
UE 702 - MIOA702T -

Partie « Framework web côté serveur »

Nathalie Hernandez
nathalie.hernandez@univ-tlse2.fr



Objectifs

- Apprendre à faire des scripts en Javascript avec NodeJS
- Savoir mettre en place un serveur web léger avec NodeJS et Express

Plan de cours

➤ NodeJS, le moteur Javascript

- Rappels Javascript serveur / Javascript client
- Module, export, require et npm
- L'accès au système de fichiers
- Tester son script nodeJS
- Mini TP
 - Installation de NodeJS
 - petit script de lecture de fichiers
- NodeJS comme serveur web
 - Rappels HTTP
 - Les requêtes et les réponses HTTP
 - Les templates
- Mini TP : appel du script de lecture via une api

➤ Express, le framework web pour NodeJS

- Introduction et fonctionnement d'Express
- Le module nodejs path
- Les objets Request et Response
- Mini TP
- Les templates
- Les middlewares d'Express
- Le routing avec Express

➤ Création d'un squelette d'application avec express-generator

NodeJS, le moteur javascript

Rappels Javascript serveur / Javascript client

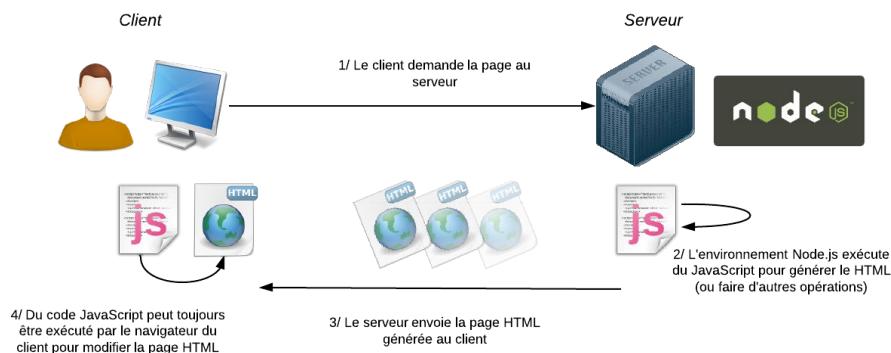
NodeJS, le moteur Javascript

Rappels Javascript serveur/Javascript client

- Le javascript est un langage de script qui peut être exécuté sur différents environnements :
 - Dans un navigateur web, on parle de javascript côté client



- Dans une machine virtuelle directement exécuté sur le système d'exploitation, on parle de javascript côté serveur

