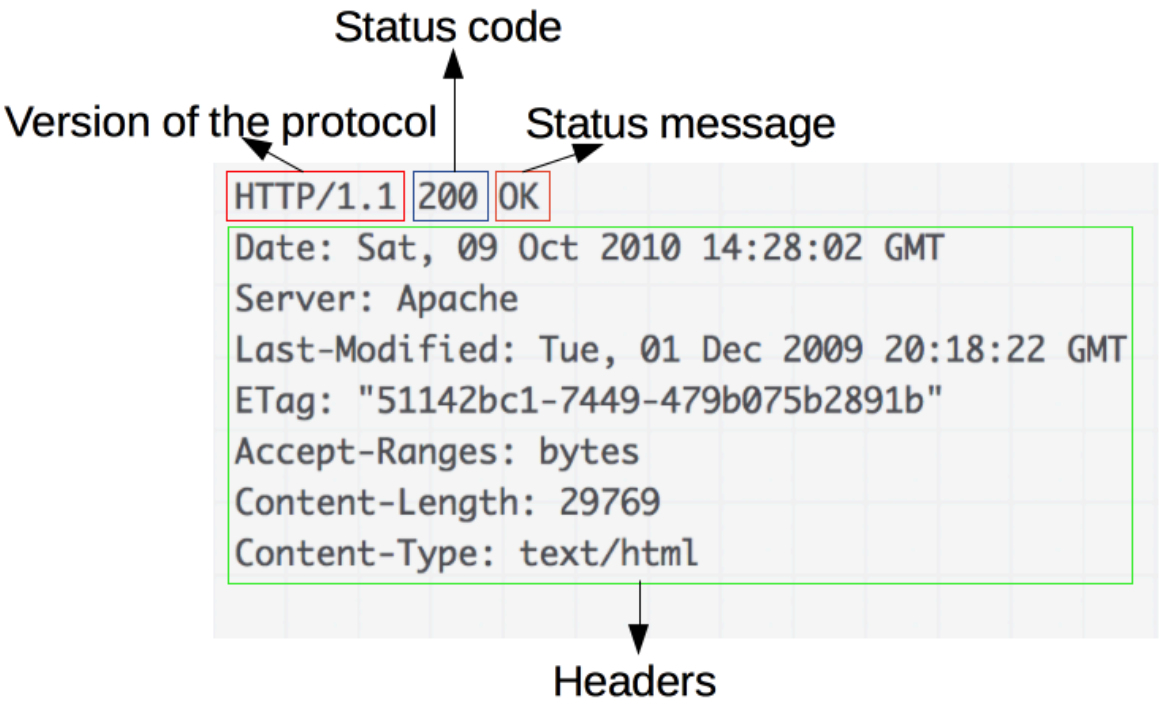
Les réponses HTTP

Temps de lecture : 7 minutes

La réponse `Http`

Le serveur après avoir traité la requête et effectué les opérations souhaitées doit renvoyer une réponse au client.

Un exemple de réponse :



La ligne réponse

Elle comporte la version du protocole utilisé puis le code du statut et le message correspondant.

Les `status code` et `status message`

Les codes de `status` permettent d'indiquer si une requête HTTP a été exécutée avec succès ou non. Un message court l'accompagne.

Les codes commençant par `2` signifie que **la requête a fonctionné**.

Les codes commençant par `3` sont utilisés pour **les redirections**.

Les codes commençant par `4` sont utilisés pour **les erreurs côté client**.

Les codes commençant par `5` sont utilisés pour **les erreurs côté serveur**.

Nous verrons ces codes au fur et à mesure mais nous allons voir maintenant les principaux :

- `200 OK` : la requête a fonctionnée et tout s'est bien déroulé.
- `201 Created` : la ressource a été créée (par exemple après un `PUT`.
- `400 Bad Request` : le serveur n'a pas pu comprendre la requête à cause d'une mauvaise syntaxe.
- `401 Unauthorized` : une authentification est nécessaire pour obtenir la réponse demandée.
- `403 Forbidden` : le client n'a pas les droits d'accès au contenu (il est authentifié mais n'a pas les bons droits).
- `404 Not Found` : la ressource n'a pas été trouvée par le serveur.
- `500 Internal Server Error` : le serveur a rencontré une situation qu'il ne sait pas traiter.

Il existe plus d'une cinquantaine de code mais la plupart ne sont pas utilisés couramment.

Les `headers`

Comme pour la requête il existe des `headers` spécifiques à la réponse `Http`.

