eBook Gratuit

APPRENEZ HTTP

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#http

Table des matières

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: http

It is an unofficial and free HTTP ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official HTTP.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Démarrer avec HTTP

Remarques

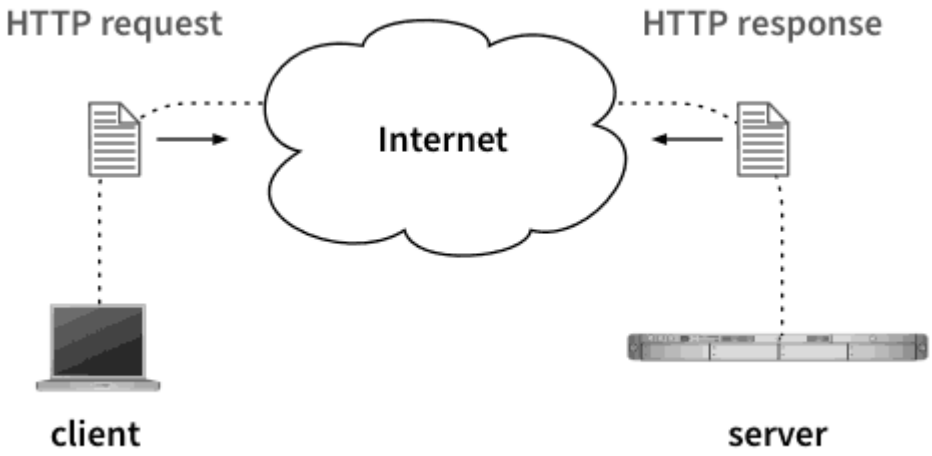
HTTP (Hypertext Transfer Protocol) utilise un modèle de requête client / serveur-réponse. HTTP est un protocole sans état, ce qui signifie qu'il ne nécessite pas que le serveur conserve des informations ou un statut sur chaque utilisateur pour la durée de plusieurs requêtes. Cependant, pour des raisons de performances et pour éviter les problèmes de latence de connexion TCP, des techniques telles que les connexions persistantes, parallèles ou en pipeline peuvent être utilisées.

Versions

Version	Remarques)	Date de sortie estimée
HTTP / 0.9	"Tel qu'implémenté"	1991-01-01
HTTP / 1.0	Première version de HTTP / 1.0, dernière version qui n'a pas été intégrée à une RFC	1992-01-01
HTTP / 1.0 r1	Premier RFC officiel pour HTTP	1996-05-01
HTTP / 1.1	Améliorations dans la gestion des connexions, prise en charge des hôtes virtuels basés sur des noms	1997-01-01
HTTP / 1.1 r1	Utilisation de mots clés ambigus nettoyée, problèmes éventuels liés au cadrage des messages	1999-06-01
HTTP / 1.1 r2	Refonte majeure	2014-06-01
HTTP / 2	Première spécification pour HTTP / 2	2015-05-01

Examples

Requêtes HTTP et réponses



HTTP décrit comment un client HTTP, tel qu'un navigateur Web, envoie une requête HTTP via un réseau à un serveur HTTP, qui envoie ensuite une réponse HTTP au client.

La requête HTTP est généralement une demande de ressource en ligne, telle qu'une page Web ou une image, mais peut également inclure des informations supplémentaires, telles que des données saisies dans un formulaire. La réponse HTTP est généralement une représentation d'une ressource en ligne, telle qu'une page Web ou une image.

HTTP / 1.0

HTTP / 1.0 a été décrit dans la RFC 1945 .

HTTP / 1.0 ne dispose pas de certaines fonctionnalités requises de facto sur le Web, telles que l'en-tête `Host` pour les hôtes virtuels.

Cependant, les clients et serveurs HTTP déclarent parfois toujours utiliser HTTP / 1.0 s'ils ont une implémentation incomplète du protocole HTTP / 1.1 (par exemple sans codage de transfert ou de pipeline), ou la compatibilité est plus importante que les performances (par exemple lors de la connexion au proxy local). les serveurs).

```
GET / HTTP/1.0
User-Agent: example/1

HTTP/1.0 200 OK
Content-Type: text/plain

Hello
```

HTTP / 1.1

HTTP / 1.1 a été initialement spécifié en 1999 dans RFC 2616 (protocole) et RFC 2617 (authentification), mais ces documents sont désormais obsolètes et ne doivent pas être utilisés comme référence:

> N'utilisez pas RFC2616. Supprimez-le de vos disques durs, de vos signets et gravez (ou recyclez de manière responsable) toutes les copies imprimées.

- Mark Nottingham, président du groupe de travail HTTP

La spécification à jour de HTTP / 1.1, qui correspond à la manière dont HTTP est implémenté aujourd'hui, se trouve dans les nouveaux RFC 723x:

- RFC 7230: Syntaxe et routage des messages
- RFC 7231: Sémantique et contenu
- RFC 7232: demandes conditionnelles
- RFC 7233: demandes de plage
- RFC 7234: mise en cache
- RFC 7235: authentification

HTTP / 1.1 ajouté, entre autres fonctionnalités:

- le codage de transfert en blocs, qui permet aux serveurs d'envoyer des réponses de taille inconnue de manière fiable,
- les connexions TCP / IP persistantes (qui étaient des extensions non standard dans HTTP / 1.0),
- les demandes de plage utilisées pour reprendre les téléchargements,
- contrôle du cache.

HTTP / 1.1 a tenté d'introduire le traitement par pipeline, ce qui a permis aux clients HTTP de réduire le temps de latence entre la demande et la réponse en envoyant plusieurs requêtes à la fois sans attendre de réponses. Malheureusement, cette fonctionnalité n'a jamais été correctement implémentée dans certains proxies, ce qui a provoqué la perte ou le réaménagement des connexions.

```
GET / HTTP/1.0
User-Agent: example/1
Host: example.com

HTTP/1.0 200 OK
Content-Type: text/plain
Content-Length: 6
Connection: close

Hello
```

HTTP / 2

HTTP / 2 (RFC 7540) a modifié le format «on the wire» de HTTP à partir d'une simple requête textuelle et des en-têtes de réponse au format de données binaires envoyé dans des trames. HTTP / 2 prend en charge la compression des en-têtes (HPACK).

Cela réduisait le temps système de requêtes et permettait de recevoir plusieurs réponses simultanément sur une seule connexion TCP / IP.

Malgré de gros changements dans le format des données, HTTP / 2 utilise toujours les en-têtes HTTP et les requêtes et réponses peuvent être traduites avec précision entre HTTP / 1.1 et 2.

HTTP / 0.9

La première version de HTTP qui a vu le jour est la 0.9, souvent appelée " HTTP as Implemented ". Une description courante de 0.9 est "un sous-ensemble du protocole HTTP [c'est-à-dire 1.0]". Cependant, cela ne permet pas d'illustrer la disparité des capacités entre 0,9 et 1,0.

Ni les requêtes ni les réponses dans les en-têtes de fonctionnalités 0.9. Les demandes consistent en une seule ligne de `GET` terminée par CRLF, suivie d'un espace, suivie de l'URL de ressource demandée. Les réponses doivent être un document HTML unique. La fin de ce document est marquée par la suppression de la connexion côté serveur. Il n'y a pas de facilités pour indiquer le succès ou l'échec d'une opération. La seule propriété interactive est la chaîne de recherche qui est étroitement liée à la `<isindex>` HTML `<isindex>` .

L'utilisation de HTTP / 0.9 est aujourd'hui exceptionnellement rare. Il est parfois vu sur les systèmes embarqués comme une alternative à TFTP .

