3IN017 - TECHNOLOGIES DU WEB

Protocole HTTP - Serveur avec Express

18 février 2025

Gilles Chagnon

Plan

Cours précédents côté client (HTML, CSS, JavaScript, React)

Ce cours : côté serveur

1. HTTP – Généralités
2. HTTP – Détails d'une requête
3. Serveur – Node.js
4. Serveur – Express, requêtes
5. Serveur – Express, routage
6. Serveur – Express, middlewares

HTTP – Généralités

Protocole HTTP (niveau 5-7 OSI)

SCIENCES SORBONNE UNIVERSITÉ

- Définit comment échanger de l'information
- Inventé par Tim Berner-Lee en 1990 (HTTP/0.9)
- Version HTTP/1.0 en 1996 (RFC 1945)

Côté client

- Navigateur
- Robots (d'indexation)
- Stockage dans le nuage : WebDAV, CardDAV
- Applications diverses

Côté serveur

On peut distinguer deux niveaux

- Applications (Apache, Nginx) avec des extensions (ex. PHP) : **Routage via le système de fichiers**
- Bibliothèques (Python HTTP, Node.JS/Express, Tomcat, *etc.*) : **Routage logiciel**

Protocole HTTP – Exemple

SCIENCES SORBONNE UNIVERSITÉ

Utilisation de HTTP/1.0 (texte)

```
[gilles@Herbert ~]$ telnet 192.168.1.21 3000
Trying 192.168.1.21...
Connected to 192.168.1.21.
Escape character is '^]'.
GET / HTTP/1.0

HTTP/1.1 200 OK
X-Powered-By: Express
Access-Control-Allow-Origin: *
Access-Control-Allow-Methods: *
Access-Control-Allow-Headers: *
Content-Type: text/html; charset=utf-8
Accept-Ranges: bytes
Content-Length: 1711
ETag: W/"6af-+M4OSPFNZpwKBdFEydrj+1+V5xo"
Vary: Accept-Encoding
Date: Sun, 19 Feb 2023 16:05:01 GMT
Connection: close

<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <link rel="icon" href="/favicon.ico" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <meta name="theme-color" content="#000000" />
    <meta
      name="description"
      content="Web site created using create-react-app"
    />
    <link rel="apple-touch-icon" href="/logo192.png" />
    <!--
      manifest.json provides metadata used when your web app is installed on a
```

Versions de HTTP

Il existe plusieurs versions du protocole HTTP. En 2024 :

- **HTTP/1.1** (22% des sites)
  - Le premier protocole standard
  - Les échanges sont lisibles par des humains
- **HTTP/2** (≈70% des sites)
  - Les communications sont binaires
  - Le serveur peut anticiper les demandes du client (push)
  - Amélioration du streaming
- **HTTP/3** (≈8% des sites)
  - standardisé en juin 2022
  - Basé sur QUIC et non plus TCP (communication multiplexe)

Source : `https://almanac.httparchive.org/en/2024/http`

Mais les principes restent les mêmes :

requête méthode/URL, en-têtes et corps

réponse statut, en-têtes et corps

HTTP – Requêtes

Anatomie d'une requête HTTP

HTTP est un protocole sans état ⇒ le serveur doit s'occuper de la gestion des sessions

- Le client envoie une requête composée de
  1. Méthode de requête (GET, POST, HEAD, *etc.*) (voir `https://fr.wikipedia.org/wiki/Liste_des_codes_HTTP`)
  2. Adresse de la ressource : Uniform Resource Identifier (URI)
  3. En-têtes
  4. Contenu (body)
- Le serveur renvoie
  1. Un code de statut (ex. 200 = OK, 404 = NOT_FOUND)
  2. En-têtes
  3. Contenu (body) : c'est un flux d'octets qui seront interprétés par le client (texte, image, *etc.*)