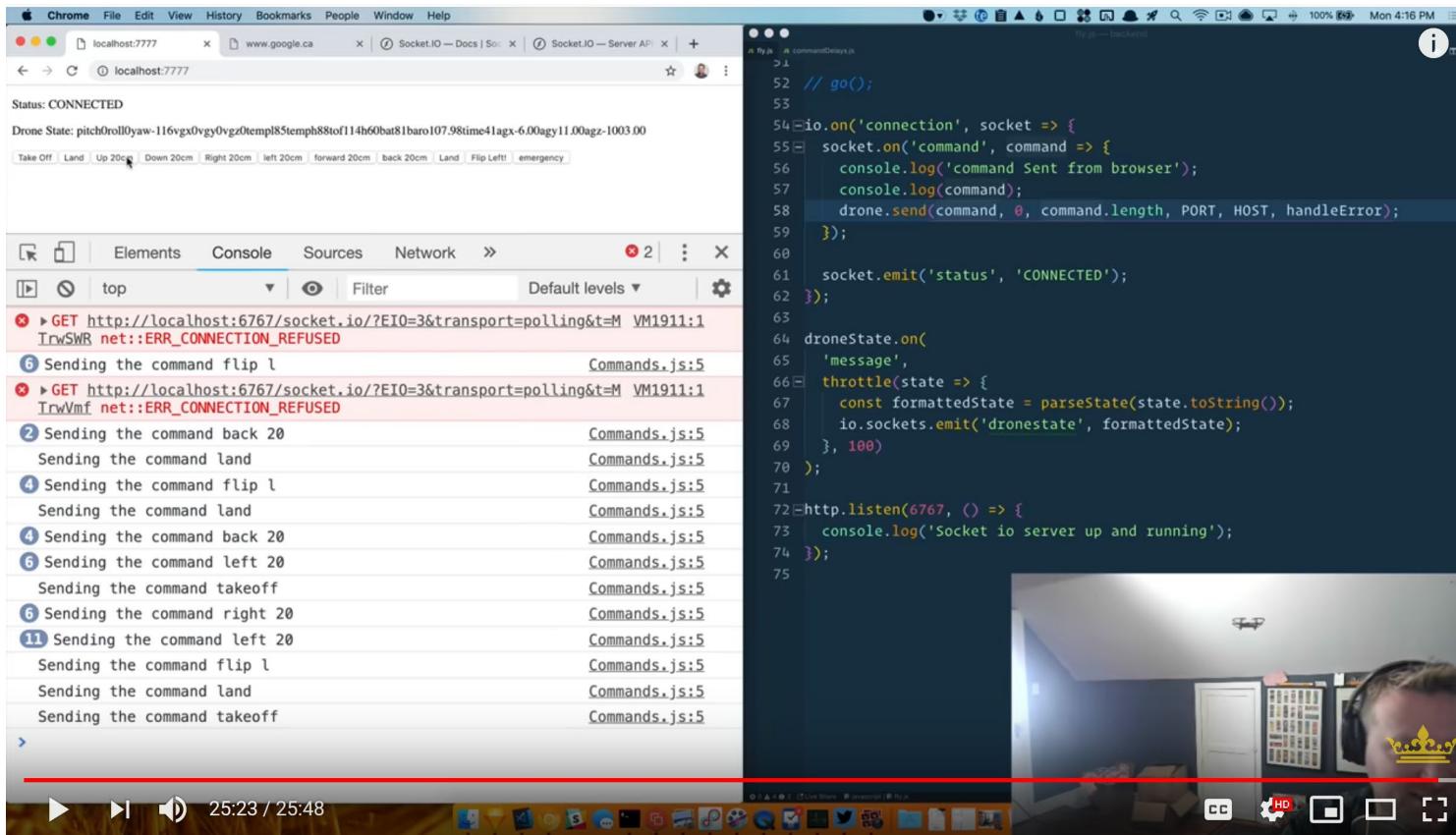


NodeJS, le moteur Javascript

Rappels Javascript serveur/Javascript client

- On peut également utiliser le javascript pour faire de l'embarqué ou des objets connectés!

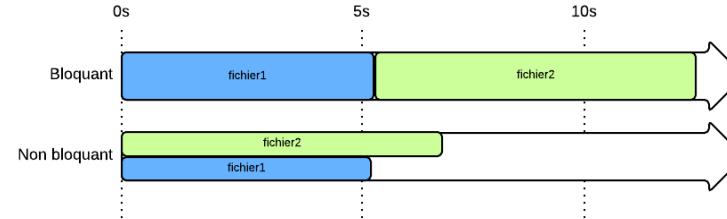
Par exemple, faire voler un drone : <https://www.youtube.com/watch?v=JzFvGf7Ywkk>



NodeJS, le moteur Javascript

Rappels Javascript serveur/Javascript client

- NodeJS est un environnement côté serveur, développé par Ryan Dahl en 2009, maintenu par la société Joyent.
- Il s'exécute sur la machine virtuelle V8, développé pour javascript par Google
- Sa rapidité vient de son modèle non bloquant, qui permet l'exécution de plusieurs instructions simultanément



- Via ses modules natifs, il peut
 - Répondre à des requêtes HTTP
 - Accéder au système de fichiers
 - Tout ce que vous faites en java peuvent être fait en node JS

<https://v8.dev>

NodeJS, le moteur Javascript

Rappels Javascript serveur/Javascript client

Quelques différences notables entre du javascript serveur et du javascript client :

- Le JS client est isolé dans le navigateur.
 - Il a accès aux api du navigateur (l'objet *document* par exemple)
 - Mais il n'a pas le droit d'accéder au système de fichiers
 - Il peut utiliser les périphériques de l'utilisateur (micro, caméra...) en demandant au navigateur de demander à l'utilisateur (« Autorisez vous www.google.com à accéder à votre micro? »)
 - Il est mono thread
- Le JS serveur s'exécute sur le système d'exploitation
 - Il a donc accès au système de fichiers
 - *Single Threaded Event Loop*

NodeJS, le moteur Javascript

Rappels Javascript serveur/Javascript client

Exemple d'un script « hello world » avec NodeJS

```
JS helloWorld.js
1   console.log("hello world !");|
```

Pour exécuter le script, via le terminal :

```
$ node helloWorld.js
hello world !
```

Et voilà!

NodeJS, le moteur Javascript

Rappels Javascript serveur/Javascript client

Exemple d'un script avec le nom de l'utilisateur en argument d'entrée

Pour l'exécuter :

```
$ node helloWorld.js blandine
Hello blandine!
```

Et si l'on ne donne pas d'argument, le script retourne une erreur :

```
$ node helloWorld.js
Missing argument! Example: node helloWorld.js YOUR_NAME
```

```
JS helloWorld.js > ...
1  #!/usr/bin/env node
2
3  'use strict';
4
5  /*
6   * The command line arguments are stored in the `process.argv` array,
7   * which has the following structure:
8   * [0] The path of the executable that started the Node.js process
9   * [1] The path to this application
10  [2-n] the command line arguments
11
12  Example: [ '/bin/node', '/path/to/yourscript', 'arg1', 'arg2', ... ]
13  src: https://nodejs.org/api/process.html#process\_process\_argv
14 */
15
16 // Store the first argument as username.
17 const username = process.argv[2];
18
19 // Check if the username hasn't been provided.
20 if (!username) {
21     // Give the user an example on how to use the app.
22     console.error('Missing argument! Example: node helloWorld.js YOUR_NAME');
23
24     // Exit the app (success: 0, error: 1).
25     process.exit(1);
26 }
27
28 // Print the message to the console.
29 console.log('Hello %s!', username);
```

NodeJS, le moteur javascript

Module, export, require et npm

NodeJS, le moteur Javascript Module, export, require et npm

NodeJS a un système de gestion de module intégré(CommonJS).

Un fichier NodeJS peut importer une fonctionnalité exposée par un autre fichier NodeJS.

Pour importer une fonctionnalité d'un fichier, utiliser le mot clé **require** :

```
const library = require('./library');
```

Dans le fichier *library.js*, la fonctionnalité doit être exposée via l'api **module.exports** :

```
const library = {
  brand: 'Ford',
  model: 'Fiesta'
}
module.exports = library
```

<https://nodejs.dev/expose-functionality-from-a-nodejs-file-using-exports>

NodeJS, le moteur Javascript Module, export, require et npm

npm est le gestionnaire de paquets standard pour NodeJS.