Lire Démarrer avec HTTP en ligne: https://riptutorial.com/fr/http/topic/984/demarrer-avec-http

Chapitre 2: Authentification

Paramètres

Paramètre	Détails
Statut de réponse	`401` si le serveur d'origine nécessite une authentification, `407` si un proxy intermédiaire requiert une authentification
En-têtes de réponse	`WWW-Authenticate` par le serveur d'origine, `Proxy-Authenticate` par un proxy intermédiaire
En-têtes de demande	`Authorization` pour autorisation sur un serveur d'origine, `Proxy-Authorization` sur un proxy intermédiaire
Schéma d'authentification	`Basic` pour l'authentification de base, mais d'autres tels que `Digest` et `SPNEGO` peuvent être utilisés. Consultez le Registre des schémas d'authentification HTTP .
Domaine	Un nom de l'espace protégé sur le serveur; un serveur peut avoir plusieurs de ces espaces, chacun avec un nom distinct et des mécanismes d'authentification.
Lettres de créance	Pour `Basic` : nom d'utilisateur et mot de passe séparés par deux points, encodés en base64; Par exemple, `username:password` codé en base64 est `dXNlcm5hbWU6cGFzc3dvcmQ=`

Remarques

L'authentification de base est définie dans la RFC2617 . Il peut être utilisé pour s'authentifier auprès du serveur d'origine après la réception d'un serveur `401 Unauthorized` et d'un serveur proxy après un `407 (Proxy Authentication Required)` . Dans les informations d'identification (décodées), le mot de passe commence après le premier deux-points. Par conséquent, le nom d'utilisateur ne peut pas contenir de deux-points, mais le mot de passe le peut.

Examples

Authentification de base HTTP

L'authentification de base HTTP fournit un mécanisme simple d'authentification. Les informations d'identification sont envoyées en texte brut, et sont donc non sécurisées par défaut. L'authentification réussie se déroule comme suit.

Le client demande une page pour laquelle l'accès est restreint:

```
GET /secret
```

Le serveur répond avec le code d'état `401 Unauthorized` et demande au client de s'authentifier:

```
401 Unauthorized
WWW-Authenticate: Basic realm="Secret Page"
```

Le client envoie l'en-tête `Authorization` . Les identifiants sont `username:password` base64 codé:

```
GET /secret
Authorization: Basic dXNlcm5hbWU6cGFzc3dvcmQ=
```

Le serveur accepte les identifiants et répond avec le contenu de la page:

```
HTTP/1.1 200 OK
```

Lire Authentification en ligne: https://riptutorial.com/fr/http/topic/3286/authentification

Chapitre 3: Codes d'état HTTP

Introduction

Dans HTTP, les codes d'état sont un mécanisme lisible par une machine qui indique le résultat d'une demande précédemment émise. De RFC 7231, sec. 6 : "L'élément de code d'état est un code entier à trois chiffres donnant le résultat de la tentative de comprendre et de satisfaire la demande."

La grammaire formelle permet que les codes soient compris entre `000` et `999` . Cependant, seule la plage comprise entre `100` et `599` a attribué une signification.

Remarques

HTTP / 1.1 définit un nombre de codes d'état HTTP numériques qui apparaissent dans la ligne d'état - la première ligne d'une réponse HTTP - pour résumer ce que le client doit faire avec la réponse.

Le premier chiffre d'un code d'état définit la classe de la réponse:

- `1xx` informationnel
- `2xx` demande de client réussie
- `3xx` Request redirected - une autre action est nécessaire, telle qu'une nouvelle demande
- Erreur client `4xx` - ne pas répéter la même demande
- Erreur `5xx` Server - peut-être réessayer

En pratique, il n'est pas toujours facile de choisir le code de statut le plus approprié.

Examples

500 Erreur de serveur interne

Une **erreur de serveur interne HTTP 500** est un message général indiquant que le serveur a rencontré quelque chose d'inattendu. Les applications (ou le serveur Web global) doivent utiliser un 500 en cas d'erreur lors du traitement de la demande, c'est-à-dire qu'une exception est levée ou qu'une condition de la ressource empêche le processus de se terminer.

Exemple de ligne d'état:

```
HTTP/1.1 500 Internal Server Error
```

404 Introuvable

HTTP 404 non trouvé signifie que le serveur n'a pas pu trouver le chemin en utilisant l'URI demandé par le client.

```
HTTP/1.1 404 Not Found
```

Le plus souvent, le fichier demandé a été supprimé, mais il peut parfois s'agir d'une mauvaise configuration de la racine du document ou d'un manque d'autorisations (bien que les autorisations manquantes déclenchent plus fréquemment HTTP 403 Forbidden).

Par exemple, Microsoft IIS écrit 404.0 (0 est le sous-statut) dans ses fichiers journaux lorsque le fichier demandé a été supprimé. Mais lorsque la requête entrante est bloquée par des règles de filtrage des requêtes, elle écrit 404.5-404.19 dans des fichiers journaux en fonction de la règle qui bloque la requête. Une référence de code d'erreur plus détaillée est disponible sur le support Microsoft .

Refuser l'accès aux fichiers protégés

Utilisez 403 Interdit lorsqu'un client a demandé une ressource inaccessible en raison des contrôles d'accès existants. Par exemple, si votre application dispose d'un itinéraire `/admin` qui ne doit être accessible qu'aux utilisateurs disposant de droits d'administration, vous pouvez utiliser 403 lorsqu'un utilisateur normal demande la page.

```
GET /admin HTTP/1.1
Host: example.com
```

```
HTTP/1.1 403 Forbidden
```

Demande réussie

Envoyez une réponse HTTP avec le code d'état `200` pour indiquer une demande réussie. La ligne d'état de réponse HTTP est alors:

```
HTTP/1.1 200 OK
```

Le texte d'état `OK` n'est qu'informatif. Le corps de la réponse (charge utile du message) doit contenir une représentation de la ressource demandée. S'il n'y a pas de représentation 201 Aucun contenu ne doit être utilisé.

Répondre à une demande conditionnelle de contenu en cache

Envoyer un état de réponse **304 non modifié à** partir de l'envoi du serveur en réponse à une demande du client contenant des en `If-Modified-Since` têtes `If-Modified-Since` et `If-None-Match` , si la ressource de demande n'a pas changé.

Par exemple, si une demande de client pour une page Web inclut l'en-tête `If-Modified-Since: Fri, 22 Jul 2016 14:34:40 GMT` et que la page n'a pas été modifiée depuis, répondez avec la ligne d'état `HTTP/1.1 304 Not Modified` .

Top 10 HTTP Status Code

2xx succès

- **200 OK** - Réponse standard pour les requêtes HTTP réussies.
- **201 Créé** - La demande a été satisfaite, entraînant la création d'une nouvelle ressource.
- **204 No Content** - Le serveur a traité la demande avec succès et ne renvoie aucun contenu.

Redirection 3xx

- **304 Not Modified** - Indique que la ressource n'a pas été modifiée depuis la version spécifiée par les en `If-Modified-Since` têtes de requête `If-Modified-Since` ou `If-None-Match` .

Erreur client 4xx

- **400 Requête** incorrecte - Le serveur ne peut pas ou ne veut pas traiter la demande en raison d'une erreur apparente du client (par exemple, syntaxe de requête mal formée, taille trop grande, cadrage de message non valide ou acheminement de requête trompeuse).
- **401 Unauthorized** - *Similaire à 403 Interdit* , mais spécifiquement utilisé lorsque l'authentification est requise et a échoué ou n'a pas encore été fournie. La réponse doit inclure un champ d'en `WWW-Authenticate` tête `WWW-Authenticate` contenant un défi applicable à la ressource demandée.
- **403 Interdit** - La demande était une requête valide, mais le serveur refuse de répondre. L'utilisateur peut être connecté mais ne dispose pas des autorisations nécessaires pour la ressource.
- **404 Introuvable** - La ressource demandée est introuvable, mais peut être disponible ultérieurement. Les demandes ultérieures du client sont autorisées.
- **409 Conflit** - Indique que la demande n'a pas pu être traitée en raison d'un conflit dans la demande, tel qu'un conflit de modification entre plusieurs mises à jour simultanées.

Erreur serveur 5xx

- **500 Internal Server Error** - Un message d'erreur générique, donné lorsqu'une condition inattendue s'est produite et qu'aucun message plus spécifique n'est approprié.

