C'est la méthode par défaut de la plupart des clients (par exemple, avec curl, c'est celle qui sera utilisée si on ne met pas l'option `-X/--request`). Elle demande au serveur d'envoyer une représentation de la ressource indiquée. Dans le cas du serveur HTTP le plus simple, les URI sont traduits en noms de fichiers locaux (et la syntaxe des URI reflète la syntaxe des noms de fichiers Unix) et ces fichiers sont alors simplement

envoyés au client. Mais on peut mettre en œuvre GET de bien d'autres façons. GET est sûre et donc idempotente et cachable.

Utilisée surtout pour le débogage, la méthode HEAD ne transfère pas la représentation, mais uniquement le code de retour et les en-têtes de la réponse. Cela permet de tester un serveur sans épuiser la capacité réseau <https://www.bortzmeyer.org/capacite.html>, par exemple dans un programme de vérification de liens. HEAD est sûre et donc idempotente et cachable. (Attention, certaines applications Web boguées renvoient un code de succès alors même qu'elles ont un problème ; pour vérifier le bon fonctionnement d'une telle application, il faut faire un GET et analyser le contenu, comme avec les options -r ou -s du check_http des plugins Nagios <https://www.monitoring-plugins.org/>.)

Au contraire, POST n'est pas sûre : elle demande qu'on traite le contenu de la requête (avec GET, la requête n'a pas de contenu, juste l'en-tête) dans le cadre d'une ressource donnée. Son utilisation la plus connue est le cas où la ressource visée est un formulaire et où la requête contient les valeurs qui vont être placées dans les champs. Dans certains cas, POST est cachable (mais, en pratique, peu de logiciels de cache en profitent).

Plus radical, PUT remplace la ressource par le contenu de la requête (ou bien crée une ressource si elle n'existait pas déjà). Elle n'est évidemment pas sûre mais elle est idempotente (le résultat, qu'on applique la requête une fois ou N fois, sera toujours une ressource dont la représentation est le contenu de la requête). Le code de retour (voir la section 6 de notre RFC) sera différent selon que la ressource a été créée ou simplement remplacée. Dans le premier cas, le client récupérera un 201, dans le second un 200. PUT et POST sont souvent confondus et on voit souvent des API REST qui utilisent POST (plus courant et plus connu des développeurs) pour ce qui devrait être fait avec PUT. La différence est pourtant claire : avec un PUT, la ressource sur le serveur est remplacée (PUT est donc idempotente), alors qu'avec POST elle est modifiée pour intégrer les données envoyées dans le corps du POST.

Voici un exemple de PUT avec l'option -T de curl (qui indique le fichier à charger) :

```
% curl -v -T test.txt http://www.example.net/data/test.txt
> PUT /data/test.txt HTTP/1.1
> User-Agent: curl/7.37.0
> Host: www.example.net
> Accept: */*
> Content-Length: 7731
...
< HTTP/1.1 201 Created
< Server: nginx/1.6.0
< Date: Fri, 30 May 2014 20:38:36 GMT
< Content-Length: 0
< Location: http://www.example.net/data/test.txt
```

(Le serveur nginx était configuré avec dav_methods PUT;.)

La méthode DELETE permet de supprimer une ressource stockée sur le serveur, comme le ferait le rm sur Unix.

La méthode CONNECT est un peu particulière car elle n'agit pas réellement sur une ressource distante : elle dit au serveur de créer un tunnel vers une destination indiquée en paramètre et de relayer ensuite les données vers cette destination. Elle sert lorsqu'on parle à un relais Web et qu'on veut chiffrer le trafic de bout en bout avec TLS. Par exemple :

```
CONNECT server.example.com:443 HTTP/1.1
Host: server.example.com:443
```

va se connecter au port 443 de `server.example.com`.

Restent les méthodes `OPTIONS` et `TRACE` qui servent pour l'auto-découverte et le débogage. Rarement mises en œuvre et encore plus rarement activées, vous trouverez peu de serveurs HTTP qui les gèrent.

Une fois les méthodes étudiées, dans la section 4, place aux en-têtes envoyés après la ligne qui contient la méthode. C'est l'objet de la section 5 du RFC. Ces en-têtes permettent au client HTTP d'envoyer plus de détails au serveur, précisant la requête.