Requêtes : méthodes

SCIENCES SORBONNE UNIVERSITÉ

GET	On veut juste obtenir la ressource (ex. page web, image, etc.)
HEAD	On veut juste obtenir les en-têtes de la ressource (pour vérifier)
POST	Envoi de données
PUT	Remplacer une ressource
DELETE	Effacer une ressource
PATCH	Modifier une ressource

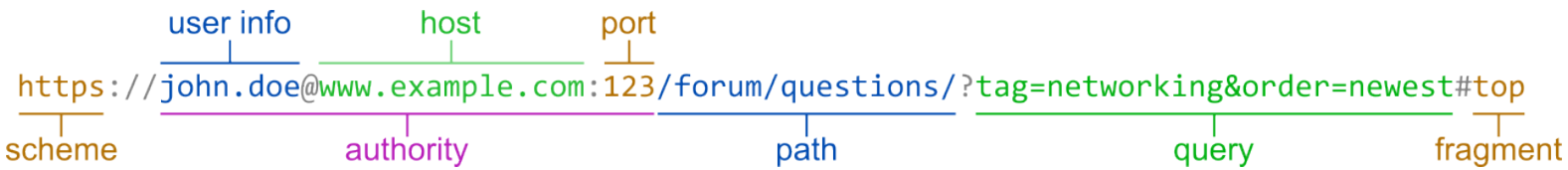
Requête : URL

En gros...

URI quelle ressource intéresse le client

Méthode comment interagir avec cette ressource

Anatomie d'une URI :



Différence URL, URI et URN

URI = URL + URN

URL Uniform Resource Locator

URN Uniform Resource Name (ne dit pas comment trouver la ressource, par exemple `ASIN: B001B8G5WM`)

URI Uniform Resource Identifier

Les principaux en-têtes de la requête

Host Le nom de l'hôte (ex. pour http ://www.sorbonne-universite.fr, c'est www.sorbonne-universite.fr)

Accept Les types (MIME) de contenu acceptés comme réponse (ex. `text/html` `text/jpg`, *etc.*)