Lire Codes d'état HTTP en ligne: https://riptutorial.com/fr/http/topic/2577/codes-d-etat-http

Chapitre 4: Encodage des réponses et compression

Examples

Compression HTTP

Le corps du message HTTP peut être compressé (depuis HTTP / 1.1). Soit par le serveur compresse la demande et ajoute un en `Content-Encoding` tête `Content-Encoding` , soit par un proxy fait et ajoute un en `Transfer-Encoding` tête `Transfer-Encoding` .

Un client peut envoyer un en `Accept-Encoding` tête de demande `Accept-Encoding` pour indiquer les codages qu'il accepte.

Les encodages les plus couramment utilisés sont:

- algorithme gzip - deflate (LZ77) avec la somme de contrôle CRC32 implémentée dans le programme de compression du fichier "gzip" (RFC1952)
- dégonfler - format de données "zlib" (RFC1950), algorithme de dégonflage (hybride LZ77 et Huffman) avec somme de contrôle Adler32

Méthodes de compression multiples

Il est possible de compresser plusieurs fois un corps de message de réponse HTTP. Les noms de codage doivent ensuite être séparés par une virgule dans l'ordre dans lequel ils ont été appliqués. Par exemple, si un message a été compressé via deflate puis gzip, l'en-tête devrait ressembler à ceci:

```
Content-Encoding: deflate, gzip
```

Plusieurs en `Content-Encoding` têtes `Content-Encoding` sont également valides, mais pas recommandés:

```
Content-Encoding: deflate
Content-Encoding: gzip
```

compression gzip

Le client envoie d'abord une requête avec un en `Accept-Encoding` tête `Accept-Encoding` qui indique qu'il prend en charge gzip:

```
GET / HTTP/1.1\r\n
Host: www.google.com\r\n
Accept-Encoding: gzip, deflate\r\n
\r\n
```

Le serveur peut alors envoyer une réponse avec un corps de réponse compressé et un en `Content-Encoding` tête `Content-Encoding` qui spécifie que le codage gzip a été utilisé:

```
HTTP/1.1 200 OK\r\n
Content-Encoding: gzip\r\n
Content-Length: XX\r\n
\r\n
... compressed content ...
```

Lire Encodage des réponses et compression en ligne:
https://riptutorial.com/fr/http/topic/5046/encodage-des-reponses-et-compression

Chapitre 5: HTTP pour les API

Remarques

Les API HTTP utilisent un large éventail de verbes HTTP et retournent généralement des réponses JSON ou XML.

Examples

Créer une ressource

Tout le monde n'est pas d'accord sur la méthode la plus sémantiquement correcte pour la création de ressources. Ainsi, votre API pourrait accepter les requêtes `POST` ou `PUT` , ou l'une ou l'autre.

Le serveur doit répondre par `201 Created` si la ressource a été créée avec succès. Choisissez le code d'erreur le plus approprié s'il ne l'était pas.

Par exemple, si vous fournissez une API pour créer des enregistrements d'employés, la demande / réponse peut ressembler à ceci:

```
POST /employees HTTP/1.1
Host: example.com
Content-Type: application/json

{
    "name": "Charlie Smith",
    "age": 38,
    "job_title": "Software Developer",
    "salary": 54895.00
}
```

```
HTTP/1.1 201 Created
Location: /employees/1/charlie-smith
Content-Type: application/json

{
    "employee": {
        "name": "Charlie Smith",
        "age": 38,
        "job_title": "Software Developer",
        "salary": 54895.00
        "links": [
            {
                "uri": "/employees/1/charlie-smith",
                "rel": "self",
                "method": "GET"
            },
            {
                "uri": "/employees/1/charlie-smith",
                "rel": "delete",
                "method": "DELETE"
            },
```

```
            {
                "uri": "/employees/1/charlie-smith",
                "rel": "edit",
                "method": "PATCH"
            }
        ]
    },
    "links": [
        {
            "uri": "/employees",
            "rel": "create",
            "method": "POST"
        }
    ]
}
```

L'inclusion des champs JSON des `links` dans la réponse permet au client d'accéder aux ressources liées à la nouvelle ressource et à l'application dans son ensemble, sans avoir à connaître leurs URI ou méthodes au préalable.

Modifier une ressource

La modification ou la mise à jour d'une ressource est un objectif commun des API. Les modifications peuvent être effectuées en envoyant des requêtes `POST` , `PUT` ou `PATCH` à la ressource respective. Bien que `POST` soit autorisé à ajouter des données à la représentation existante d'une ressource, il est recommandé d'utiliser `PUT` ou `PATCH` car elles transmettent une sémantique plus explicite.

Votre serveur doit répondre avec `200 OK` si la mise à jour a été effectuée ou `202 Accepted` si l'application n'a pas encore été effectuée. Choisissez le code d'erreur le plus approprié s'il ne peut pas être complété.

Mises à jour complètes

`PUT` a la sémantique de remplacer la représentation actuelle par la charge utile incluse dans la requête. Si la charge utile n'est pas du même type de représentation que la représentation actuelle de la ressource à mettre à jour, le serveur peut décider de l'approche à suivre. RFC7231 définit que le serveur peut soit

- Reconfigurer la ressource cible pour refléter le nouveau type de média
- Transforme la représentation PUT en un format compatible avec celui de la ressource avant de l'enregistrer en tant que nouvel état de ressource
- Rejeter la demande avec une réponse `415 Unsupported Media Type` indiquant que la ressource cible est limitée à un ensemble (spécifique) de types de média.

Une ressource de base contenant une représentation JSON HAL comme ...

```
{
    "name": "Charlie Smith",
    "age": 39,
```

```
    "job_title": "Software Developer",
    "_links": {
        "self": { "href": "/users/1234" },
        "employee": { "href": "http://www.acmee.com" },
        "curies": [{ "name": "ea", "href": "http://www.acmee.com/docs/rels/{rel}", templated":
true}],
        "ea:admin": [
            "href": "/admin/2",
            "title": "Admin"
        ]
    }
}
```

... peut recevoir une demande de mise à jour comme celle-ci

```
PUT /users/1234 HTTP/1.1
Host: http://www.acmee.com
Content-Type: "application/json; charset=utf-8"
Content-Length: 85
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)

{
    "name": "Charlie Gold-Smith",
    "age": 40,
    "job_title": "Senior Software Developer"
}
```

Le serveur peut maintenant remplacer l'état de la ressource par le corps de requête donné et changer le type de contenu de `application/hal+json` à `application/json` ou convertir le contenu JSON en représentation JSON HAL, puis remplacer le contenu de la ressource. avec la transformée ou rejeter la demande de mise à jour en raison d'un type de support inaplicable avec une réponse `415 Unsupported Media Type` type de support `415 Unsupported Media Type` .

Il y a une différence entre remplacer le contenu directement ou d'abord transformer la représentation en un modèle de représentation défini, puis remplacer le contenu existant par le contenu transformé. Une demande `GET` ultérieure renverra la réponse suivante lors d'un remplacement direct:

```
GET /users/1234 HTTP/1.1
Host: http://www.acmee.com
Accept-Encoding: gzip, deflate
Accept-Language: en-us
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
ETag: "e0023aa4e"

{
    "name": "Charlie Gold-Smith",
    "age": 40,
    "job_title": "Senior Software Developer"
}
```

tandis que l'approche de transformation puis de remplacement renverra la représentation suivante:

```
GET /users/1234 HTTP/1.1
Host: http://www.acmee.com
Accept-Encoding: gzip, deflate
Accept-Language: en-us
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
ETag: e0023aa4e

{
    "name": "Charlie Gold-Smith",
    "age": 40,
    "job_title": "Senior Software Developer",
    "_links": {
        "self": { "href": "/users/1234" },
        "employee": { "href": "http://www.acmee.com" },
        "curies": [{ "name": "ea", "href": "http://www.acmee.com/docs/rels/{rel}", templated":
true}],
        "ea:admin": [
            "href": "/admin/2",
            "title": "Admin"
        ]
    }
}
```

Effets secondaires

Notez que `PUT` est autorisé à avoir des effets secondaires bien qu'il soit défini comme un fonctionnement idempotent! Ceci est documenté dans RFC7231 comme

> Une demande PUT appliquée à la ressource cible **peut avoir des effets secondaires sur d'autres ressources** . Par exemple, un article peut avoir une URI pour identifier "la version actuelle" (une ressource) distincte des URI identifiant chaque version particulière (des ressources différentes partageant à un moment donné le même état que la ressource de version actuelle). Une demande PUT réussie sur l'URI de la "version actuelle" peut donc créer une nouvelle ressource de version en plus de modifier l'état de la ressource cible et peut également entraîner l'ajout de liens entre les ressources associées.