- Il existe plus de 300 000 paquets sur le registre npm. C'est le plus gros registre de paquet pour un langage de programmation.
- Il existe des paquets pour à peu près tout (et n'importe quoi aussi ^^)
- Certains paquets sont exclusifs pour nodeJS, d'autres pour le javascript client (dans le navigateur donc).
- Certains paquets peuvent être utilisés sur les deux plate formes.
- Des alternatives à *npm* existent. Par exemple, *yarn*.

NodeJS, le moteur Javascript

Module, export, require et npm

Installer un paquet se fait via la commande *install* :

```
npm install <package-name>
```

- Cette commande installe le paquet *package-name* et crée :
 - un fichier *package-lock.json* qui contient la version du paquet installé
 - Un dossier *node_modules* contenant le paquet installé

package-lock.json et *node_modules* sont créés dans le dossier où l'on a exécuté la commande *npm install*

NodeJS, le moteur Javascript Module, export, require et npm

Lorsque l'on récupère un projet javascript existant, *npm* va nous aider à récupérer la liste de toutes les dépendances facilement.

Il faut :

- un fichier *package.json* à la racine du projet
- Et lancer la commande *npm install*

Le fichier *package.json* contient

- les informations essentielles du projet (nom, type de licence, auteur...)
- Les dépendances npm
- Des tâches pour lancer des scripts de tests, de packagin ou autre

NodeJS, le moteur Javascript

Module, export, require et npm

Exemple de fichier package.json :

```
{} package.json > ...
1  {
2    "name": "m1-ice-example",
3    "version": "1.0.0",
4    "description": "",
5    "main": "script.js",
6    "dependencies": {
7      "lodash": "^4.17.15"
8    },
9    "devDependencies": {},
10   "scripts": {
11     "test": "echo \\\"Error: no test specified\\\" && exit 1"
12   },
13   "author": "Blandine",
14   "license": "MIT"
15 }
```

<https://www.npmjs.com/>

NodeJS, le moteur Javascript Module, export, require et npm

Pour générer ce fichier package.json, lancer la commande

```
npm init
```

Et suivre les instructions.

Pour ajouter de nouveaux paquets au projet, et les enregistrer :

```
npm install --save-prod <package-name> //pour un paquet utilisé en production
```

```
npm install --save-dev <package-name>
```

```
// pour un paquet utilisé en développement (par exemple, les outils de tests)
```

<https://docs.npmjs.com/cli/install>

NodeJS, le moteur javascript

L'accès au système de fichier

NodeJS, le moteur Javascript

L'accès au système de fichiers

Il existe deux manières de lire un fichier avec NodeJS

- Synchrone : la lecture bloque l'exécution du script
- Asynchrone : la lecture est non bloquante, le script continue à s'exécuter et un callback est appelé lorsque la lecture est terminée.

Toute la puissance de NodeJS est justement sa capacité à traiter les instructions en asynchrone, nous allons donc nous concentrer sur cette façon de faire.

Pour lire un fichier en asynchrone, on utilise la méthode *readFile* de la class *fs* de NodeJS :

- On récupère la class *fs* via *require*
- *readFile* a trois paramètres en entrée :

- Le chemin du fichier
- L'encoding du fichier (attention : si on ne le précise pas, on récupère un buffer)
- Une fonction de callback, qui sera appelée de manière asynchrone, lorsque la lecture du fichier sera terminée. Le callback a deux paramètres : soit une erreur, si la lecture ne s'est pas bien passée, soit les données du fichier

```
fs.readFile('/Users/joe/test.txt', 'utf8', (err, data) => {
  if (err) {
    console.error(err)
    return
  }
  console.log(data)
})
```

<https://nodejs.org/api/fs.html>

NodeJS, le moteur javascript

Mini TP

NodeJS, le moteur Javascript

Mini TP : Script de lecture de fichier

- Installer NodeJS
- Créer un dossier *my-app*
- Dans ce dossier *my-app*, créer le fichier package.json avec *npm init*
- Créer un fichier *data.csv* contenant :

```
User1;toulouse;  
User2;toulouse;
```

- Créer un fichier *script.js* qui contiendra le code de l'application
- Coder un script qui lit le fichier *data.csv* et affiche son contenu dans le terminal

Résultat attendu :

```
$ node script.js  
data :  
: User1;toulouse;  
User2;toulouse;
```

- **Ajout** : le nom du fichier de données est fourni en argument lors de l'appel au script.
- **Ajout** : une nouvelle ligne de données est fourni en argument et est rajoutée au fichier de données
- **Ajout** : une nouvelle version du script doit être réalisée pour prendre en compte un fichier au format JSON

NodeJS, le moteur javascript

NodeJS comme serveur web

NodeJS, le moteur Javascript NodeJs comme serveur web

NodeJS a des fonctionnalités natives pour fonctionner en serveur web, c'est-à-dire être capable de répondre à des requêtes HTTP.

Le module nodeJS pour cela est *http*.