Content-Length La taille (en octets) du corps de la requête

Cookie Liste de cookies HTTP

User-Agent Le type de client

Les principaux en-têtes de la réponse

Content-Length La taille (en octets) du corps de la réponse

Set-Cookie Demande pour stocker un cookie

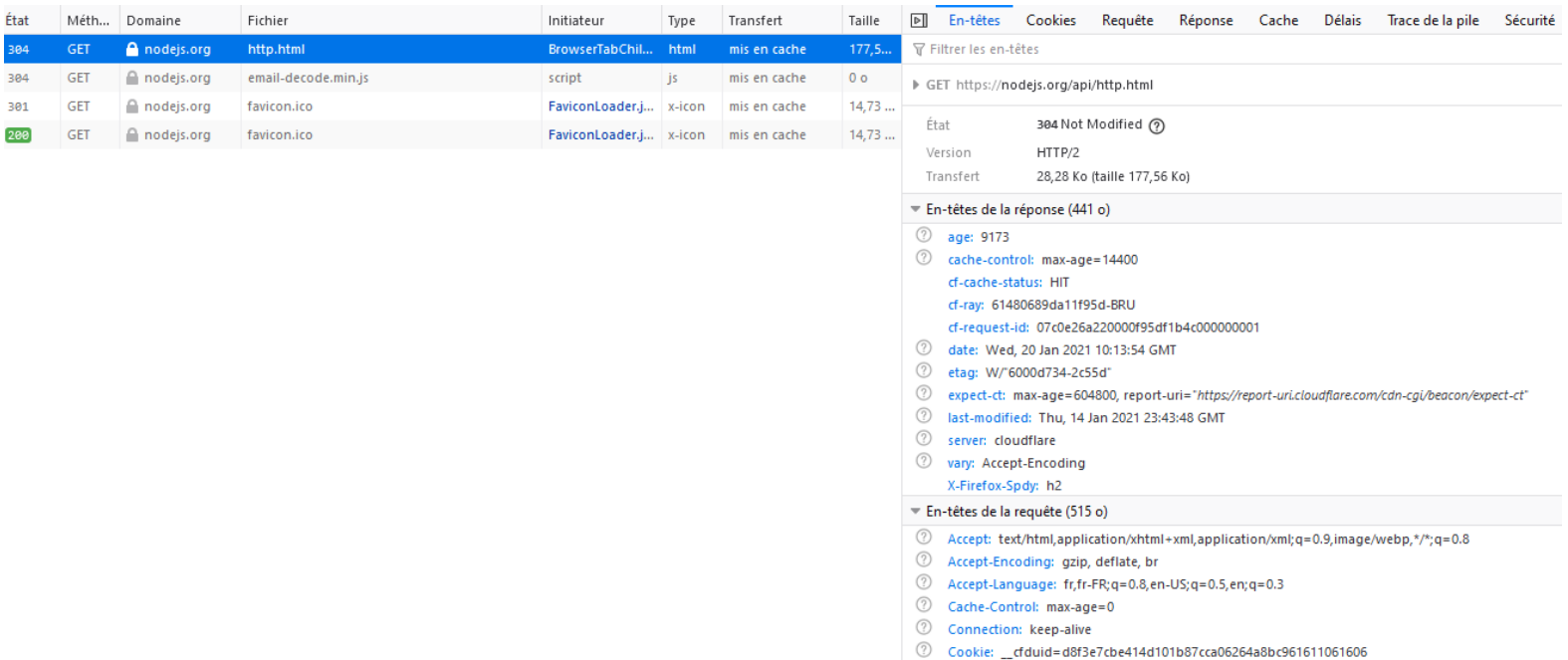
last-modified Date de dernière modification

Exemple

```
GET /wiki/Wikip%C3%A9dia:Accueil_principal HTTP/2
Host: fr.wikipedia.org
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0)
Gecko/20100101 Firefox/110.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/
Accept-Language: fr,fr-FR;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate, br DNT: 1 Connection: keep-alive
Cookie: WMF-Last-Access=19-Feb-2023;
WMF-Last-Access-Global=19-Feb-2023;
frwikimwuser-sessionId=c1e4f928db7a845c2811;
frwikiwmE-sessionTickLastTickTime=1676830803585;
frwikiwmE-sessionTickTickCount=1 Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document Sec-Fetch-Mode: navigate Sec-Fetch-Site:
cross-site If-Modified-Since: Sun, 19 Feb 2023 16:47:23 GMT TE:
trailers
```

Anatomie d'une requête HTTP

Utilisez le mode développeur pour observer les échanges réseaux (F12 sur la plupart des navigateurs)



Information de session

Problème : HTTP est un protocole sans état

Comment identifier une personne connectée en particulier quand plusieurs requêtes sont envoyées ? Comment savoir à qui renvoyer la réponse ?

Solution 1 : on modifie les URL

Exemple `http://monserveur.fr/show/1?user=x2u3sda21`

- – Compliqué à gérer
- – Problème de sécurité

Solution 2 : les cookies

Contrat entre le client (header de requête HTTP) et le serveur (header de réponse HTTP)

- \+ Une fois établi, le cookie est renvoyé à chaque requête
- – Si l'utilisateur ne veut pas de cookies...

Information de session (cookies)

Principe et illustration :

1. le serveur demande le stockage d'un cookie :
   Le serveur renvoie un en-tête "Set-Cookie" avec une valeur suivant le format standard :
   - Cookie qui expire avec la session (fermeture navigateur) :
     ```
     Set-Cookie: id=a3fWa
     ```
   - Cookie qui expire après 2592000 secondes (30 jours) :
     ```
     Set-Cookie: id=a3fWa; Max-Age=2592000
     ```
2. Le client communique les cookies stockés :
   Le client envoie avec *chaque demande* l'en-tête "cookie" avec la liste de tous les cookies enregistrés
   ```
   cookie: id=a3fWa; adv\_id=3asd21
   ```

Cookies et sécurité

Notion de sécurité : on ne communique que les cookies associés au domaine.
On ne renvoie pas les cookies de mabanqueenligne.fr à arnaqueur.com !