D'abord (section 5.1), les en-têtes de **contrôle**. Ce sont ceux qui permettent de diriger le traitement de la requête par le serveur. Le plus connu est `Host:`, défini dans le RFC 7230. Mais il y a aussi `Expect:`, qui permet de réclamer de la part du serveur qu'il mette en œuvre certaines fonctions. Si ce n'est pas le cas, le serveur peut répondre 417 (≪ *"I'm sorry, Dave"* ≫). La seule valeur actuellement définie pour `Expect:` est `100-continue` qui indique que le client va envoyer de grandes quantités de données et veut recevoir une réponse intérimaire (code de réponse 100).

Les autres en-têtes de contrôle sont définis dans d'autres RFC de la famille. Ceux relatifs aux caches, comme `Cache-Control:` ou `Pragma:`, sont dans le RFC 7234. `Range:`, lui, figure dans le RFC 7233.

Après les en-têtes de contrôle, il y a ceux liés aux **requêtes conditionnelles**, comme `If-Match:` ou `If-Modified-Since:`. Ils sont décrits dans le RFC 7232.

Troisième catégorie d'en-têtes transmis lors des requêtes, ceux liés à la négociation de contenu <https://www.bortzmeyer.org/negociation-contenu-http.html> (section 5.3), comme `Accept:`. Ils vont permettre d'indiquer le genre de contenu que le client préfère. Comme ces choix ne sont pas binaires (≪ je gère PNG et JPEG, ex-aequo, et je peux me débrouiller avec GIF s'il n'y a vraiment pas le choix ≫), les en-têtes de cette catégorie prennent un paramètre indiquant la qualité. Le paramètre se nomme `q` et sa valeur est le poids attribué à une certaine préférence, exprimée sous forme d'un nombre réel entre 0 et 1. Ainsi, lorsqu'une requête `GET` vient avec cet en-tête `Accept:` :

```
Accept: audio/*; q=0.2, audio/basic
```

elle indique que le client préfère `audio/basic` (pas de qualité indiquée donc on prend celle par défaut, 1). Pour l'exemple cité plus haut avec les formats d'image, cela pourrait être :

```
Accept: image/bmp; q=0.5, image/jpeg, image/gif; q=0.8, image/png
```

indiquant une préférence pour JPEG et PNG (pas de qualité indiquée, donc 1 pour tous les deux), avec un repli vers GIF et, dans les cas vraiment où il n'y a rien d'autre, BMP (notez que le type `image/bmp` n'est pas enregistré mais on le rencontre quand même souvent).

Même principe pour sélectionner un encodage des caractères avec `Accept-Charset:` (rappelez-vous que le terme *"charset"* utilisé à l'IETF est incorrect, il désigne plus que le jeu de caractères). Et l'encodage des données ? Il peut se sélectionner avec `Accept-Encoding:` :

```
Accept-Encoding: gzip;q=1.0, identity; q=0.5, *;q=0
```

(`identity` signifie « aucune transformation ». La qualité zéro indiquée à la fin signifie qu'en aucun cas on n'acceptera cette solution.)

Il existe enfin un en-tête `Accept-Language:` pour indiquer les langues préférées mais, en pratique, il ne sert pas à grand'chose <https://www.bortzmeyer.org/web-et-version-originale.html>.

Catégorie suivante d'en-têtes, ceux d'authentification. C'est le cas de `Authorization:` défini dans le RFC 7235.

Une dernière catégorie d'en-têtes est représentée par les en-têtes de **contexte** (section 5.5), qui donnent au serveur quelques informations sur son client. Ils sont trois, `From:` qui contient l'adresse de courrier électronique de l'utilisateur. Il n'est guère utilisé que par les robots, pour indiquer une adresse à laquelle se plaindre si le robot se comporte mal. En effet, son envoi systématique poserait des gros problèmes de protection de la vie privée. Le deuxième en-tête de cette catégorie est `Referer:` qui indique l'URI où le client a obtenu les coordonnées de la ressource qu'il demande. (À noter que le nom est une coquille ; en anglais, on écrit "*referrer*".) Si je visite l'article de Wikipédia sur le Chaperon Rouge et que j'y trouve un lien vers `http://www.example.org/tales/redridinghood.html`, lors de la connexion au serveur `www.example.org`, le navigateur enverra :

```
Referer: http://fr.wikipedia.org/wiki/Le_Petit_Chaperon_rouge
```

Cet en-tête pose lui aussi des problèmes de vie privée. Il peut renseigner le serveur sur l'historique de navigation, les requêtes effectuées dans un moteur de recherche, etc. Notamment, le navigateur ne doit pas envoyer cet en-tête si l'URI de départ était local, par exemple de plan `file:`.