NodeJS, le moteur javascript

NodeJS comme serveur web

Rappels HTTP

NodeJS, le moteur Javascript

NodeJs comme serveur web – Rappels HTTP

- HTTP : HyperText Transfert Protocol, est un protocole de communication client/serveur pour le web
 - Un client ouvre une connexion vers un serveur
 - Le client effectue une requête et attend
 - Le serveur répond
- Les requêtes HTTP peuvent être de type :
 - GET : Récupération de données
 - POST : Envoi de données au serveur
 - PUT : Mise à jour de données
 - DELETE : Suppression de données
- Les réponses ont un code, et éventuellement des données. Les codes les plus courants sont :
 - 200 : tout s'est bien passé
 - 404 : la ressource n'a pas été trouvée
 - 401 : accès non autorisé
 - 500 : une erreur s'est produite

<https://developer.mozilla.org/fr/docs/Web/HTTP>

<https://developer.mozilla.org/fr/docs/Web/HTTP/M%C3%A9thode/GET>

<https://developer.mozilla.org/fr/docs/Web/HTTP/M%C3%A9thode/POST>

<https://developer.mozilla.org/fr/docs/Web/HTTP/M%C3%A9thode/PUT>

<https://developer.mozilla.org/fr/docs/Web/HTTP/M%C3%A9thode/DELETE>

NodeJS, le moteur javascript

NodeJS comme serveur web

Requetes et réponses HTTP

NodeJS, le moteur Javascript

NodeJs comme serveur web – Requetes et réponses HTTP

Exemple d'un serveur web simple :

```
js server.js > ...
1 const http = require('http') // import du module http
2
3 const port = 3000; // numéro de port sur lequel le serveur va écouter
4
5 const server = http.createServer( // création du serveur HTTP
6   (req, res) => { // à chaque requête arrivant au serveur, cette fonction sera appelée avec req (contenu de la requête) et res (objet de réponse à compléter et renvoyer)
7     res.statusCode = 200 // le code HTTP de retour sera 200
8     res.setHeader('Content-Type', 'text/html') // on informe qu'on va renvoyer du HTML
9     res.end('<h1>Hello, World!</h1>') // HTML que l'on envoie
10  }
11
12 server.listen(port, () => { // quand le serveur a été créé, il se met à écouter sur le port 3000
13   console.log(`Server running at port ${port}`)
14 })
```

On exécute le serveur de la même façon qu'un script simple, puis on peut l'interroger via le navigateur (ou curl, ou autre...)

```
$ node server.js
Server running at port 3000
```



<https://nodejs.dev/build-an-http-server>

NodeJS, le moteur Javascript

NodeJs comme serveur web – Requetes et réponses HTTP

Utiliser le module *http* de nodeJS, sans surcouche, est assez fastidieux. C'est pour cela que nous verrons en seconde partie de ce cours, *express*, un framework web pour Node qui nous simplifiera la vie.

Mais en attendant ...

L'objet *request* passé en paramètre de la fonction *createServer* est un stream. Il faut donc lire ce stream, morceau par morceau (appelé *chunk*) pour récupérer le contenu de la requête.

Imaginons que l'on souhaite envoyer une requête contenant du JSON. Voici le script pour que le serveur puisse le lire :

```
 5  const server = http.createServer((req, res) => {
 6    let data = [];
 7    req.on("data", chunk => {
 8      data.push(chunk);
 9    });
10   req.on("end", () => {
11     JSON.parse(data).todo; // 'Achète du pain!'
12   });
13   res.statusCode = 200;
14   res.setHeader("Content-Type", "text/html");
15   res.end("<h1>Bien reçu!</h1>");
16 });


```

<https://nodejs.dev/get-http-request-body-data-using-nodejs>

NodeJS, le moteur javascript

NodeJS comme serveur web

Les templates

NodeJS, le moteur Javascript

NodeJs comme serveur web – Les templates

Comme avec d'autres serveurs web (Django, php...), on peut renvoyer directement des templates HTML.

Il n'y a pas de solution de templating natif avec NodeJS, mais beaucoup de librairies existent :

- Pug : <https://github.com/pugjs/pug>
- Ejs : <https://github.com/tj/ejs>
- React (et oui, mais généré sur le serveur!) : <https://github.com/reactjs/express-react-views>

Dans ce cours, nous allons utiliser Pug.

Installation via *npm* : `npm install pug`

Création d'un fichier template.pug

```
template.pug
1  p #{name}'s Pug source code!
```

On renvoie ce fichier avec notre serveur :