Produire des entrées de journal supplémentaires n'est généralement pas considéré comme un effet secondaire, car il ne s'agit certainement pas d'un état d'une ressource en général.

Mises à jour partielles

RFC7231 mentionne ceci concernant les mises à jour partielles:

> Les mises à jour de contenu partielles sont possibles en ciblant une ressource identifiée séparément avec un état recouvrant une partie de la ressource plus grande ou en utilisant une méthode différente qui a été spécifiquement définie pour les mises à jour partielles (par exemple, la méthode PATCH définie dans RFC5789).

Les mises à jour partielles peuvent donc être effectuées en deux versions:

- Demander à une ressource d'intégrer plusieurs sous-ressources plus petites et de ne mettre à jour qu'une sous-ressource respective au lieu de la ressource complète via `PUT`
- Utiliser `PATCH` et indiquer au serveur quoi mettre à jour

Mise à jour partielle avec état superposé

Si une représentation d'utilisateur doit être partiellement mise à jour en raison du déplacement d'un utilisateur vers un autre emplacement, au lieu de mettre à jour l'utilisateur directement, la ressource associée doit être mise à jour directement, ce qui correspond à une mise à jour partielle de la représentation de l'utilisateur.

Avant le déplacement, un utilisateur avait la représentation suivante

```
GET /users/1234 HTTP/1.1
Host: http://www.acmee.com
Accept: application/hal+json; charset=utf-8
Accept-Encoding: gzip, deflate
Accept-Language: en-us
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
ETag: "e0023aa4e"

{
    "name": "Charlie Gold-Smith",
    "age": 40,
    "job_title": "Senior Software Developer",
    "_links": {
        "self": { "href": "/users/1234" },
        "employee": { "href": "http://www.acmee.com" },
        "curies": [{ "name": "ea", "href": "http://www.acmee.com/docs/rels/{rel}", templated":
true}],
        "ea:admin": [
            "href": "/admin/2",
            "title": "Admin"
        ]
    },
    "_embedded": {
        "ea:address": {
            "street": "Terrace Drive, Central Park",
            "zip": "NY 10024"
            "city": "New York",
            "country": "United States of America",
            "_links": {
                "self": { "href": "/address/abc" },
                "google_maps": { "href": "http://maps.google.com/?ll=40.7739166,-73.970176" }
            }
        }
    }
}
```

Lorsque l'utilisateur se déplace vers un nouvel emplacement, il met à jour ses informations de localisation comme suit:

```
PUT /address/abc HTTP/1.1
Host: http://www.acmee.com
Content-Type: "application/json; charset=utf-8"
```

```
Content-Length: 109
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)

{
    "street": "Standford Ave",
    "zip": "CA 94306",
    "city": "Pablo Alto",
    "country": "United States of America"
}
```

Avec la sémantique de transformation-avant-remplacement pour le type de média incompatible entre la ressource d'adresse existante et celle de la requête, comme décrit ci-dessus, la ressource d'adresse est maintenant mise à jour et a pour conséquence qu'une requête `GET` ultérieure sur la ressource utilisateur la nouvelle adresse de l'utilisateur est renvoyée.

```
GET /users/1234 HTTP/1.1
Host: http://www.acmee.com
Accept: application/hal+json; charset=utf-8
Accept-Encoding: gzip, deflate
Accept-Language: en-us
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
ETag: "e0023aa4e"

{
    "name": "Charlie Gold-Smith",
    "age": 40,
    "job_title": "Senior Software Developer",
    "_links": {
        "self": { "href": "/users/1234" },
        "employee": { "href": "http://www.acmee.com" },
        "curies": [{ "name": "ea", "href": "http://www.acmee.com/docs/rels/{rel}", templated":
true}],
        "ea:admin": [
            "href": "/admin/2",
            "title": "Admin"
        ]
    },
    "_embedded": {
        "ea:address": {
            "street": "Standford Ave",
            "zip": "CA 94306",
            "city": "Pablo Alto",
            "country": "United States of America"
            "_links": {
                "self": { "href": "/address/abc" },
                "google_maps": { "href": "http://maps.google.com/?ll=37.4241311,-122.1524475"
}
            }
        }
    }
}
```

Correction de données partielles

`PATCH` est défini dans RFC5789 et ne fait pas directement partie des spécifications HTTP en soi. Une erreur courante consiste à dire que l' envoi de champs uniquement à mettre à jour est

suffisant dans une requête `PATCH` . La spécification indique donc

> La méthode PATCH demande qu'un ensemble de modifications décrites dans l'entité de demande soit appliqué à la ressource identifiée par l'URI de demande. L'ensemble des modifications est représenté dans un format appelé "document de correctif" identifié par un type de média.

Cela signifie qu'un client doit calculer les étapes nécessaires à la transformation de la ressource de l'état A à l'état B et envoyer ces instructions au serveur.

JSON Patch est un type de média populaire basé sur JSON .

Si l'âge et le titre de travail de notre exemple d'utilisateur changent et qu'un champ supplémentaire représentant le revenu de l'utilisateur doit être ajouté, une mise à jour partielle à l'aide de `PATCH` utilisant JSON Patch peut ressembler à ceci:

```
PATCH /users/1234 HTTP/1.1
Host: http://www.acmee.com
Content-Type: application/json-patch+json; charset=utf-8
Content-Length: 188
Accept: application/json
If-Match: "e0023aa4e"

[
    { "op": "replace", "path": "/age", "value": 40 },
    { "op": "replace", "path": "/job_title", "value": "Senior Software Developer" },
    { "op": "add", "path": "/salery", "value": 63985.00 }
]
```

`PATCH` peut mettre à jour plusieurs ressources à la fois et appliquer les modifications de manière atomique, ce qui signifie que toutes les modifications doivent être appliquées ou aucune, ce qui impose un fardeau transactionnel à l'implémenteur de l'API.

Une mise à jour réussie peut renvoyer quelque chose comme ceci

```
HTTP/1.1 200 OK
Location: /users/1234
Content-Type: application/json
ETag: "df00eb258"

{
    "name": "Charlie Smith",
    "age": 40,
    "job_title": "Senior Software Developer",
    "salary": 63985.00
}
```

mais n'est pas limité à `200 OK` codes de réponse `200 OK` seulement.

Pour éviter les mises à jour intermédiaires (modifications effectuées entre la récupération précédente de l'état de représentation et la mise à jour), l'en `If-Unmodified-Since` tête `ETag` , `If-Match` ou `If-Unmodified-Since` doit être utilisé.

La gestion des erreurs

La spécification de `PATCH` recommande la gestion des erreurs suivante:

Type	Code d'erreur
Document de correctif mal formé	`400 Bad Request`
Document de correctif non pris en charge	`415 Unsupported Media Type`
Demande non traitable, c.-à-d. Si le ressource devient invalide en appliquant le correctif	`422 Unprocessable Entity`
Ressource introuvable	`404 Not Found`
Etat conflictuel, c'est-à-dire renommer (déplacer) un champ qui n'existe pas	`409 Conflict`
Modification en conflit, c'est-à-dire si le client utilise un en `If-Unmodified-Since` tête `If-Match` ou `If-Unmodified-Since` dont la validation a échoué. Si aucune condition préalable n'était disponible, le dernier code d'erreur devrait être renvoyé	`412 Precondition Failed` ou `409 Conflict`
Modification concomitante, c.-à-d. Si la demande doit être appliquée avant l'acception, les demandes `PATCH` supplémentaires	`409 Conflict`

Supprimer une ressource

Une autre utilisation courante des API HTTP consiste à supprimer une ressource existante. Cela devrait normalement être fait en utilisant les requêtes `DELETE` .