Enfin, `User-Agent:`, le troisième en-tête de contexte, permet d'indiquer le logiciel du client et son numéro de version. Comme certains sites Web, stupidement, lisent cet en-tête et adaptent leur contenu au navigateur (une violation hérétique des principes du Web), les navigateurs se sont mis à mentir de plus en plus, comme le raconte une jolie histoire <http://webaim.org/blog/user-agent-string-history/>. Par exemple, le navigateur que j'utilise en ce moment envoie :

```
User-Agent: Mozilla/5.0 (X11; Linux i686; rv:29.0) Gecko/20100101 Firefox/29.0 Iceweasel/29.0.1
```

(Au passage, si vous voulez voir tout ce que votre navigateur envoie, vous pouvez essayer ce service </apps/env>.)

Si vous utilisez Apache, et que vous voulez conserver, dans le journal, la valeur de certains en-têtes rigolos, Apache permet de le faire <http://httpd.apache.org/docs/2.4/mod/mod_log_config.html#formats> pour n'importe quel en-tête. Ainsi :

```
LogFormat "[%h]:%{remote}p %l %u %t \"%r\" %>s %b \"%{Referer}i\" \"%{User-Agent}i\" %v" combinedv6
```

va enregistrer le `Referer:` et le `User-Agent:` ce qui donnera :

```
[2001:db8:22::864:89]:37127 - - [12/Jun/2014:10:09:17 +0200] "GET /greylisting.html HTTP/1.1" 200 3642 "http
```

J'ai déjà parlé du **code de retour** HTTP, les fameux trois chiffres qui indiquent si la requête a réussi ou pas. La section 6 le décrit plus en profondeur. Ce code est composé d'une classe, indiquée par le premier chiffre, et d'un code particulier dans les deux chiffres suivants. Des nouveaux codes sont régulièrement créés et un client HTTP doit donc se préparer à rencontrer de temps en temps des codes inconnus. En revanche, le nombre de classes est fixe. Ce sont :

— 1xx : codes informatifs indiquant que la requête a été reçue mais le travail demandé n'est pas encore terminé (par exemple 100 qui signifie « patientez deux secondes, ça arrive » ou 101 lorsqu'on utilise WebSocket).
— 2xx : la requête est un succès (le code le plus fréquent est 200 « tout va bien, voici ta réponse » mais il y en a plusieurs autres comme 201 indiquant que la ressource n'existait pas mais a été créée avec succès, par exemple par un `PUT`).
— 3xx : codes de redirection, indiquant que le client va devoir aller voir ailleurs pour terminer sa requête (300 pour indiquer qu'il y a plusieurs choix possibles et que le client doit se décider). 301 et 302 permettent désormais de changer la méthode utilisée (`POST` en `GET` par exemple) 307 et 308 ne le permettent pas. 301 et 308 sont des redirections permanentes (le navigateur Web peut changer ses signets), les autres sont temporaires. Si vous utilisez Apache, la directive `Redirect` permet de faire des 301 (`Redirect temp`) ou des 302 (`Redirect permanent`), pour les autres, il faut indiquer explicitement le code (cf. la documentation <`http://httpd.apache.org/docs/2.4/mod/mod_alias.html#redirect`>). Attention à bien détecter les boucles (redirection vers un site qui redirige...)
— 4xx : erreur située du côté du client, qui doit donc changer sa requête avant de réessayer. C'est par exemple le fameux 404, « ressource non trouvée » ou le non moins célèbre 403 « accès interdit ». À noter que, si vous êtes administrateur d'un serveur et que vous savez que la ressource a définitivement disparu, vous pouvez envoyer un 410, qui indique une absence définitive (`Redirect gone /PATH` dans Apache, au lieu d'un simple `Redirect` mais ce n'est pas forcément respecté <`https://www.bortzmeyer.org/dinos-partis.html`>.) Ah, et si vous voyez un 402, sortez vos bitcoins, cela veut dire "*Payment required*".
— 5xx : erreur située du côté du serveur, le client peut donc essayer une requête identique un peu plus tard (c'est par exemple le 500, « erreur générique dans le serveur » lorsque le programme qui produisait les données s'est planté pour une raison ou l'autre).