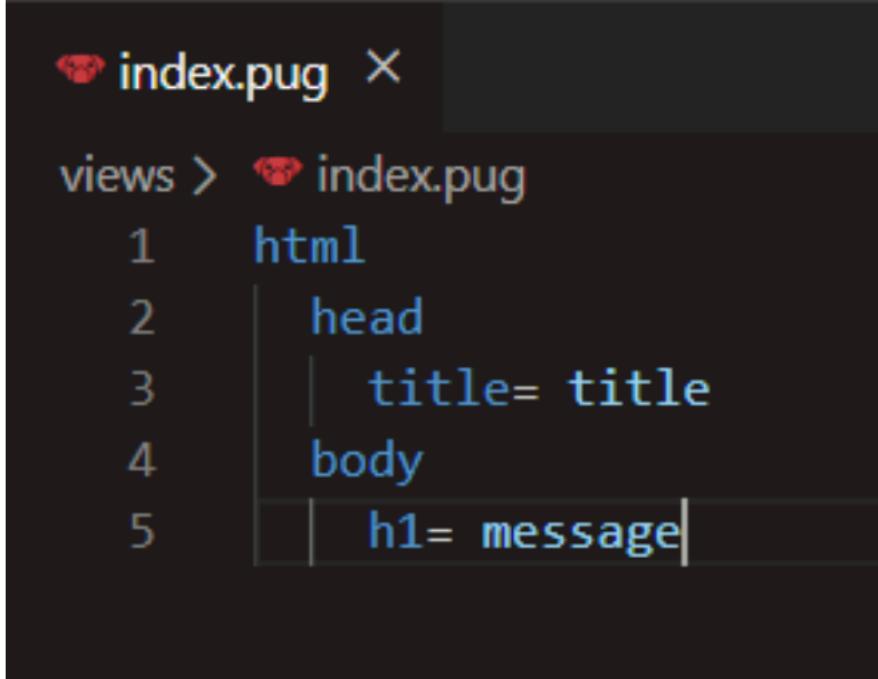
NodeJS, le moteur Javascript

NodeJs comme serveur web – Les templates

Server.js :

```
JS server.js > ...
1  const http = require("http");
2  const pug = require("pug"); // import du module pug
3
4  const compiledFunction = pug.compileFile('template.pug'); // compilation du template
5  const port = 3000;
6
7  const server = http.createServer((req, res) => {
8      const generatedTemplate = compiledFunction({
9          name: 'Blandine'
10     }); // génération du template avec une variable pour "name"
11
12     res.statusCode = 200;
13     res.setHeader("Content-Type", "text/html");
14     res.end(generatedTemplate); // on renvoie le template
15   });
16
17   server.listen(port, () => {
18     console.log(`Server running at port ${port}`);
19   });

```



The image shows a code editor window with a dark theme. The title bar says "index.pug" with a close button. Below the title bar, it shows the path "views > index.pug". The code itself is a Pug template:

```
1  html
2   |   head
3   |     title= title
4   |   body
5     h1= message
```

NodeJS, le moteur javascript

Mini TP

NodeJS, le moteur Javascript

Mini TP : appel du script de lecture via une API web

- Transformer le script de lecture de fichier pour en faire un serveur web
- Lorsque l'on envoie une requête au serveur, il renvoie une page HTML contenant :
 - Le titre : « Voici le contenu du fichier »
 - Un tableau du contenu du fichier, avec des entêtes :

Identifiant	Ville
User1	Toulouse
User2	Toulouse

<https://pugjs.org/language/interpolation.html>
<https://pugjs.org/language/iteration.html>

Bonus : le nom du fichier de données est fourni en paramètre de la requête.

Express, le framework web pour NodeJS

Introduction

Express, le framework web pour NodeJS

Introduction

Express est un framework web minimaliste pour NodeJS. Il contient les principales fonctionnalités nécessaires à la mise en place d'un serveur web :

- Des méthodes utilitaires pour gérer les appels HTTP
- Des middlewares (des fonctions qui viennent s'intercaler dans le processus requête/réponse) pour gérer la sécurité, l'accès à la base de données, le routing ...etc

L'installation d'express se fait, comme d'habitude, via npm :

```
npm install express
```

Au niveau du code, Express est une surcouche au module http natif de NodeJS, que nous avons vu précédemment, pour donner au développeur une api plus abstraite et légère.

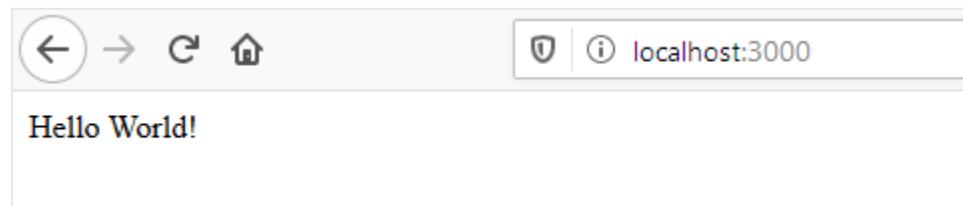
<https://expressjs.com/en/starter/installing.html>

Express, le framework web pour NodeJS

Introduction

Exemple d'un serveur js avec express :

```
JS expressServer.js > ...
1  const express = require('express'); // import de la librairie
2  const app = express(); // express expose un objet app qui sera notre serveur.
3  // on ne touche plus directement au module http de nodeJS.
4  // c'est express qui le fait pour nous
5
6  const port = 3000; // définition du port
7
8  // déclaration d'une route / avec la fonction get() qui retournera 'Hello world'
9  app.get('/', (req, res) => res.send('Hello World!'));
10
11 // démarrage du serveur (comme lorsqu'on est en pur nodeJS)
12 app.listen(port, () => console.log(`Example app listening on port ${port}!`));
```



Express, le framework web pour NodeJS

La gestion des fichiers statiques avec le module nodeJS path

Express, le framework web pour NodeJS

La gestion des fichiers statiques avec le module nodeJS path

Un serveur web renvoie du contenu dynamique généré à la volée comme du HTML ou du json mais aussi des ressources statiques :

- Images
- Icônes
- Fichiers CSS

Ces ressources ne sont pas modifiés et on veut donc pouvoir les renvoyer simplement, sans avoir à définir des routes pour chaque requête.

Toutes ces ressources vont être dans un dossier public, et on va demander à express de renvoyer son contenu à la demande.

<https://expressjs.com/en/starter/static-files.html>

Express, le framework web pour NodeJS

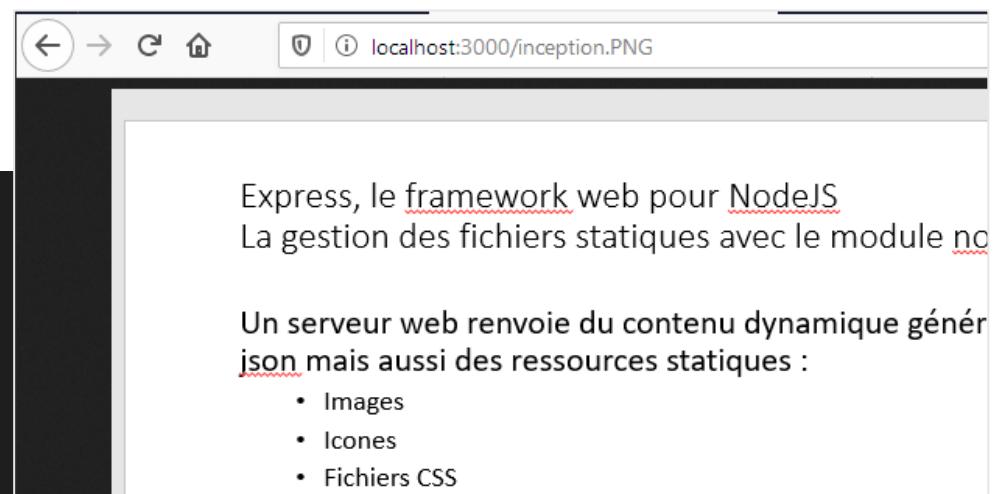
La gestion des fichiers statiques avec le module nodeJS path

