Les requêtes HTTP

Temps de lecture : 9 minutes

Rappels sur le protocole Http

Comme nous l'avons vu ce protocole est basé sur un modèle **client / serveur**.

Cela signifie que des requêtes sont effectuées par un client à un serveur et qu'il s'attend à recevoir des réponses pour chaque requête.

Le client, appelé **agent utilisateur**, est toujours celui qui initie la requête `Http`.

Lorsque votre navigateur visite une URL il va envoyer des requêtes successives pour récupérer le HTML, le CSS et le JS de votre première page.

Étapes d'un échange `Http`

Etape 1 - Le client commence par établir une ou plusieurs connexions de la couche transport, avec le protocole `TCP` pour `HTTP`.

Le nombre de connexion `TCP` ouvertes dépendra de votre navigateur et de la version HTTP utilisée.

En HTTP 1.1, par exemple, Chrome va commencer par en ouvrir 3 et pourra en ouvrir jusqu'à 10 si le nombre de requêtes HTTP en fil d'attente le nécessite afin de paralléliser les requêtes. A noter **qu'une seule requête peut être en cours (`flying`)** donc ce n'est pas du vrai parallélisme et cela n'améliore pas énormément les performances.

En HTTP 2, une seule connexion `TCP` est ouverte mais il n'y a pas de limites sur le nombre de requêtes en cours, c'est ce qu'on appelle le `multiplexing`. Plusieurs requêtes transitent **en même temps** ce qui augmente les performances.

Une fois le ou les connexions établies (avec le `three-way handshake` que nous avons vu).

Etape 2 - Le client envoi un message Http. Si c'est en version 1 il est lisible, en version 2 c'est en binaire donc totalement illisible mais les principes sont les mêmes.

Voici un exemple de message HTTP de type requête :

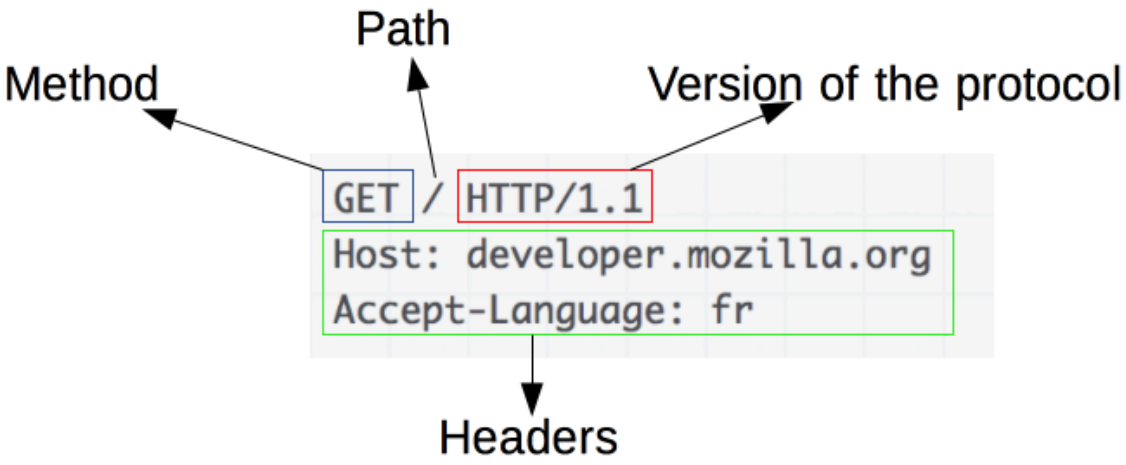
```
GET / HTTP/1.1
Host: dyma.fr
Accept-Language: fr
```

Etape 3 - le serveur gère la requête et renvoi une réponse.

```
HTTP/1.1 200 OK
Cache-Control: no-cache
Server: libnhttpd
Date: Wed Jul 4 15:32:03 2018
Connection: Keep-Alive:
Content-Type: application/json
Content-Length: 24860
```

Qu'est-ce qu'une requête `Http` ?

Une requête HTTP se décompose comme suit :



La ligne requête (`request line`)

La première ligne est appelée `request line` elle comporte trois parties.

Premièrement, **la méthode** `Http` : `OPTIONS`, `GET`, `HEAD`, `POST`, `PUT`, `PATCH`, `DELETE`, `TRACE` ou `CONNECT`. Elle exprime l'intention de la requête.