La liste complète des codes enregistrés (rappelez-vous qu'elle est parfois allongée) est stockée à l'IANA <`https://www.iana.org/assignments/http-status-codes/http-status-codes.xhtml`> mais c'est plus rigolo de regarder la fameuse page des codes HTTP représentés par des chats <`https://www.flickr.com/photos/girliemac/sets/72157628409467125/with/6508023065/`>, où les images ont été très bien choisies (ce sont des images de cette collection qui sont affichées par ce blog en cas d'erreur). Il existe aussi une page équivalente <`http://httpstatusdogs.com/`> avec des chiens.

Derrière la première ligne de la réponse, celle qui contient ce code de retour en trois chiffres, les entêtes de réponse. La section 7 du RFC les décrit en détail. Là encore, plusieurs catégories. La première est celle du **contrôle**. C'est le cas de `Date:` qui indique date et heure du serveur. Le format de cette information est un sous-ensemble de celui du RFC 5322 (et, hélas, pas du RFC 3339, bien plus simple et lisible). À noter qu'on trouve parfois des serveurs utilisant d'autres formats : c'était mal spécifié au début de HTTP. Un exemple avec le format recommandé :

```
% curl -v http://www.hackersrepublic.org/
...
< HTTP/1.0 200 OK
< Server: Apache/2.4.6
< Date: Sat, 14 Jun 2014 12:11:19 GMT
```

`Location:` sert en cas de redirection à indiquer le nouvel URI. Par exemple :

```
% curl -v http://www.bortzmeyer.org/eusthatius-test-grammars.html
...
> GET http://www.bortzmeyer.org/eusthatius-test-grammars.html HTTP/1.1
...
< HTTP/1.0 301 Moved Permanently
< Date: Sat, 14 Jun 2014 12:13:21 GMT
< Location: http://www.bortzmeyer.org/eustathius-test-grammars.html
```

(Redirection mise en place suite à une coquille dans le lien depuis un site important.)

Le champ `Vary:` est plus subtil. Il indique de quels paramètres de la requête dépend le résultat obtenu. C'est indispensable pour les caches : si une réponse varie selon, mettons, la langue demandée, un autre client qui demande une autre langue ne doit pas recevoir le même contenu, même si l'URL est identique. Un cache Web doit donc utiliser comme clé d'une ressource, non pas l'URL seul mais la combinaison de l'URL et du contenu de `Vary:`. Voici un exemple sur ce blog, où le format d'image peut être négocié <`https://www.bortzmeyer.org/negociation-contenu-http.html`> :

```
% curl -v http://www.bortzmeyer.org/images/nat66
...
> GET /images/nat66 HTTP/1.1
> Accept: */*
...
< HTTP/1.1 200 OK
< Content-Location: nat66.gif
< Vary: negotiate,accept
...
```

C'est la version GIF qui a été choisie et le `Vary:` indique bien que cela dépendait de l'en-tête `Accept:`.

Troisième catégorie de réponses, les **validateurs**, comme `Last-Modified:`. Leur utilisation principale est pour des requêtes conditionnelles ultérieures (RFC 7232). Ainsi, une réponse avec un `Last-Modified:`, indiquant la date de dernier changement, permettra au client de demander plus tard « cette ressource, si elle n'a pas changé depuis telle date », limitant ainsi le débit réseau si la ressource est inchangée. Autre en-tête validateur, `Etag:`, dont la valeur est une étiquette ("*entity tag*") identifiant de manière unique une version donnée d'une ressource. Ainsi :

```
% curl -v https://www.laquadrature.net/fr/snowden-terminator-et-nous
...
< HTTP/1.1 200 OK
< ETag: "da6e32e8d35ff7cf11f9c83d814b9328"
...
```

La ressource `snowden-terminator-et-nous` de ce serveur est identifiée par l'étiquette `da6e32e8d35ff7cf11` (probablement un condensat MD5).