On utilise la fonction *static()* d'express:

```
express.static(root); // root étant le chemin du dossier des ressources statiques
```

Intégré à notre serveur précédent, pour servir le dossier public qui contient une image inception.PNG (url => localhost:3000/inception.PNG) :

```
JS expressServer.js > ...
1  const express = require('express');
2  const app = express();
3
4  app.use(express.static('public'))
5
6  const port = 3000;
7
8  app.get('/', (req, res) => res.send('Hello World!'));
9
10 app.listen(port, () => console.log(`Example app listening on port ${port}!`));
```



Express, le framework web pour NodeJS

La gestion des fichiers statiques avec le module nodeJS path

Le code précédent fonctionne, mais on peut faire mieux.

En effet, on spécifiait le chemin relatif du dossier public. Dans une configuration simple, ça va, mais si le serveur est lancé depuis un autre dossier par exemple, ça ne va pas fonctionner.

On va donc utiliser le chemin absolu du dossier, que l'on va définir à la volée avec le module nodeJS *path* :

```
app.use(express.static(path.join(__dirname, 'public')));
```

path.join va concaténer les chemins qu'on lui donne en paramètre.

__dirname est un objet global qui contient le chemin absolu du répertoire contenant le script en train de s'exécuter. Dans notre cas, le chemin absolu du répertoire contenant *server.js*

https://nodejs.org/api/path.html#path_path_join_paths

<https://alligator.io/nodejs/how-to-use dirname/>

Express, le framework web pour NodeJS

Les objets *request* et *response*

Express, le framework web pour NodeJS

Les objets *request* et *response*

```
app.get('/', (req, res) => res.send('Hello World!'));
```

Lorsque l'on définit une route avec Express, (par exemple une route / comme ci-dessus), la fonction de callback qui sera appelé a deux paramètres :

- *req* : qui est un objet Request fourni par Express. C'est une représentation de la requête HTTP qui contient les propriétés de la requête. Le nom *req* est une convention.

Exemple de récupération d'un paramètre :

```
app.get('/', (req, res) => res.send('Hello World!'));

// on définit une route /user avec une partie dynamique /:id
// "/user/1" "/user/2" "/user/3" sont trois requêtes qui appellerait cette route
app.get('/user/:id', function (req, res) {
  res.send('user ' + req.params.id) // le paramètre id se trouve dans l'objet params de l'objet req
})
```

De nombreuses méthodes et propriétés sont disponibles sur cet objet. Par exemple, *req.body* permet de récupérer les informations d'une requête *POST*. N'hésitez pas à consulter la documentation de l'API Express en fonction des cas d'utilisation : <https://expressjs.com/fr/4x/api.html#req>

Express, le framework web pour NodeJS

Les objets *request* et *response*

- *res* : qui est un objet Response fourni par Express. C'est une représentation de la réponse HTTP qui sera envoyée. Le nom *res* est une convention.

La méthode principale est *send*, pour envoyer une réponse HTTP, contenant éventuellement des informations comme un code HTTP, du JSON, de l'HTML...etc :

```
res.send(Buffer.from('whoop')) // Buffer
res.send({ some: 'json' }) // JSON
res.send('<p>some html</p>') // HTML
res.status(404).send('Sorry, we cannot find that!') // erreur HTTP et String
res.status(500).send({ error: 'something blew up' }) // erreur HTTP et objet JS
```

La fonction *res.status* permet d'ajouter un code d'erreur à la réponse HTTP.

De nombreuses méthodes et propriétés sont disponibles sur cet objet. N'hésitez pas à consulter la documentation de l'API Express en fonction des cas d'utilisation : <https://expressjs.com/fr/4x/api.html#res>

Express, le framework web pour NodeJS

Les objets *request* et *response*

Lorsqu'on écrit un serveur, c'est pratique de pouvoir l'interroger facilement.

localhost:3000 depuis le navigateur, c'est sympa, mais assez limité dès qu'on va vouloir écrire des requêtes un peu complexes comme des *POST*.

On va donc utiliser l'outil *POSTMAN* (ci-dessous, on interroge la route */user/:id* donc on a parlé juste avant):

The screenshot shows the Postman application window. On the left, there's a sidebar with 'History' selected, showing dates like 'Today', 'January 17', and 'October 30'. The main area has a title bar 'Untitled Request' with a 'GET' method and the URL 'http://localhost:3000/user/1'. Below this, under 'Params', there's a table with one row: 'Key' (Value) and 'Value' (Description). At the bottom, the status bar shows 'Status: 200 OK Time: 11ms Size: 209 B'. To the right of the interface, three blue curly braces group elements: the URL section, the parameters section, and the response section.