Si la suppression a réussi, le serveur doit renvoyer `200 OK` ; un code d'erreur approprié s'il ne l'était pas.

Si notre employé Charlie Smith a quitté l'entreprise et que nous voulons maintenant supprimer ses dossiers, cela pourrait ressembler à ceci:

```
DELETE /employees/1/charlie-smith HTTP/1.1
Host: example.com
```

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    'links': [
        {
            'uri': '/employees',
            'rel': 'create',
            'method': 'POST'
```

```
        }
    ]
}
```

Liste des ressources

La dernière utilisation commune des API HTTP consiste à obtenir une liste des ressources existantes sur le serveur. De telles listes doivent être obtenues en utilisant `GET` requêtes `GET` , car elles ne *récupèrent que des* données.

Le serveur doit renvoyer `200 OK` s'il peut fournir la liste, ou un code d'erreur approprié si ce n'est pas le cas.

Voici une liste de nos employés:

```
GET /employees HTTP/1.1
Host: example.com
```

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    'employees': [
        {
            'name': 'Charlie Smith',
            'age': 39,
            'job_title': 'Software Developer',
            'salary': 63985.00
            'links': [
                {
                    'uri': '/employees/1/charlie-smith',
                    'rel': 'self',
                    'method': 'GET'
                },
                {
                    'uri': '/employees/1/charlie-smith',
                    'rel': 'delete',
                    'method': 'DELETE'
                },
                {
                    'uri': '/employees/1/charlie-smith',
                    'rel': 'edit',
                    'method': 'PATCH'
                }
            ]
        },
        {
            'name': 'Donna Prima',
            'age': 30,
            'job_title': 'QA Tester',
            'salary': 77095.00
            'links': [
                {
                    'uri': '/employees/2/donna-prima',
                    'rel': 'self',
                    'method': 'GET'
                },
```

```
                {
                    'uri': '/employees/2/donna-prima',
                    'rel': 'delete',
                    'method': 'DELETE'
                },
                {
                    'uri': '/employees/2/donna-prima',
                    'rel': 'edit',
                    'method': 'PATCH'
                }
            ]
        }
    ],
    'links': [
        {
            'uri': '/employees/new',
            'rel': 'create',
            'method': 'PUT'
        }
    ]
}
```

Lire HTTP pour les API en ligne: https://riptutorial.com/fr/http/topic/3423/http-pour-les-api

Chapitre 6: Mise en cache des réponses HTTP

Remarques

Les réponses sont mises en cache séparément pour chaque URL et chaque méthode HTTP.

La mise en cache HTTP est définie dans la RFC 7234 .

Glossaire

- **fresh** - état d'une réponse mise en cache, qui n'a pas encore expiré. En règle générale, une nouvelle réponse peut *satisfaire des* demandes sans qu'il soit nécessaire de contacter le serveur dont la réponse provient.
- **stale** - état d'une réponse mise en cache, qui a dépassé sa date d'expiration. En règle générale, les réponses obsolètes ne peuvent pas être utilisées pour *satisfaire* une demande sans vérifier auprès du serveur si elle est toujours valide.
- **satisfaire** - réponse en cache *satisfait* une demande lorsque toutes les conditions de la demande correspondent à la réponse mise en cache, par exemple , ils ont la même méthode HTTP et URL, la réponse est frais ou la demande permet des réponses périmées, têtes de requête correspondent en- têtes listés en réponse de `Vary` - tête, etc. .
- **revalidation** - vérifie si une réponse en cache est fraîche. Cela se fait généralement avec une *requête conditionnelle* contenant `If-Modified-Since` ou `If-None-Match` et le statut de réponse `304` .
- **cache privé** - cache pour un seul utilisateur, par exemple dans un navigateur Web. Les caches privées peuvent stocker des réponses personnalisées.
- **cache public** - cache partagé entre de nombreux utilisateurs, par exemple sur un serveur proxy. Un tel cache peut envoyer la même réponse à plusieurs utilisateurs.

Examples

Réponse cache pour tout le monde pendant 1 an

```
Cache-Control: public, max-age=31536000
```

`public` signifie que la réponse est la même pour tous les utilisateurs (elle ne contient aucune information personnalisée). `max-age` est en secondes à partir de maintenant. 31536000 = 60 * 60 * 24 * 365.

Ceci est recommandé pour les actifs statiques qui ne sont jamais destinés à changer.

Cache personnalisé réponse pendant 1 minute

```
Cache-Control: private, max-age=60
```

`private` spécifie que la réponse ne peut être mise en cache que pour l'utilisateur qui a demandé la ressource et ne peut pas être réutilisée lorsque d'autres utilisateurs demandent la même ressource. Ceci est approprié pour les réponses qui dépendent des cookies.

Arrêtez l'utilisation des ressources mises en cache sans vérifier d'abord avec le serveur

```
Cache-Control: no-cache
```

Le client se comportera comme si la réponse n'était pas mise en cache. Ceci est approprié pour les ressources qui peuvent changer de manière imprévisible à tout moment et que les utilisateurs doivent toujours voir dans la dernière version.

Les réponses avec `no-cache` seront plus lentes (latence élevée) en raison de la nécessité de contacter le serveur chaque fois qu'elles sont utilisées.

Cependant, pour économiser de la bande passante, les clients *peuvent* toujours stocker ces réponses. Les réponses avec `no-cache` ne seront pas utilisées pour satisfaire les demandes sans contacter le serveur à chaque fois pour vérifier si la réponse mise en cache peut être réutilisée.

Demander des réponses à ne pas stocker du tout

```
 Cache-control: no-store
```

Indique aux clients de ne pas mettre la réponse en cache de quelque manière que ce soit et de l’oublier le plus rapidement possible.

Cette directive a été conçue à l'origine pour les données sensibles (aujourd'hui, HTTPS devrait être utilisé à la place), mais peut être utilisé pour éviter de polluer les caches avec des réponses qui ne peuvent pas être réutilisées.

Il convient uniquement dans des cas spécifiques où les données de réponse sont toujours différentes, par exemple un point de terminaison d'API renvoyant un grand nombre aléatoire. Sinon, `no-cache` et la revalidation peuvent être utilisés pour avoir un comportement de réponse "irréversible", tout en pouvant économiser de la bande passante.

En-têtes obsolètes, redondants et non standard

- `Expires` - spécifie la date à laquelle la ressource devient obsolète. Il repose sur des serveurs et des clients disposant d’horloges précises et de fuseaux horaires appropriés. `Cache-control: max-age` a priorité sur `Expires` et est généralement plus fiable.
- `post-check` directives `post-check` et `pre-check` sont des extensions non standard d'Internet Explorer qui permettent d'utiliser des réponses obsolètes. L'alternative standard est `stale-while-revalidate` .

- `Pragma: no-cache` - obsolète en 1999 . `Cache-control` doit être utilisé à la place.

Modification des ressources en cache

La méthode la plus simple pour contourner le cache consiste à modifier l'URL. Ceci est utilisé comme une bonne pratique lorsque l'URL contient une version ou une somme de contrôle de la ressource, par exemple

```
http://example.com/image.png?version=1
http://example.com/image.png?version=2
```

Ces deux URL seront mises en cache séparément, donc même si `…?version=1` était mis en cache *pour toujours* , une nouvelle copie pourrait être immédiatement récupérée sous la forme `…?version=2` .

Veuillez ne pas utiliser d'URL aléatoires pour contourner les caches. Utilisez `Cache-control: no-cache` ou `Cache-control: no-store` place. Si des réponses avec des URL aléatoires sont envoyées sans la directive `no-store` , elles seront stockées inutilement dans des caches et expulseront des réponses plus utiles du cache, dégradant les performances de la totalité du cache.

Lire Mise en cache des réponses HTTP en ligne: https://riptutorial.com/fr/http/topic/3296/mise-en-cache-des-reponses-http

Chapitre 7: Origine croisée et contrôle d'accès

Remarques

Le partage de ressources entre origines est conçu pour autoriser les requêtes dynamiques entre domaines, en utilisant souvent des techniques telles que AJAX . Pendant que le script effectue la majeure partie du travail, le serveur HTTP doit prendre en charge la requête en utilisant les en-têtes appropriés.