Il y a deux autres catégories pour les en-têtes de réponse, la troisième comprend les en-têtes utilisées pour l'authentification (RFC 7235) comme `WWW-Authenticate:`. Et la quatrième est composée des en-têtes indiquant le contexte. La plus connue est `Server:` qui indique le(s) logiciel(s) utilisé(s) par le serveur. Par exemple, dans le cas de ce blog (et changeons un peu, utilisons wget au lieu de curl) :

```
% wget --server-response --output-document /dev/null http://www.bortzmeyer.org/
...
 HTTP/1.1 200 OK
 Server: Apache/2.2.22 (Debian)
 ...
```

La section 9.6 rappelle que, contrairement à une idée reçue, les indications sur la version du logiciel que transporte cet en-tête ne posent guère de problèmes de sécurité. Les attaquants ne s'y fient pas (ils savent que cet en-tête peut être modifié par l'administrateur du serveur et que, de toute façon, la vulnérabilité n'est pas liée à une version, certains systèmes "*patchent*" le logiciel mais sans changer le numéro de version) et essaient donc toutes les attaques possibles (le serveur HTTP qui héberge ce blog reçoit souvent des tentatives d'attaques exploitant des failles d'IIS, alors que c'est un Apache et qu'il l'annonce).

Toutes ces listes de codes, en-têtes, etc, ne sont pas figées. Régulièrement, de nouveaux RFC les mettent à jour et la version faisant autorité est donc stockée dans un registre à l'IANA. La section 8 rappelle la liste de ces registres :

— Un nouveau registre pour les méthodes <`https://www.iana.org/assignments/http-methods/http-methods.xhtml#methods`> (`GET`, `PUT`, etc, le RFC 7237 enregistre formellement les anciennes méthodes). Les éventuelles nouvelles méthodes doivent être génériques, c'est-à-dire s'appliquer à tous les genres de ressources. Lors de l'enregistrement, il faudra bien préciser si la méthode est idempotente, sûre, etc.
— Un autre registre pour les codes de retour <`https://www.iana.org/assignments/http-status-codes/http-status-codes.xhtml#http-status-codes-1`> comme 200 ou 404. L'ajout d'un nouveau code nécessite le processus "*IETF review*" décrit dans le RFC 5226, section 4.1.
— Encore un autre pour les en-têtes <`https://www.iana.org/assignments/message-headers/message-headers.xhtml`>, qu'ils soient dans les requêtes ou dans les réponses. Ce registre est partagé avec d'autres protocoles qui utilisent un format similaire, notamment le courrier électronique. Les procédures sont celles du RFC 3864. Autrefois, il était fréquent de définir des en-têtes sans les enregistrer, en les préfixant d'un `X-`. Cette pratique a été **abandonnée** par le RFC 6648.
— Et enfin un dernier registre pour le codage du contenu <`https://www.iana.org/assignments/http-parameters/http-parameters.xhtml#content-coding`> (en fait pas tout à fait le dernier, j'en ai omis certains).

Reste le gros morceau de la sécurité, en section 9. Notre RFC étudie successivement plusieurs points qui peuvent poser problème. D'abord, l'attaque basée sur le nom de fichier. Si un serveur HTTP imprudent transforme directement le chemin dans l'URL en un nom de fichier du système de fichiers local, il peut sans s'en douter donner accès à des endroits non prévus. Par exemple, sur un serveur Unix, lorsque la requête est :

```
GET /../../../../../../../../etc/passwd HTTP/1.1
```

un serveur mal programmé donnerait accès au fichier (normalement non distribué `/etc/passwd`), car `..`, sur Unix, désigne le répertoire situé un cran au dessus (et le répertoire courant, si c'est la racine, donc l'attaquant a intérêt à mettre beaucoup de `..` pour être sûr d'atteindre la racine avant de redescendre vers `/etc`).