Voici un exemple de `headers` de réponse, aucun ne sont obligatoires :

```
Access-Control-Allow-Origin: *
Connection: Keep-Alive
Content-Encoding: gzip
Content-Type: text/html; charset=utf-8
Date: Mon, 18 Jul 2016 16:06:00 GMT
Etag: "c561c68d0ba92bbeb8b0f612a9199f722e3a621a"
Last-Modified: Mon, 18 Jul 2016 02:36:04 GMT
Server: Apache
Set-Cookie: mykey=myvalue; Max-Age=31449600; Path=/; secure
Transfer-Encoding: chunked
```

`Access-Control-Allow-Origin: *` : nous développerons ce sujet plus tard, mais cela est relié aux requêtes `CORS` (Cross-origin resource sharing). Le serveur peut contrôler l'accès

d'un agent utilisateur à des ressources. Le serveur indique par `*` que toute origine peut accéder à ses ressources.

`Connection: Keep-Alive` : permet de maintenir la connexion `TCP` ouverte après la requête. N'est utilisé qu'en `HTTP/1.1`, `HTTP/2` maintient la connexion sans avoir à spécifier de `header`.

`Content-Encoding: gzip` : permet d'indiquer à l'agent utilisateur que le `body` a été compressé en utilisant `gzip` pour qu'il puisse le décrypter. Les échanges `Http` sont majoritairement compressés pour réduire les latences sur le réseau.

`Content-Type: text/html; charset=utf-8` : permet d'indiquer à l'agent utilisateur le `media-type` et l'`encoding` de la ressource renvoyée.

Le `media-type` utilise le standard `MIME` (Multipurpose Internet Mail Extensions) permettant d'indiquer la nature et le format d'un document. Il y a cinq grand types (`text`, `image`, `video`, `audio` et `application`) et beaucoup de sous-types (par exemple, `text/css`, `image/png`, `application/json`, `application/pdf`, `video/mp4`). `application` signifie que ce sont n'importe quel type de données binaires.

Le `ETag` : est un identifiant unique pour une version spécifique d'une ressource, il est géré par une librairie le plus souvent et permet une mise en cache optimal des ressources pour n'envoyer la ressource que si elle a changé depuis la dernière visite.

`Date` spécifie juste le moment de la réponse et `LasModified` est la date de la dernière modification de la ressource.

`Set-Cookie: mykey=myvalue; Max-Age=31449600; Path=/; secure` : le serveur demande à l'agent utilisateur de sauvegarder ce cookie (par exemple dans le navigateur si c'est l'agent utilisateur). Il demande de sauvegarder une paire clé / valeur. Il spécifie qu'elle sera valide pour `Max-Age` de `31449600` millisecondes et qu'ensuite le navigateur doit considérer ce `cookie` comme expiré. Il spécifie `secure`, c'est-à-dire que ce `cookie` ne pourra être envoyé au serveur que par protocole sécurisé `TLS` ou `SSL` pour `HTTPS` par exemple. Il indique `Path=/` signifiant que le chemin de la requête doit contenir `/` pour que le `cookie` soit envoyé.

L'objet `Node.js` : `http.ServerResponse`

Les réponses envoyées par votre serveur web `Node.js` seront des instances de la classe `http.ServerResponse`.

`Node.js` vous fournit cet objet en deuxième argument de votre fonction `listener` :

```
const server = http.createServer();
server.on('request', (request, response) => {
  // la même chose qu'avant
});
```

Il est passé en deuxième argument de l'événement `request`.

Il s'agit également d'un `stream` et donc d'un `EventEmitter`.

Il s'agit d'un `stream writable`, il est possible d'écrire dedans mais pas de le lire, c'est logique puisque nous envoyons un flux depuis un serveur vers un agent utilisateur.

Il possède deux événements, mais surtout un grand nombres de méthodes et de propriétés. Nous allons voir les principales.

La méthode `setHeader()`

Cette méthode permet de définir les `headers` de la réponse du serveur.

Vous pouvez par exemple définir le `Content-Type` qui est important pour que l'agent utilisateur (le plus souvent le navigateur) puisse savoir comment gérer le contenu et l'afficher :

```
response.setHeader('Content-Type', 'text/html');
```

Vous pouvez passer plusieurs valeurs dans certains `headers` comme les `cookies` :

```
response.setHeader('Set-Cookie', ['key1=val1', 'key2=val2']);
```

Le propriétés `statusCode` et `statusMessage`

Vous pouvez définir le code et le message du statut de la réponse comme ceci :

```
response.statusCode = 404;
response.statusMessage = 'Not found';
```

La méthode `writeHead()`

Contrairement aux propriétés précédentes et à `setHeader()` vous écrivez directement les `headers` dans le `writable stream` de la réponse.