- Cookie sans domaine

  ```
  Set-Cookie: id=a3fWa
  ```

  Envoie le cookie seulement au même serveur (i.e. si le serveur est www.mozilla.org, n'envoie qu'à ce site et pas www2.mozilla.org)

- Cookie avec domaine

  ```
  Set-Cookie: id=a3fWa; Domain=mozilla.org
  ```

  Envoie le cookie à tous les sous-domaines de mozilla.org, i.e. les sites dont le nom de l'hôte se termine par .mozilla.org (ex. www.mozilla.org, sub.www.mozilla.org,*etc.*)

- Cookie avec domaine et chemin

  ```
  Set-Cookie: id=a3fWa; Domain=mozilla.org; Path=/profile
  ```

  N'envoie le cookie que si le chemin commence par `profile`

HTTP et sécurité : protocole HTTPS

Principe :

1. le client demande une connexion sécurisée et **propose** des méthodes de chiffrement ;
2. le serveur **choisit** une méthode de chiffrement et **renvoie** un certificat (délivré par une autorité tierce) ;
3. le client **vérifie** le certificat (à partir d'une liste d'autorités autorisées)
4. le certificat **contient une clé publique** pour crypter les données (mécanisme asymétrique) ;
5. le client **envoie** une clé de cryptage (symétrique) au serveur : cette clé sera utilisée pour toutes les communications à partir de ce moment-là

Serveur – Node.js

Problématique

Dans le cadre de l'UE, le client ET le serveur sont codés en JavaScript :

- Client = JavaScript chargé et interprété **par le navigateur**
- Serveur = Code JavaScript exécuté par **Node.JS**

Nous nous intéressons maintenant au côté **serveur**.

Il existe beaucoup de serveurs Web (Apache, Nginx, Lightspeed, IIS, *etc.*) mais nous utiliserons Express (un module Node.JS).

Et nous avons de bonnes raisons pour cela !

Regardons tout d'abord *sans* bibliothèque...

Hello World

SCIENCES SORBONNE UNIVERSITÉ

```
1  const http = require("http");
2  const monServeur = http.createServer( (req, res) =>{
3          // On peut accéder à req.headers, req.method, req.url
4
5          // Envoi de l'entête
6          res.writeHead(200,{"Content-Type": "text/plain"});
7          // Envoi du contenu
8          res.write('Hello World '+req.url);
9          // Nécessaire pour indiquer au client que l'envoi est
           ↪  terminé, et qu'il n'y a plus rien à attendre
10         res.end();
11 });
12 // Lancement du serveur sur le port 3000
13 monServeur.listen(3000);
```

où `req` et `res` sont deux objets :

- `req` représente la requête HTTP **reçue**
- `res`représente la réponse HTTP **en cours**

Le client envoie une requête GET au serveur sur le port 3000. Celui-ci répond par une réponse HTTP avec le code 200 (OK), et le texte « Hello World » puis l'URL à laquelle a été adressée la requête.

Le routage

createServer ne gère pas les routes : le codage d'un serveur peut donc être fastidieux !

```
if (req.url == '/hello') {
        // URL hello
} else if (req.url == '/world') {
        // Répondre à /world
} else if (m = req.url.match(/document\/(\w+)/)) {
        // Renvoie le document
        const docid = m[1]
}

// etc etc
```

Recourir à une bibliothèque

Deux possibilités dans le cadre de ce cours, les deux utilisant Node.js :

- Next.js : *framework* basé sur React, côté serveur pour faire du *server side rendering*, et plus efficace pour les moteurs de recherche
- Express.js : simplicité pour le routage et outils supplémentaires pour le HTTP (cache, redirection...)

