7 minutes

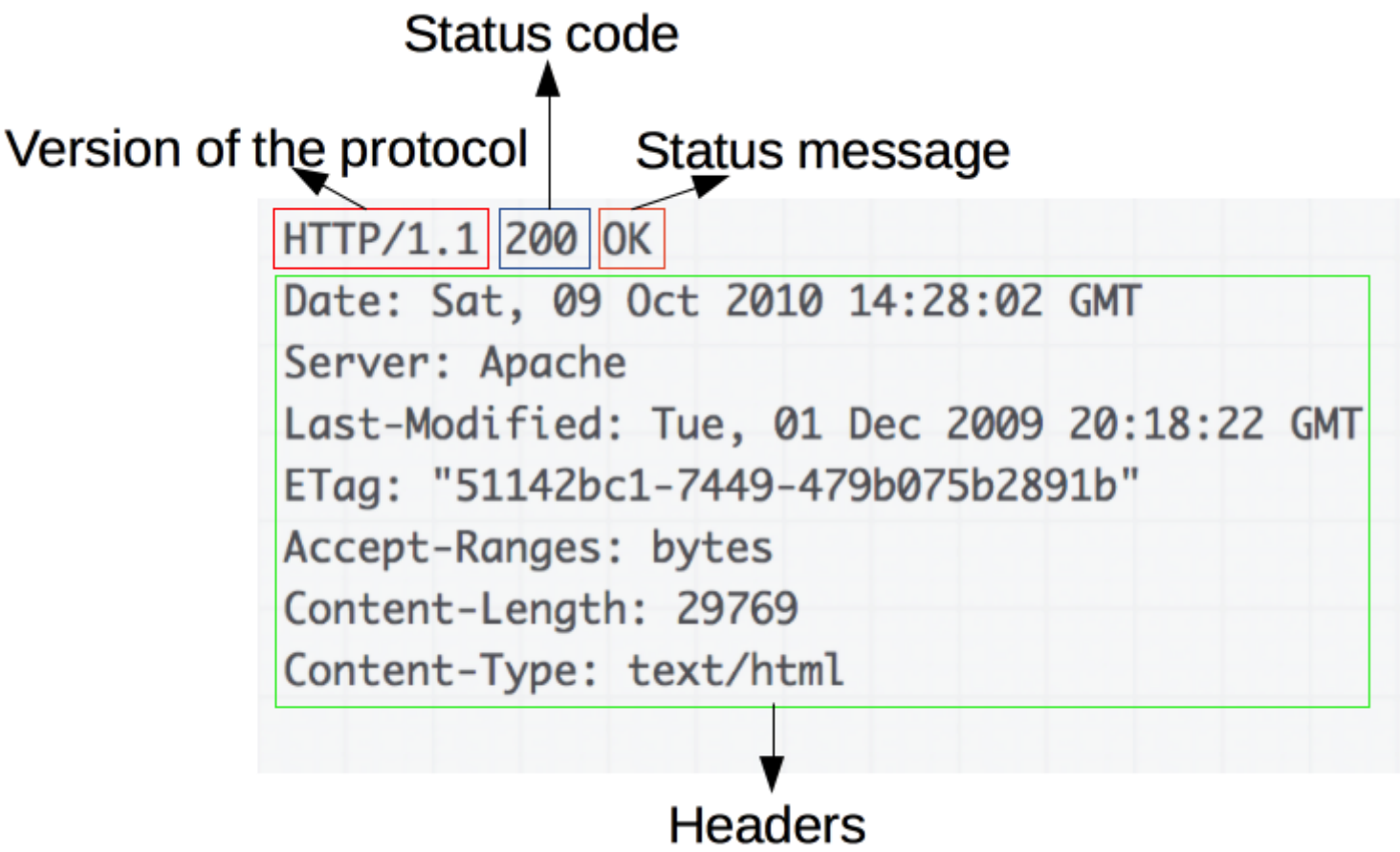
Première requête HTTP avec fetch

Avant d'envoyer notre première requête, nous avons besoin d'étudier ce qu'un serveur va nous retourner : une réponse `HTTP`.

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` secondes 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.

La `Web API fetch`

`fetch` est une `Web API` disponible dans tous les navigateurs qui permet d'envoyer des requêtes `HTTP`.

Syntaxe

La syntaxe de l'`API` est très simple :

```
const promesse = fetch(url, [options]);
```

Le premier paramètre est l'`URL` cible de la requête.

Le second paramètre est un objet d'options que nous étudierons en détails.

La `Web API` retourne une promesse qui sera tenue si le serveur répond.

La promesse est résolue avec un objet `Response`.

A ce stade, vous pouvez accéder aux propriétés suivantes :

`url` : url de la requête.

`redirected` : booléen indiquant si la requête a été redirigée par le serveur.

`status` : code du statut de la requête.

`ok` : booléen pour savoir si la requête s'est bien déroulée (`true` si le code du statut est compris entre 200 et 299).

`type` : type de la requête `cors` ou `basic` (nous y reviendrons).

`statusText` : message du statut de la requête.

`headers` : headers de la réponse. Il faut utiliser la méthode `get()` et passer le nom du `header` à récupérer.

Envoyer une requête `GET`

Sans passer d'option, `fetch` va effectuer une requête `HTTP` avec la méthode `GET` :

```
const reponse = await fetch("https://jsonplaceholder.typicode.com/users");
```

Vous pouvez à ce stade accéder aux propriétés que nous avons vues :

```
console.log(reponse.ok);
console.log(reponse.status);
console.log(reponse.statusText);
console.log(reponse.redirected);
console.log(reponse.type);
console.log(reponse.url);
for (const [cle, valeur] of reponse.headers) {
  console.log(`${cle} : ${valeur}`);
}
```

Parser la réponse

Pour lire le contenu de la réponse, il faut la parser, c'est-à-dire attendre que tout le `body` soit reçu et le lire dans un format particulier.

Toutes les méthodes de parsing retournent elle-même une promesse :

`text()` permet de lire la réponse et de la parser au format `text`.

`json()` permet de lire la réponse et de la parser au format `json`.

`formData()` permet de lire la réponse et de la parser au format `formData` (que nous verrons).

`blob()` permet de lire la réponse et de la parser au format `blob` (qui est un objet de données binaires avec un type).

`blobarrayBuffer` permet de lire la réponse et de la parser au format `ArrayBuffer` (représentation bas niveau des données binaires).

Dans la plupart des cas vous utilisez la méthode `json()` qui vous permettra de lire le `body` au format `JSON` :

```
const reponse = await fetch("https://jsonplaceholder.typicode.com/users");
const donnees = await reponse.json();
```

Code exécutable

vous pouvez effectuer vos tests :
https://codesandbox.io/embed/js-c15-l2-1-43hxb