Nous les reverrons en détails, mais par exemple `GET` signifie donne moi la ressource à l'URI indiqué.

Deuxièmement, `Request-URI` : est l'identifiant de la ressource demandée (`Uniform Resource Identifier`). Par exemple `/` ou `/app/hello.html` ou `http://dyma.fr/hello.html` sont des URI.

Troisièmement, la version du protocole : 1.1 ou 2.

Les `Headers`

Il existe de nombreux `headers Http`.

Leur forme est toujours constituée du nom du `header` suivi d'un deux-points puis sa valeur.

Nous allons prendre quelques exemples, mais nous ne les traiterons pas tous car il y en a plus de 50 et que vous n'aurez pas souvent à modifier ceux par défaut.

Exemple de `Authorization`. Cet `header` nous sera utile lorsque nous verrons comment gérer l'authentification `JWT` :

```
Authorization: Bearer mF_9.B5f-4.1JqM
```

Aussi simple que cela ! Le parser `Http` va récupérer ce `header` et les bindings de `Node.js` le mettront ensuite dans un objet JS auquel il sera simple d'accéder comme nous l'étudierons.

Un autre exemple sont les `cookies` :

```
Cookie: name1=value1;name2=value2;name3=value3
```

Le plus souvent nous ne définirons pas de cookies nous-mêmes mais nous utiliserons des librairies. Nous verrons comment lorsque nous étudierons l'authentification basée sur un système de `sessions`.

D'une manière plus générale ces `headers` de requête permettent de contrôler principalement :

- la **mise en cache**
- **l'authentification**,
- **la connexion** `TCP` (juste pour dire si elle doit être `keep-alive` ou non dans la version 1.1),

- **le contenu** (quelles méthodes de compression ou d'encodage et quelles langues sont acceptées par le client),
- les **cookies**, les `CORS` (pour Cross-origin resource sharing, nous y reviendrons plus tard dans la formation),
- **le contexte de requête** (comme par exemple l'`host` (domaine du serveur sur lequel est effectuée la requête), le `user-agent` (type de navigateur etc),

Le corps

Pour les requêtes dont la méthode est par exemple `POST`, elles transmettent des données, par exemple le contenu d'un formulaire.

Elles ont donc un corps appelé `body`.

L'objet `IncomingMessage` de `Node.js`

`Node.js` vous fournit un objet spécial à chaque requête passé en premier argument de votre fonction `listener`.

Il s'agit d'une instance de la classe `http.IncomingMessage`.

Cette classe possède des événements, des méthodes et des propriétés utilent pour lire les `headers`, les données etc.

Nous allons voir les plus importantes.

La propriété `headers`

Cette propriété permet d'accéder aux `headers` qui ont été parsés et mis dans un objet `Javascript`.

Nous pouvons donc faire par exemple :

```
console.log(request.headers);
// { 'user-agent': 'curl/7.22.0',
//   host: '127.0.0.1:8000',
//   accept: '*/*' }
```

La propriété `method`

Permet d'accéder à la méthode de la requête `Http` :

```
console.log(request.method);
// 'GET'
```

La propriété `url`

Permet d'accéder à l'URI de la première ligne de la requête que nous avons vu :

```
GET /status?name=ryan HTTP/1.1

Accept: text/plain
```

Pour cette requête nous aurions :

```
console.log(request.url);
// '/status?name=ryan'
```

Vous pouvez utiliser le module `core` de `Node.js` appelé `url` pour parser l'URL et obtenir un objet Javascript avec toutes les informations.

Ainsi dans l'exemple pour auriez :

```
require('url').parse('/status?name=ryan')
Url {
  protocol: null,
  slashes: null,
  auth: null,
  host: null,
  port: null,
  hostname: null,
  hash: null,
  search: '?name=ryan',
  auth: 'name=ryan',
  pathname: '/status',
  path: '/status?name=ryan',
  href: '/status?name=ryan' }
```

Nous n'avons pas besoin du reste pour le moment.

L'objet `IncomingMessage` est un `Stream`

L'objet `IncomingMessage` est en fait un `stream` qui sont eux mêmes des `EventEmitter`s.

Nous avons vu comment sont transférés les paquets en utilisant `TCP`, ils arrivent en continu et sont donc comme un flux de données.

Il est donc logique qu'une requête soit traité comme un flux !

Les `Streams`

Pour aller un peu plus loin, nous allons voir ce que sont les `streams` de `Node.js`.