URL

Paramètres de la requête

Réponse du serveur

Attention ! Le serveur doit bien sur être lancé pour que ça fonctionne! (`$ node server.js`)

Express, le framework web pour NodeJS

Les templates

Express, le framework web pour NodeJS

Les templates

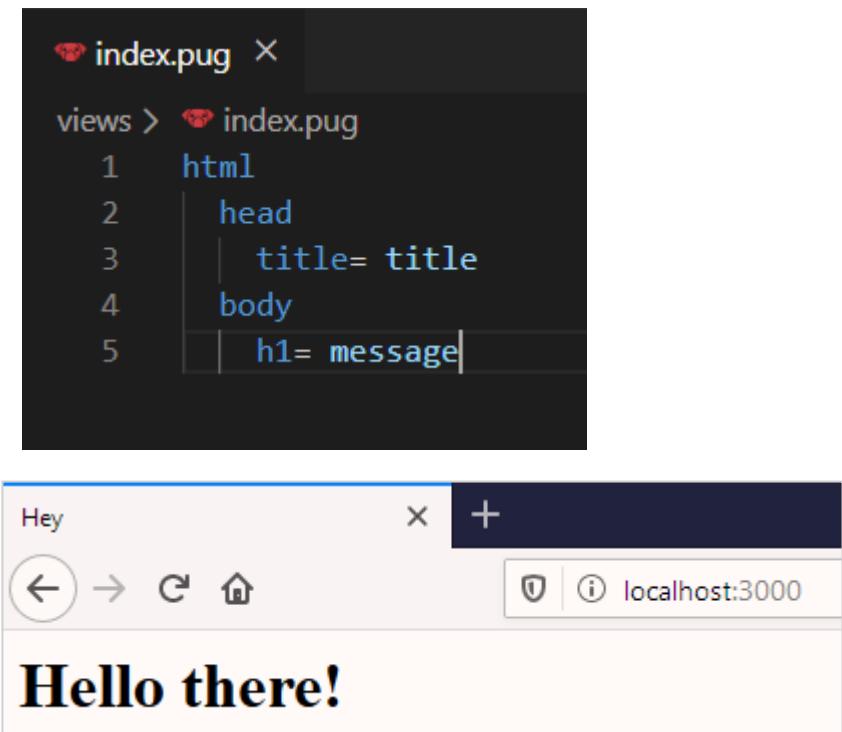
On peut continuer à utiliser le moteur de template pug avec express, mais :

- on le définit plus simplement avec `app.set('view engine', 'pug')`
- Les noms des fichiers de templates sont automatiquement cherché dans le dossier `/views` par express (**il faut donc créer ce dossier et mettre tous vos templates dedans!**)
- On utilise la fonction `render` de `res` pour renvoyer le template.

Exemple d'une route / qui renvoie un template index.pug :

```
1 const express = require('express');
2 const app = express();
3 const port = 3000;
4
5 app.set('view engine', 'pug');
6
7 app.get('/', function (req, res) {
8   res.render('index', { title: 'Hey', message: 'Hello there!' })
9 })
10
11 app.listen(port, () => console.log(`Example app listening on port ${port}!`));
```

<http://expressjs.com/en/guide/using-template-engines.html#using-template-engines-with-express>



Express, le framework web pour NodeJS

Mini TP

Express, le framework web pour NodeJS

Mini TP : Transformation du serveur avec Express

- Installer Express
- Installer Postman : <https://www.postman.com/downloads/>
- Transformer le serveur des TP précédents en serveur express et interrogez-le avec postman
- **Bonus** : Ajouter une gestion des fichiers statiques qui permet de récupérer des photos de chat (à vous de les trouver!).

Tuto Postman en vidéo : https://www.youtube.com/watch?v=FjgYtQK_zLE

Express, le framework web pour NodeJS

Quelques middlewares

Express, le framework web pour NodeJS

Quelques middlewares

Les middlewares sont des fonctions qui ont accès à l'objet de requête *req* à l'objet de réponse *res*.

Les middlewares vont venir s'intercaler dans le cycle de requêtage pour exécuter des fonctions.

Lorsqu'une fonction middleware a finit de s'exécuter, elle appelle *next()* pour que le middleware suivant s'exécute.

Un middleware peut effectuer les tâches suivantes :

- Exécuter du code
- Effectuer des changements sur les objets de requête et de réponses
- Stopper le cycle requête/réponse (par exemple dans le cas d'un middleware de sécurité), les middleware suivant ne s'exécuteront pas.
- Appeler le middleware suivant avec *next()*

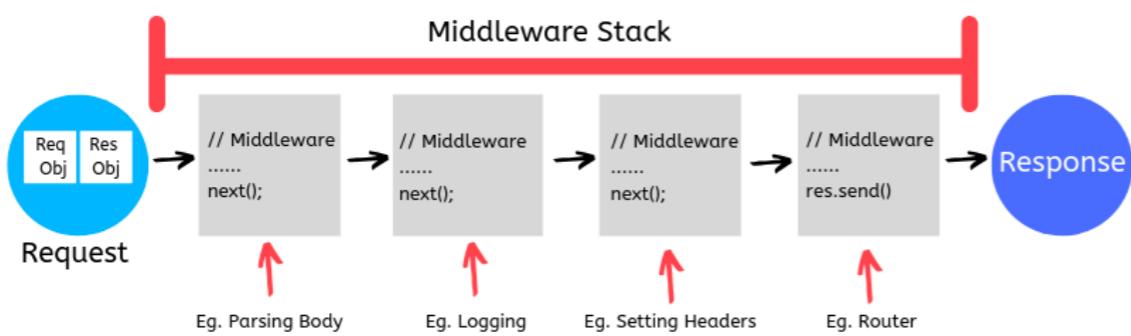
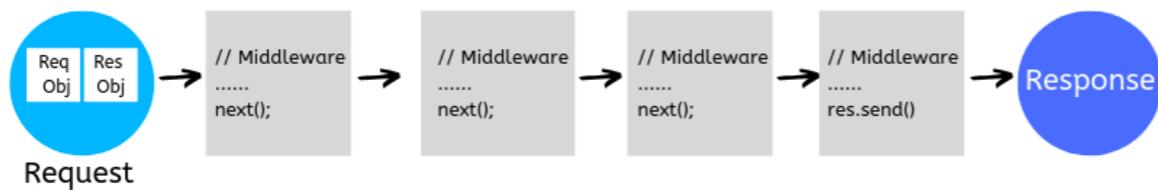
<http://expressjs.com/en/guide/using-middleware.html>