Nous allons développer de simples API, pour lesquelles Express est plus adapté.

Express – Requêtes

Répondre à des requêtes : Express

Bibliothèque qui :

- permet le routage des requêtes
- permet de récupérer facilement des informations
- peut avoir de nouvelles fonctionnalités *via* des *middlewares*

```
1  const express = require('express');
2  const app = express();
3
4  app.get('/', (req, res) =>{
5          res.send("Hello World!");
6  });
7  app.get('/about', (req, res) =>{
8          res.send("Parlons un peu de moi...");
9  })
10 app.listen(3000, () => {console.log("Serveur OK!")});
```

Traitement des requêtes : grands principes

Principe 1 : chaîne de traitement

Une fonction de traitement peut avoir jusqu'à trois paramètres :

- La requête HTTP `req`
- La réponse HTTP `res`
- Une fonction de chaînage `next`

```
// Ajout d'un traitement
app.use((req, res, next) => {
        console.log("Appelé avec le chemin ", req.path)

        // Sans ça, le traitement suivant n'est pas appelé
        next()
})
```

Traitement des requêtes : grands principes

Principe 2 : la réponse

- En-Têtes HTTP `res.set(key, value)`
- Statuts HTTP `res.status(code)` (défaut : 200/OK)
- Données `res.send(data)` ou rien `res.end()`
  Attention : les en-têtes/statuts ne peuvent plus être envoyés une deuxième fois.

```
// Exemple
res.status(500);
res.send("Erreur serveur, désolé !");
```

Traitement des requêtes : grands principes

SCIENCES SORBONNE UNIVERSITÉ

Exemple

```
app.use((req, res, next) => {
        // N'agit que si
        // (1) la méthode est GET
        // (2) le chemin est "/about"
        if (req.method === "GET" && req.path === "/about") {
                // (implicite) Status 200
                // (implicite) Content-Type: text/html, Content-Size:...
                res.send("Let's talk about me...");
        } else {
                next();
        }
})
```

. . . est équivalent à. . .

```
app.get('/about', (req, res) => {
        res.send("Let's talk about me...")
});
```

Traitement des requêtes : grands principes

Les *middlewares*

- Les fonctions transmises sont appelés des *middlewares*
- Beaucoup de bibliothèques fournissent des *middlewares* Express :
  - Pour s'occuper des cookies (codage/décodage)
  - Pour s'occuper de l'authentification
  - *etc.*
- La fonction `app.use` est, elle, fournie par défaut par Express (pour le routage)
- Deux exemples de *middlewares* (voir plus loin) : sessions et fichiers statiques

Traitement des requêtes : La réponse

Statut et corps

- Pour spécifier le statut HTTP, on utilise `res.status(code)`
- `res.send(data)` renvoie un type de réponse différent en fonction de `data` :
  - `string` text/html
  - `object` ou tableau application/json
- On peut utiliser des méthodes spéciales comme raccourcis :
  - `res.redirect(url)` pour rediriger vers une autre page
  - `res.sendStatus(code)` pour envoyer le code et un message générique ("Page not found")
  - `res.sendFile(path [, options, fn])` pour renvoyer un fichier
  - `res.json(obj)` pour renvoyer du JSON
  - *etc.*

Express – Routage

Principe

Le routage Express permet de raccourcir l'écriture par rapport à Node.js seul.