Un `stream` est une interface abstraite fournie par `Node.js` permettant de gérer les flux de données.

Nous pouvons d'abord voir que beaucoup d'objets dans `Node.js` sont des `streams` :

Nous retrouvons par exemple notre module `fs`.

La méthode `fs.createReadStream`

Rappelez-vous de la méthode `fs.readFile()` : nous chargions tout le fichier dans un `Buffer` avant de pouvoir agir dessus dans la fonction de `callback`.

En outre, si le fichier faisait 1 Go, nous avions vu que la lecture avec cette méthode prendrait 1 Go de RAM pour mettre en `Buffer` tout le fichier.

Heureusement, avec les `streams`, vous pouvez **agir sur le flux sans avoir à tout mettre dans un `Buffer` en mémoire** !

Prenons un exemple très parlant. Vous avez un fichier de 200 Mo que vous voulez transférer à votre utilisateur quand il veut le télécharger.

Vous ne voulez pas augmenter la consommation de RAM de 200 Mo pour chaque demande de téléchargement.

C'est un cas parfait pour l'utilisation d'un `stream` :

```
const fs = require('fs');
const http = require('http');

http.createServer((req, res) => {
  const src = fs.createReadStream('./gros-rapport.pdf');
  src.pipe(res);
});
```

Impressionné ? En deux ligne, nous avons créé un flux de lecture à partir du fichier de 200 Mo qui va automatiquement créer un petit `Buffer` et transférer le flux dans la réponse (nous verrons que l'objet réponse est aussi un `stream`).

La méthode `pipe()` permet juste de prendre un flux en lecture (qui est un `readable stream` de la classe `stream.Readable`) et de le brancher à un flux d'écriture (un `writableStream` de la classe `stream.Writable`).

Retour sur `IncomingMessage`

Notre objet requête est un `readable stream`. Nous prenons le flux de paquets et les lisons au fur et à mesure.

Cela est particulièrement utile quand nous recevons des données avec une requête `POST`.

Prenons un exemple et expliquons le pas à pas :

```
const http = require('http');

http.createServer((request, response) => {
```

```
    const { headers, method, url } = request;
    let body = [];
    request.on('error', (err) => {
      console.error(err);
    }).on('data', (chunk) => {
      body.push(chunk);
    }).on('end', () => {
      body = Buffer.concat(body).toString();
    });
  }).listen(8080);
```

Nous créons notre serveur `Http` qui va écouter le port `8080`.

Nous utilisons l'affectation par décomposition d'ES6 afin d'extraire de l'objet `request` les propriétés `headers`, `method` et `url`.

```
  const { headers, method, url } = request;
```

Ensuite nous créons un tableau vide et nous créons un `listener` pour l'événement `data`. Mais oui, rappelez-vous un `stream` est un type particulier d'`EventEmitter`, il a donc bien une propriété `on()` qui permet d'utiliser un `listener`.

`error`, `data` et `end` sont des événements de la classe `stram.Readable` dont notre `IncomingMessage` hérite.

`error` n'a pas besoin d'explication particulière, si nous recevons un événement de type `error`, nous nous contentons d'utiliser la console.

Il faut seulement rappeler que si une erreur n'est pas gérée par un `listener` elle va être `throw` et votre processus `Node.js` va crasher.

`data` est un type d'événement intéressant, il représente un `Buffer` qui a été créé et qui es prêt à être utilisé.

Représentez-vous le processus de cette manière : vous avez un flux de données qui rempli un `Buffer`, une fois suffisamment rempli, un événement `data` est émis.

Vous recevez le `Buffer` appelé `chunk` qui est juste un ensemble de `bits` en mémoire.

Une fois qu'il vous le passe dans la fonction `listener`, le `Buffer` est vidé car vous avez la responsabilité du morceau qui vous ai remis, et il se reremplit pendant ce temps.

Cette architecture permet de garder une utilisation mémoire énormément plus faible.

`end` est l'événement émis lorsqu'il ne reste plus de données à consommer.

Enfin, nous concaténons nos `Buffer` qui sont dans notre tableau et nous convertissons notre `Buffer` en `string` :

```
Buffer.concat(body).toString()
```

Nous verrons dans la formation qu'il existe des librairies qui vont parser notre `body` et que nous n'aurons pas le faire nous même.

Le code ci-dessus ne fonctionnera pas car nous ne répondons rien pour le moment au client ! C'est l'objet de la leçon suivante.