Autre attaque possible, l'injection de commandes ou de code. Le contenu du chemin dans l'URL, ou celui des autres paramètres de la requête, ne mérite aucune confiance : il est complètement sous le contrôle du client, qui peut être un attaquant, et qui peut donc inclure des caractères spéciaux, interprétés par un des logiciels qui manipulent ce contenu. Imaginons par exemple que le contenu de l'en-tête `Referer:` soit mis dans une base de données relationnelle et que le client ait envoyé un entête :

```
Referer: http://www.google.com/' ; DROP TABLE Statistics; SELECT'
```

Comme l'apostrophe et le point-virgule sont des caractères spéciaux pour le langage SQL, on pourrait réussir ici une injection SQL <https://www.bortzmeyer.org/sql-injection.html> : le code SQL situé entre les deux apostrophes (ici, une destruction de table) sera exécuté. Ces attaques par injection sont bien connues, relativement faciles à empêcher (les données issues de l'extérieur ne doivent pas être passées à un autre logiciel avant désinfection), mais encore fréquentes.

L'actualité (les révélations de Snowden) poussent évidemment à se préoccuper des questions de vie privée. Un client HTTP peut envoyer plein d'informations révélatrices (comme la localisation physique de l'utilisateur, son adresse de courrier électronique, des mots de passe...) Le logiciel, qui connait ces informations, doit donc faire attention à ne pas les divulguer inutilement. Certaines personnes utilisent l'URI comme un mot de passe (en y incluant des données secrètes et en comptant que l'URI ne sera pas publié) ce qui est une très mauvaise idée. En effet, les URI sont partagés, par les systèmes de synchronisation de signets, par les navigateurs qui consultent des listes noires d'URI, par des utilisateurs qui n'étaient pas conscients que c'était un secret, par l'en-tête `Referer:`... Bref, il ne faut pas compter sur le secret de l'URI. Créer un site Web confidentiel et compter sur le fait qu'on n'a envoyé l'URI qu'à un petit groupe restreint de personnes est une très mauvaise stratégie de sécurité. Autre piège pour la vie privée, les informations apparemment purement techniques et non personnelles transmises par le navigateur Web, comme le `User-Agent:`, les en-têtes de négociation de contenu (comme `Accept-Language:`), mais aussi la liste des polices ou bien d'autres caractéristiques. Prises ensemble, ces informations permettent le "*fingerprinting*", l'identification d'un navigateur unique au milieu de millions d'autres, grâce à ses caractéristiques uniques. Le "*fingerprinting*" marche bien car, en pratique, la combinaison de toutes ces informations techniques est souvent unique. Vous ne me croyez pas ? Regardez le Panopticlick <https://panopticlick.eff.org/>.

Voilà, le gros du RFC est passé, il était long mais c'est parce que HTTP est riche et plus complexe qu'il n'en a l'air. Les annexes de ce RFC fournissent encore quelques informations intéressantes. Ainsi, l'annexe A explique les différences entre HTTP et MIME. HTTP se sert de beaucoup de pièces empruntées au courrier électronique et à MIME et, à première vue, lire le RFC 2045 suffit pour les utiliser. Mais HTTP ayant des caractéristiques différentes de celles du courrier, ce n'est pas tout à fait vrai. Par exemple, l'entête `MIME-Version:` n'est pas obligatoire et les formats de date sont plus stricts. Plus gênant, HTTP n'a pas les limites de longueur de ligne de MIME et ne replie donc pas les lignes trop longues.

La liste des changements entre le précédent RFC, le RFC 2616 et ce RFC figure dans l'annexe B. Rappelez-vous que le protocole est le même, HTTP 1.1 et qu'il n'y a donc pas normalement, sauf bogue dans la spécification, d'incompatibilité entre les deux RFC. Les changements sont normalement uniquement dans la rédaction de la norme. Parmi les changements qui peuvent quand même avoir des conséquences pratiques :

— Les méthodes possibles sont désormais dans un registre IANA <https://www.iana.org/assignments/http-methods/http-methods.xhtml#methods>.
— ISO 8859-1 n'est plus le jeu de caractères par défaut des en-têtes (on trouvait très peu d'en-têtes qui tiraient profit de cette règle : mon blog en avait un et plusieurs personnes m'avaient fait remarquer, bien à tort, que c'était illégal). Même chose pour les contenus de type texte.
— Les codes de retour de redirection 301 et 302 autorisent explicitement le changement de méthode (de `POST` en `GET`, par exemple), alignant la norme avec la réalité du comportement des logiciels.