Examples

Client: envoi d'une requête de partage de ressources d'origine croisée (CORS)

Une *demande d'* origine croisée doit être envoyée, y compris l'en-tête d' `Origin` . Cela indique d'où provient la demande. Par exemple, une requête d'origine croisée de `http://example.com` à `http://example.org` ressemblerait à ceci:

```
GET /cors HTTP/1.1
Host: example.org
Origin: example.com
```

Le serveur utilisera cette valeur pour déterminer si la demande est autorisée.

Serveur: répondre à une requête CORS

La réponse à une demande CORS doit inclure un en `Access-Control-Allow-Origin` tête `Access-Control-Allow-Origin` , qui détermine les origines autorisées à utiliser la ressource CORS. Cet en-tête peut prendre l'une des trois valeurs suivantes:

- Une origine Cela permet *uniquement les* demandes de *cette origine* .
- Le personnage `*` . Cela permet des demandes de *toute origine* .
- La chaîne `null` . Cela *ne* permet *aucune demande CORS* .

Par exemple, à la réception d'une demande CORS provenant de l'origine `http://example.com` , si `example.com` est une origine autorisée, le serveur renvoie cette réponse:

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: example.com
```

Une réponse d'origine quelconque permettrait également cette requête, à savoir:

```
HTTP/1.1 200 OK
```

```
Access-Control-Allow-Origin: *
```

Autoriser les informations d'identification ou la session de l'utilisateur

Autoriser l'envoi d'informations d'identification utilisateur ou la session de l'utilisateur avec une requête CORS permet au serveur de conserver les données utilisateur sur les requêtes CORS. Cela est utile si le serveur doit vérifier si l'utilisateur est connecté avant de fournir des données (par exemple, si vous effectuez une action uniquement si un utilisateur est connecté, cela nécessitera l'envoi de la requête CORS avec les informations d'identification).

Cela peut être réalisé côté serveur pour les requêtes en amont, en envoyant l'en `Access-Control-Allow-Credentials` tête `Access-Control-Allow-Credentials` en réponse à la demande de contrôle en amont `OPTIONS` . Prenez le cas suivant d'une requête CORS pour `DELETE` une ressource:

```
OPTIONS /cors HTTP/1.1
Host: example.com
Origin: example.org
Access-Control-Request-Method: DELETE
```

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: example.org
Access-Control-Allow-Methods: DELETE
Access-Control-Allow-Credentials: true
```

La ligne `Access-Control-Allow-Credentials: true` indique que la demande `DELETE` CORS suivante peut être envoyée avec les informations d'identification de l'utilisateur.

Demandes de contrôle en amont

Une requête CORS de base est autorisée à utiliser l'une des deux méthodes suivantes:

- OBTENIR
- POSTER

et seulement quelques en-têtes de sélection. Les requêtes POST CORS peuvent en outre choisir parmi seulement trois types de contenu.

Pour éviter ce problème, les demandes qui souhaitent utiliser d'autres méthodes, en-têtes ou types de contenu doivent d'abord émettre une demande de *contrôle en amont* , qui est une demande `OPTIONS` incluant des en-têtes de demande de contrôle d'accès. Par exemple, il s'agit d'une demande de contrôle en amont qui vérifie si le serveur acceptera une demande `PUT` incluant un en-tête `DNT` :

```
OPTIONS /cors HTTP/1.1
Host: example.com
Origin: example.org
Access-Control-Request-Method: PUT
Access-Control-Request-Headers: DNT
```

Serveur: répondre aux demandes de contrôle en amont

Lorsqu'un serveur reçoit une demande de contrôle en amont, il doit vérifier s'il prend en charge la méthode et les en-têtes requis et renvoyer une réponse indiquant sa capacité à prendre en charge la demande, ainsi que toute autre donnée autorisée (telle que les informations d'identification).

Celles-ci sont indiquées dans les en-têtes Autoriser le contrôle d'accès. Le serveur peut également renvoyer un en-tête `Max-Age` contrôle d'accès, indiquant la durée pendant laquelle la réponse en amont peut être mise en cache.

Voici à quoi peut ressembler un cycle demande-réponse pour une demande de contrôle en amont:

```
OPTIONS /cors HHTP/1.1
Host: example.com
Origin: example.org
Access-Control-Request-Method: PUT
Access-Control-Request-Headers: DNT
```

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: example.org
Access-Control-Allow-Methods: PUT
Access-Control-Allow-Headers: DNT
```

Lire Origine croisée et contrôle d'accès en ligne: https://riptutorial.com/fr/http/topic/3424/origine-croisee-et-controle-d-acces

Chapitre 8: Réponses HTTP

Paramètres

Code de statut	Phrase de motivation - Description
100	**Continuer** - le client doit envoyer la partie suivante d'une requête en plusieurs parties.
101	**Changer de protocole** - le serveur modifie la version ou le type de protocole utilisé dans cette communication.
200	**OK** - le serveur a reçu et complété la demande du client.
201	**Créé** - le serveur a accepté la demande et créé une nouvelle ressource, disponible sous l'URI dans l'en-tête `Location` .
202	**Accepté** - le serveur a reçu et accepté la demande du client, mais il n'a pas encore démarré ni terminé le traitement.
203	**Informations non autorisées** - le serveur renvoie des données pouvant être un sous-ensemble ou un surensemble des informations disponibles sur le serveur d'origine. Principalement utilisé par les mandataires.
204	**Pas de contenu** - utilisé à la place de 200 (OK) lorsqu'il n'y a pas de corps dans la réponse.
205	**Réinitialiser le contenu** - identique à 204 (aucun contenu), mais le client doit recharger la vue de document active.
206	**Contenu partiel** - utilisé à la place de 200 (OK) lorsque le client a demandé un en-tête `Range` .
300	**Choix multiples** - la ressource demandée est disponible sur plusieurs URI et le client doit rediriger la demande vers un URI spécifié dans la liste du corps du message.
301	**Déplacé de manière permanente** - la ressource demandée n'est plus disponible sur cet URI et le client doit rediriger cette requête et toutes les futures requêtes vers l'URI spécifié dans l'en-tête `Location` .
302	**Trouvé** - la ressource réside temporairement sous un URI différent. Cette demande doit être redirigée sur la confirmation de l'utilisateur vers l'URI dans l'en-tête `Location` , mais les futures demandes ne doivent pas être modifiées.
303	**Voir Autres** - très similaire à 302 (trouvé), mais ne nécessite pas de saisie de

Code de statut	Phrase de motivation - Description
	l'utilisateur pour rediriger vers l'URI fourni. L'URI fourni doit être récupéré avec une requête GET.
304	**Non modifié** - le client a envoyé un en `If-Modified-Since` tête `If-Modified-Since` ou similaire, et la ressource n'a pas été modifiée depuis ce point; le client doit afficher une copie en cache de la ressource.
305	**Use Proxy** - La ressource demandée doit être à nouveau demandée via le proxy spécifié dans le champ d'en-tête `Location` .
307	**Redirect temporaire** - identique à 302 (trouvé), mais les clients HTTP 1.0 ne prennent pas en charge 307 réponses.
400	**Mauvaise demande** - le client a envoyé une demande mal formée contenant des erreurs de syntaxe et doit modifier la demande pour corriger ceci avant de le répéter.
401	**Non autorisé** - la ressource demandée n'est pas disponible sans authentification. Le client peut répéter la demande en utilisant un en-tête d' `Authorization` pour fournir des détails d'authentification.
402	**Paiement requis** - code de statut réservé, non spécifié, à utiliser par les applications nécessitant un abonnement utilisateur pour afficher le contenu.
402	**Interdit** - le serveur comprend la requête, mais refuse de la remplir en raison des contrôles d'accès existants. La demande ne doit pas être répétée.
404	**Introuvable** - aucune ressource disponible sur ce serveur ne correspond à l'URI demandé. Peut être utilisé à la place de 403 pour éviter d'exposer les détails du contrôle d'accès.
405	**Méthode non autorisée** - la ressource ne prend pas en charge la méthode de requête (verbe HTTP); l'en-tête `Allow` liste les méthodes de requête acceptables.
406	**Non acceptable** - La ressource possède des caractéristiques qui violent les en-têtes d'acceptation envoyés dans la demande.
407	**Authentification proxy requise** - similaire à 401 (non autorisé), mais indique que le client doit d'abord s'authentifier auprès du proxy intermédiaire.
408	**Délai d'expiration de la demande** - le serveur attendait une autre demande du client, mais aucune n'était fournie dans un délai acceptable.
409	**Conflit** - la demande n'a pas pu être complétée car elle était en conflit avec l'état actuel de la ressource.
410	**Gone** - similaire à 404 (non trouvé), mais indique un retrait permanent. Aucune

Code de statut	Phrase de motivation - Description
	adresse de transfert n'est disponible.
411	**Longueur requise** - le client n'a pas spécifié d'en `Content-Length` tête `Content-Length` valide et doit le faire avant que le serveur accepte cette demande.
412	**Condition préalable Echec** - la ressource n'est pas disponible avec toutes les conditions spécifiées par les en-têtes conditionnels envoyés par le client.
413	**Entité de demande trop grande** - le serveur est actuellement incapable de traiter un corps de message de la longueur que le client a envoyé.
414	**Request-URI Too Long** - Le serveur refuse la requête car l'URI de demande est plus long que ce que le serveur est prêt à interpréter.
415	**Type de** support **non pris en charge** - le serveur ne prend pas en charge le type MIME ou de média spécifié par le client et ne peut pas traiter cette demande.
416	**Plage demandée non satisfaite** - le client a demandé une plage d'octets, mais le serveur ne peut pas fournir de contenu à cette spécification.
417	**Echec de l'attente** - les contraintes spécifiées par le client dans l'en-tête `Expect` que le serveur ne peut pas rencontrer.
500	**Erreur de serveur interne** - le serveur a rencontré une condition ou une erreur inattendue qui l'empêche de répondre à cette requête.
501	**Non implémenté** - le serveur ne prend pas en charge la fonctionnalité requise pour exécuter la demande. Généralement utilisé pour indiquer une méthode de requête qui n'est prise en charge sur *aucune* ressource.
502	**Passerelle** incorrecte - le serveur est un proxy et a reçu une réponse non valide du serveur en amont lors du traitement de cette requête.
503	**Service indisponible** - le serveur est soumis à une charge élevée ou est en cours de maintenance, et n'a pas la capacité de répondre à cette demande pour le moment.
504	**Délai d'expiration de la passerelle** - le serveur est un proxy et n'a pas reçu de réponse du serveur en amont à temps.
505	**Version HTTP non prise en charge** - le serveur ne prend pas en charge la version du protocole HTTP avec laquelle le client a fait sa demande.

Examples

Format de réponse de base

Lorsqu'un serveur HTTP reçoit une requête HTTP bien formée, il doit traiter les informations contenues dans la demande et renvoyer une réponse au client. Une simple réponse HTTP 1.1 peut ressembler à l'une des suivantes, généralement suivie d'un certain nombre de champs d'en-tête et éventuellement d'un corps de réponse:

```
HTTP/1.1 200 OK \r\n
```

```
HTTP/1.1 404 Not Found \r\n
```

```
HTTP/1.1 503 Service Unavailable \r\n
```

Une simple réponse HTTP 1.1 a ce format:

```
HTTP-Version Status-Code Reason-Phrase CRLF
```

Comme dans une requête, `HTTP-Version` indique la version du protocole HTTP utilisée; pour HTTP 1.1, cela doit toujours être la chaîne `HTTP/1.1` .

`Status-Code` est un `Status-Code` trois chiffres qui indique l'état de la demande du client. Le premier chiffre de ce code est la *classe de statut* , qui place le code de statut dans l'une des 5 catégories de réponse [1] :

- `1xx` **Informational** - le serveur a reçu la demande et le traitement se poursuit
- `2xx` **Success** - le serveur a accepté et traité la demande
- **Redirection** `3xx` - une action supplémentaire est nécessaire chez le client pour compléter la demande
- **Erreurs client** `4xx` - le client a envoyé une requête mal formée ou ne pouvant être satisfaite
- `5xx` **erreurs de serveur** - la requête était valide, mais le serveur ne peut pas le remplir pour le moment

`Reason-Phrase` est une courte description du code d'état. Par exemple, le code `200` a une phrase de raison de `OK` ; le code `404` a une phrase `Not Found` . Une liste complète des phrases de motif est disponible dans Paramètres, ci-dessous ou dans la spécification HTTP .

La ligne se termine par un retour chariot - un saut de ligne, généralement représenté par `\r\n` .

En-têtes supplémentaires

Comme une requête HTTP, une réponse HTTP peut inclure des en-têtes supplémentaires pour modifier ou augmenter la réponse fournie.

Une liste complète des en-têtes disponibles est définie au §6.2 de la spécification . Les en-têtes les plus couramment utilisés sont:

- `Server` , qui fonctionne comme un en `User-Agent` tête de requête `User-Agent` pour le serveur;
- `Location` utilisé sur les réponses d'état 201 et 3xx pour indiquer un URI à rediriger; et

- `ETag` , qui est un identificateur unique pour cette version de la ressource renvoyée pour permettre aux clients de mettre en cache la réponse.

Les en-têtes de réponse viennent après la ligne d'état et, comme avec les en- têtes de requête, sont formés comme suit:

```
Name: Value CRLF
```

`Name` fournit le nom de l'en-tête, tel que `ETag` ou `Location` , et `Value` fournit la valeur définie par le serveur pour cet en-tête. La ligne se termine par un CRLF.

Une réponse avec des en-têtes peut ressembler à ceci:

```
HTTP/1.1 201 Created \r\n
Server: WEBrick/1.3.1 \r\n
Location: http://example.com/files/129742 \r\n
```

Corps de message

Comme avec les corps de requête , les réponses HTTP peuvent contenir un corps de message. Cela fournit des données supplémentaires que le client traitera. Notamment, 200 réponses OK à une requête GET bien formée doivent toujours fournir un corps de message contenant les données demandées. (S'il n'y en a pas, 204 No Content est une réponse plus appropriée).

Un corps de message est inclus après tous les en-têtes et un double CRLF. Comme pour les requêtes, sa longueur en octets doit être indiquée avec l'en `Content-Length` tête `Content-Length` . Une réponse réussie à une demande GET peut donc ressembler à ceci:

```
HTTP/1.1 200 OK\r\n
Server: WEBrick/1.3.1\r\n
Content-Length: 39\r\n
ETag: 4f7e2ed02b836f60716a7a3227e2b5bda7ee12c53be282a5459d7851c2b4fdfd\r\n
\r\n
Nobody expects the Spanish Inquisition.
```

Lire Réponses HTTP en ligne: https://riptutorial.com/fr/http/topic/3077/reponses-http

Chapitre 9: Requêtes HTTP

Paramètres

Méthode HTTP	Objectif
`OPTIONS`	Récupérez des informations sur les options de communication (méthodes et en-têtes disponibles) disponibles sur l'URI de requête spécifié.
`GET`	Récupérer les données identifiées par l'URI de la demande ou les données produites par le script disponible à l'URI de la demande.
`HEAD`	Identique à `GET` sauf qu'aucun corps de message ne sera renvoyé par le serveur: uniquement les en-têtes.
`POST`	Soumettez un bloc de données (spécifié dans le corps du message) au serveur pour l'ajouter à la ressource spécifiée dans l'URI de la demande. Le plus couramment utilisé pour le traitement de formulaire.
`PUT`	Stockez les informations jointes (dans le corps du message) en tant que ressource nouvelle ou mise à jour sous l'URI de la requête donnée.
`DELETE`	Supprimer ou mettre en file d'attente pour suppression la ressource identifiée par l'URI de la demande.
`TRACE`	Essentiellement, une commande echo: un serveur HTTP compatible et fonctionnel doit renvoyer l'intégralité de la requête en tant que corps d'une réponse 200 (OK).

Remarques

La méthode `CONNECT` est réservée par la spécification des définitions de méthode pour une utilisation avec des proxys capables de basculer entre les modes de proxy et de tunneling (comme pour le tunneling SSL).

Examples

Envoi manuel d'une requête HTTP minimale à l'aide de Telnet

Cet exemple montre que HTTP est un protocole de communication Internet basé sur du texte et affiche une requête HTTP de base et la réponse HTTP correspondante.

Vous pouvez utiliser Telnet pour envoyer manuellement une requête HTTP minimale depuis la ligne de commande, comme suit.

1. Démarrez une session Telnet sur le serveur Web `www.example.org` sur le port 80:

   ```
   telnet www.example.org 80
   ```

   Telnet rapporte que vous avez connecté au serveur:

   ```
   Connected to www.example.org.
   Escape character is '^]'.
   ```

2. Entrez une ligne de demande pour envoyer un chemin d'URL de requête GET `/` , en utilisant HTTP 1.1

   ```
   GET / HTTP/1.1
   ```

3. Entrez une ligne de champ d'en-tête HTTP pour identifier la partie du nom d'hôte de l'URL requise, requise dans HTTP 1.1.

   ```
   Host: www.example.org
   ```

4. Entrez une ligne vide pour compléter la demande.

   Le serveur Web envoie la réponse HTTP, qui apparaît dans la session Telnet.

La session complète est la suivante. La première ligne de la réponse est la *ligne d'état HTTP* , qui inclut le code d'état 200 et le texte d'état *OK* , qui indiquent que la demande a été traitée avec succès. Ceci est suivi par un certain nombre de champs d'en-tête HTTP, une ligne vide et la réponse HTML.

```
$ telnet www.example.org 80
Trying 2606:2800:220:1:248:1893:25c8:1946...
Connected to www.example.org.
Escape character is '^]'.
GET / HTTP/1.1
Host: www.example.org

HTTP/1.1 200 OK
Accept-Ranges: bytes
Cache-Control: max-age=604800
Content-Type: text/html
Date: Thu, 21 Jul 2016 15:56:05 GMT
Etag: "359670651"
Expires: Thu, 28 Jul 2016 15:56:05 GMT
Last-Modified: Fri, 09 Aug 2013 23:54:35 GMT
Server: ECS (lga/1318)
Vary: Accept-Encoding
X-Cache: HIT
x-ec-custom-error: 1
Content-Length: 1270

<!doctype html>
<html>
<head>
    <title>Example Domain</title>
```

```
    <meta charset="utf-8" />
    <meta http-equiv="Content-type" content="text/html; charset=utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
</head>
<body>
<div>
    <h1>Example Domain</h1>
    <p>This domain is established to be used for illustrative examples in documents. You may
use this
    domain in examples without prior coordination or asking for permission.</p>
    <p><a href="http://www.iana.org/domains/example">More information...</a></p>
</div>
</body>
</html>
```

(élément de `style` et lignes vierges supprimées de la réponse HTML, par souci de concision.)

Format de demande de base

Dans HTTP 1.1, une requête HTTP minimale se compose d'une ligne de demande et d'un en-tête d' `Host` :

```
GET /search HTTP/1.1 \r\n
Host: google.com \r\n
\r\n
```

La première ligne a ce format:

```
Method Request-URI HTTP-Version CRLF
```

`Method` doit être une méthode HTTP valide. l'un de [1] [2] :

- `OPTIONS`
- `GET`
- `HEAD`
- `POST`
- `PUT`
- `DELETE`
- `PATCH`
- `TRACE`
- `CONNECT`

`Request-URI` indique soit l'URI, soit le chemin d'accès à la ressource demandée par le client. Cela peut être soit:

- un URI complet, comprenant schéma, hôte, port (facultatif) et chemin d'accès; ou
- un chemin, auquel cas l'hôte doit être spécifié dans l'en-tête de l' `Host`

`HTTP-Version` indique la version du protocole HTTP que le client utilise. Pour les requêtes HTTP 1.1, cela doit toujours être `HTTP/1.1` .

La ligne de demande se termine par un retour chariot - un saut de ligne, généralement représenté par `\r\n` .

Champs d'en-tête de demande

Les champs d'en-tête (généralement appelés simplement «en-têtes») peuvent être ajoutés à une requête HTTP pour fournir des informations supplémentaires à la demande. Un en-tête a une sémantique similaire aux paramètres transmis à une méthode dans tout langage de programmation prenant en charge de telles choses.

Une requête avec des en `User-Agent` têtes `Host` , `User-Agent` et `Referer` peut ressembler à ceci:

```
GET /search HTTP/1.1 \r\n
Host: google.com \r\n
User-Agent: Chrome/54.0.2803.1 \r\n
Referer: http://google.com/ \r\n
\r\n
```

Une liste complète des en-têtes de requête HTTP 1.1 pris en charge peut être trouvée dans la spécification . Les plus courants sont:

- `Host` - la partie nom d'hôte de l'URL de la demande (obligatoire dans HTTP / 1.1)
- `User-Agent` - une chaîne qui représente l'agent utilisateur demandant;
- `Referer` - l'URI dont le client a été renvoyé ici; et
- `If-Modified-Since` - donne une date que le serveur peut utiliser pour déterminer si une ressource a changé et indique que le client peut utiliser une copie en cache si ce n'est pas le cas.

Un en-tête doit être formé comme `Name: Value CRLF` . `Name` est le nom de l'en-tête, tel que `User-Agent` . `Value` correspond aux données qui lui sont affectées et la ligne doit se terminer par un CRLF. Les noms d'en-tête sont insensibles à la casse et ne peuvent utiliser que des lettres, des chiffres et des caractères `!#$%&'*+-.^_`|~` (RFC7230 section 3.2.6 Composants de valeur de champ).

Le nom du champ d'en-tête `Referer` est une faute de frappe pour «référent», introduit accidentellement dans RFC1945 .

Corps de message

Certaines requêtes HTTP peuvent contenir un corps de message. Ce sont des données supplémentaires que le serveur utilisera pour traiter la demande. Les corps de message sont le plus souvent utilisés dans les requêtes POST ou PATCH et PUT, pour fournir de nouvelles données que le serveur doit appliquer à une ressource.

Les demandes qui incluent un corps de message doivent toujours inclure sa longueur en octets avec l'en `Content-Length` tête `Content-Length` .

Un corps de message est inclus *après* tous les en-têtes et un double CRLF. Un exemple de requête PUT avec un corps peut ressembler à ceci:

```
PUT /files/129742 HTTP/1.1\r\n
Host: example.com\r\n
User-Agent: Chrome/54.0.2803.1\r\n
```

```
Content-Length: 202\r\n
\r\n
This is a message body. All content in this message body should be stored under the
/files/129742 path, as specified by the PUT specification. The message body does
not have to be terminated with CRLF.
```

`HEAD` requêtes `HEAD` et `TRACE` ne doivent pas inclure de corps de message.

Lire Requêtes HTTP en ligne: https://riptutorial.com/fr/http/topic/2909/requetes-http

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec HTTP	Community, DaSourcerer, Kornel, Peter Hilton
2	Authentification	DaSourcerer, Peter Hilton, Stefan Kögl
3	Codes d'état HTTP	ArtOfCode, DaSourcerer, Deltik, Kornel, Lex Li, mnoronha, Peter Hilton, Rptk99, Sender, Xevaquor
4	Encodage des réponses et compression	Jeff Bencteux, Peter Hilton
5	HTTP pour les API	ArtOfCode, mnoronha, Peter Hilton, Roman Vottner
6	Mise en cache des réponses HTTP	DaSourcerer, Kornel
7	Origine croisée et contrôle d'accès	ArtOfCode
8	Réponses HTTP	ArtOfCode, Jeff Bencteux, Peter Hilton
9	Requêtes HTTP	artem, ArtOfCode, Jeff Bencteux, Peter Hilton