Un chemin (ou un ensemble de chemins) est traité par des fonctions (*callbacks*, HANDLER)

```
app.METHOD(PATH, HANDLER)
```

où :

- `METHOD` est une méthode de demande HTTP (get, post, delete, patch, etc. ou all pour toutes)
- `PATH` est un *motif de chemin* sur le serveur (plus de détails plus tard)
- `HANDLER` est la fonction exécutée lorsque la route est mise en correspondance (arguments `req`, `res`, `next`)

Motifs de chemin

- sans paramètre :
  - `'/about'` : le chemin doit être exactement "/about"
  - `'/about/*'` : le chemin doit commencer par "/about/", ex. : /about/me, /about/me/and/you, ...
- avec paramètre :
  - `'/user/:user_id'` : le chemin doit être exactement "/user/..." : /user/232a (mais pas /user/23/2a)
  - `'/user/:user_id(\\d+)'` : user_id doit être une suite de chiffres
  - `'/post/:user_id(\\d+)/:post_id(\\d+)'` : avec deux paramètres

Liens utiles :

- pour tester les chemins : `http://forbeslindesay.github.io/express-route-tester/`
- documentation : `https://expressjs.com/fr/api.html#path-examples`

Grouper les motifs de chemin

Comment faire pour avoir un préfixe commun ?

```
const api = express.Router();

api.get("/user/:user_id", user_get);
api.post("/user/:user_id", user_post);

// Répond à /api/user/23 (GET ou POST)
app.use('/api', api);
```

Dans cet exemple, le chemin '/api/ZZZ' sera vu comme '/ZZZ' par le routeur (api)

Comment faire pour grouper des traitements ?

SCIENCES SORBONNE UNIVERSITÉ

Traitement de paramètre

```
const router = express.Router()
router.param('user_id', (req, res, next, user_id) => {
        // Tous les chemins du routeur avec :user_id seront
        // traités par cette fonction (une seule fois !)
        console.log("Fonction appelée en premier");
        // db.getuser renvoie un utilisateur avec l'ID correspondant
        req.user = db.getuser(user_id)

        // On appelle la prochaine fonction de traitement
        next()
})

// req.user sera l'utilisateur référencé par :user_id
router.get("/users/:user_id", (req, res) =>{
        console.log("Fonction appelée en deuxième");
        res.end();
})
router.patch("/users/:user_id", user_patch)
```

Avec le code précédent, l'appel à l'URL (...)/user/4242 affichera successivement dans la console "Fonction appelée en premier" puis "Fonction appelée en deuxième".

Comment faire pour grouper des traitements ?

SCIENCES SORBONNE UNIVERSITÉ

Traitement commun à une route

```
const router = express.Router()
// Ne pas mettre de ; en fins de ligne, ces requêtes sont chaînées
router
        .route("/user/:user_id")
        .all((req, res, next) => {
                req.monapp_user = db.getuser(req.params.user_id)
                next()
        })
        .get((req, res, next) =>
                res.send("get user "+req.monapp_user.name)
        )
        .post((req, res, next) =>
                res.send("post user "+req.monapp_user.name)
        )
```

Express, middlewares

Principe

Les *middlewares* sont des méthodes de l'objet `express` fournissant des raccourcis pour des actions usuelles.

Servir des fichiers statiques

Comment faire pour renvoyer des fichiers statiques, comme des images ou des feuilles de style CSS ?

On peut évidemment tout coder mais il y a plus simple... avec le *middleware* `static` :

```
app.use('/static', express.static('/path/to/public/folder'))
```

L'URI
`/static/css/style.css`
correspond au fichier
`/path/to/public/folder/css/style.css` On peut par exemple définir plusieurs répertoires contenant des fichiers statiques...

```
app.use('/static', express.static('/public'));
app.use('/static', express.static('/files'));
```

Servir des fichiers statiques (2)

Express recherche les fichiers relatifs au répertoire statique, donc le nom du répertoire statique ne fait pas partie de l'URL. On peut maintenant charger les fichiers présents dans le répertoire `public`:

```
http://localhost:3000/images/kitten.jpg
http://localhost:3000/css/style.css
http://localhost:3000/js/app.js
http://localhost:3000/images/bg.png
http://localhost:3000/hello.html
```

1. Si le fichier n'existe pas, appelle `next()`
2. Si le fichier existe, le renvoie (avec tous les bons en-têtes, taille et type)

Comment gérer des sessions ?