Vous les envoyez donc dans le flux, alors qu'avant `Node.js` le fera pour vous avant d'envoyer le `body`.

Définir les `headers` comme précédemment s'appelle une définition **implicite des** `headers` alors qu'utiliser `writeHead()` permet de les écrire **explicitement** dans le flux.

Voici un exemple :

```
response
  .writeHead(200, {
    'Content-Length': Buffer.byteLength(body),
    'Content-Type': 'text/plain'
  })
```

La méthode `write()`

Pour écrire des données sur le flux qui seront dans le `body` de la réponse, il suffit d'utiliser `write()` autant de fois que vous voulez.

Sa signature est :

```
response.write(chunk[, encoding][, callback])
```

Une `chunk` est un morceau de données qui peut être de type `Buffer` ou `string`.

Par défaut l'`encoding` est défini à `utf-8`.

Par exemple vous pouvez écrire :

```
response.write('<html>');
response.write('<body>');
response.write('<h1>Hello, World!</h1>');
response.write('</body>');
response.write('</html>');
```

La méthode `end()`

Cette méthode signale au serveur `Node.js` que tous les `headers` de la réponse et que le `body` ont été envoyées.

Le serveur doit donc considérer que le message `Http` est complet. Il est **obligatoire d'utiliser** `res.end()` pour chaque réponse.

La signature de la méthode est :

```
response.end([data][, encoding][, callback])
```

Par défaut l'`encoding` est `utf-8`.

Vous pouvez envoyer des données dans `body` en passant un objet `data`.

Il s'agit d'un raccourci syntaxique équivalent à :

```
response.write(data, encoding);
response.end(callback);
```

En reprenant l'exemple précédent nous pouvons donc écrire :

```
const body = '<html><body><h1>Hello, World!</h1></body></html>';
response
  .writeHead(200, {
    'Content-Length': Buffer.byteLength(body),
    'Content-Type': 'text/plain'
  })
  .end(body);
```

Exemple de petit serveur

Nous allons créer un serveur qui va renvoyer ce qu'il reçoit.

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

http.createServer((req, res) => {
  if (req.method === 'POST' && req.url === '/echo') {
    let body = [];
    req.on('data', (chunk) => {
      body.push(chunk);
```

```
    }).on('end', () => {
      body = Buffer.concat(body).toString();
      res.end(body);
    });
  } else {
    res.statusCode = 404;
    res.end();
  }
}).listen(8080);
```

Nous créons un serveur avec `createServer()` et lui passons notre `listener`.

Nous aurions également pu faire `server.on('request, (req, res) => …` mais nous préférons la notations courte.

Nous contrôlons la méthode utilisée pour la requête et l'URL demandée.

Si la méthode est `POST` et si l'url est `'/echo'` alors nous allons écrire ce que nous recevons du `readable stream`, c'est-à-dire de la requête, dans un tableau.

Ensuite, lorsque nous recevons l'événement `end` et que nous avons donc tout reçu de la requête, nous allons renvoyer le tout après l'avoir concaténé et converti en `string` en utilisant le raccourci `res.end(data)`.

Si la méthode n'est pas `POST` ou l'url n'est pas `/echo`, nous renvoyons un `statusCode` de `404` pour `NOT FOUND`, nous n'oublions pas de terminer la réponse avec `res.end()`.

Nous pouvons dans ce cas faire mieux : rappelez vous la méthode `pipe()` disponible sur les `streams readables` :

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

http.createServer((req, res) => {
  if (req.method === 'POST' && req.url === '/echo') {
    req.pipe(res);
  } else {
    res.statusCode = 404;
    res.end();
  }
}).listen(8080);
```

Nous écrivons dans le flux de sorti à mesure que nous recevons dans le flux d'entrée ! Magique non ?

Plus qu'à rajouter la gestion des erreurs éventuelles, sinon rappelez vous que si les erreurs ne sont pas gérées dans `Node.js` et qu'elles surviennent, votre application crashera :

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

http.createServer((req, res) => {
  req.on('error', err => {
    console.error(err);
    res.statusCode = 400; // BAD REQUEST
    res.end();
  });
  res.on('error', err => {
    console.error(err);
  });
  if (req.method === 'POST' && req.url === '/echo') {
    req.pipe(res);
  } else {
    res.statusCode = 404;
    res.end();
  }
}).listen(8080);
```