Solution 1 : cookies

- Codage : utilisez `app.cookie()`, cf la documentation
- Décodage : à vous de jouer (middleware conseillé cookie-parser)...

```
const cookieParser = require('cookie-parser')
app.use(cookieParser())

app.use((req, res, next) => {
        // cookie-parser initialise req.cookies
        let req.session_id = req.cookies.session_id
        if (req.session_id === undefined) {
                const options = {}
                req.session_id = session_uid_gen()
                res.cookie("session_id", req.session_id, options)
        }
        next();
})
```

Comment gérer des sessions ?

Solution 2 : *Middleware* `express-session`

C'est la solution la plus simple, que nous allons utiliser en projet.

- Permet d'obtenir un objet `req.session` où on peut stocker des informations
- Par défaut, les sessions sont en mémoire (mais on peut le paramétrer)

```
var session = require('express-session')
app.set('trust proxy', 1) // black magic

// Ajout du middleware (à faire avant les routes !)
app.use(session({
        secret: 'M07D3P455353Cr37',
        name: 'sessionId'
}));

app.use('/about/me', (req, res, next) => {
        if (req.session.aboutme_views) {
                req.session.aboutme_views += 1
        } else {
                req.session.aboutme_views = 1
        }
})
```

Gestion du JSON

Comment gérer du JSON ? Pour décoder du JSON (PUT/PATCH), on utilise le *middleware* `json` d'express.

```
router.use(express.json())
```

Fonctionnement :

- Si le type de fichier (`content-type`) est `application/json`...
- ... remplace req.body par un objet correspondant au JSON
- ... et permet de décompresser le JSON si jamais il était compressé

Il y a aussi l'équivalent `text` pour le texte et `urlencoded` pour les formulaires

Gestion du JSON – Exemple

```
// On appelle la méthode json()
app.use(express.json());

// Lecture du type de fichier (voir ci-après)
app.post('/', function (req, res) {
   console.log(req.body.name)
   res.end();
})

// Écoute du port
app.listen(PORT, function(err){
   if (err) console.log(err);
   console.log("Le serveur est à l'écoute sur le port ", PORT);
});
```

Si on fait une requête à l'API en :

- définissant le `content-type` dans l'entête à `application/json`
- passant dans le corps (`body`) de la requête `{"name" : "Appel API"}`

. . . alors la console du serveur affichera "Le serveur est à l'écoute sur le port " (3000 par exemple) puis "Appel API".

Traitement d'erreur

Gérer les erreurs :

```
// À mettre après les autres middlewares et routes
app.use(function(err, req, res, next) {
        res.status(500);
        res.render('error', { error: err });
});
```

Signaler une erreur :

- On transmet un argument à `next`
- Seuls les *middlewares* avec 4 arguments seront appelés pour le traitement

```
// À mettre après les autres middlewares et routes
app.param("user_id", function(req, res, next, user_id) {
        const user = user_get(user_id)
        if (user === undefined) {
                        // Aïe...
                        next("Utilisateur non défini")
        } else next()
});
```

Bilan

HTTP

Protocole de communication

- Requête = méthode, URI, en-têtes et corps
- Réponse = statut, en-têtes et corps
- (pas vu en cours) Le standard permet aussi de gérer le cryptage, les flux de données, *etc.*

Express

Surcouche de Node.JS permettant de gérer les requêtes HTTP

- Accès direct à la requête (`req`) et la réponse (`res`)
- Basés sur une succession de traitements (*middleware*) :
  - Définition de traitements pour les chemins, ex. `.get(...)`
  - Définition de *middlewares* pour les fichiers, le JSON, *etc.*
  - Gestion des erreurs
  - Arrêt du traitement si pas d'appel à `next()`

À vous de trouver les *middlewares* qui vous intéressent :
`https://expressjs.com/fr/resources/middleware.html`