https://www.youtube.com/watch?v=_GJKAs7A0_4

<https://www.youtube.com/watch?v=lY6icfhap2o>

Express, le framework web pour NodeJS

Les middlewares



Express, le framework web pour NodeJS

Quelques middlewares

Exemple d'un middleware qui va logguer toutes les requêtes effectuées sur le serveur :

```
2  const express = require('express');
3  const app = express();
4  const port = 3000;
5
6  app.use(function (req, res, next) {
7    console.log('Request:', req.path);
8    next();
9  })
```

Désormais, on peut voir dans la console tous les appels :

```
Example app listening on port 3000!
Request: /
Request: /test
```

Et le serveur continue à fonctionner de la même façon, en appelant, après le middleware de log, la route adéquate.

Express, le framework web pour NodeJS

Quelques middlewares

Middlewares TIERS

Il existe plein de middleware. On les définit toujours avec la fonction express `app.use()` au début de la définition du server.

```
$ npm install cookie-parser
```

```
var express = require('express')
var app = express()
var cookieParser = require('cookie-parser')

// load the cookie-parsing middleware
app.use(cookieParser())
```

Un parser de cookie, miam!

```
$ npm install body-parser
```

API

```
var bodyParser = require('body-parser')
```

Un parser de body http

Liste non exhaustive de middleware existants : <http://expressjs.com/en/resources/middleware.html>

Express, le framework web pour NodeJS

Quelques middlewares

Un middleware d'authentification (très, très utilisé) :

```
$ npm install passport
```

Usage

Strategies

Passport uses the concept of strategies to authenticate requests. Strategies can range from verifying username and password credentials, delegated authentication using [OAuth](#) (for example, via [Facebook](#) or [Twitter](#)), or federated authentication using [OpenID](#).

Before authenticating requests, the strategy (or strategies) used by an application must be configured.

```
passport.use(new LocalStrategy(
  function(username, password, done) {
    User.findOne({ username: username }, function (err, user) {
      if (err) { return done(err); }
      if (!user) { return done(null, false); }
      if (!user.verifyPassword(password)) { return done(null, false); }
      return done(null, user);
    });
  }
));
```

Express, le framework web pour NodeJS

Le routing

Express, le framework web pour NodeJS

Les routing

Le routing (ou routage), est la façon dont une application répond à une demande client adressée à un nœud final spécifique, c'est-à-dire la combinaison d'une URL (un chemin), et d'une méthode HTTP (GET, POST...Etc).

Comme on a pu déjà le voir, la définition d'une route a la structure suivante:

app.METHOD(PATH, HANDLER)

- app est l'instance d'express
- METHOD est une méthode de demande HTTP (GET, POST...)
- PATH est un chemin sur le serveur
- HANDLER est la fonction de callback

Plus de détails sur le routage : <http://expressjs.com/en/guide/routing.html>

Express, le framework web pour NodeJS

Les routing

Exemple de routing et quatre méthodes HTTP : GET, POST, PUT, DELETE

```
// GET /
app.get('/', function (req, res) {
| res.send('Hello World!');
});

// POST /
app.post('/', function (req, res) {
| res.send('Got a POST request');
});

// PUT /user
app.put('/user', function (req, res) {
| res.send('Got a PUT request at /user');
});

// DELETE /user
app.delete('/user', function (req, res) {
| res.send('Got a DELETE request at /user');
});
```

Express, le framework web pour NodeJS

La gestion des erreurs

Express, le framework web pour NodeJS

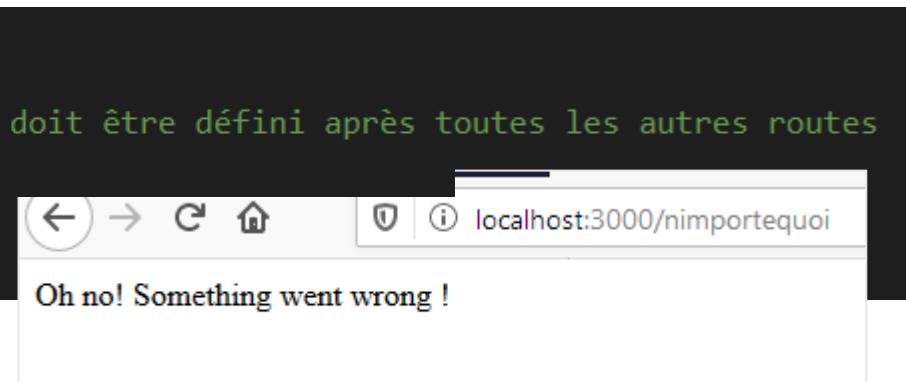
La gestion des erreurs - 404

Lorsque le serveur reçoit une requête pour laquelle il n'a pas de ressources, il renvoie une erreur HTTP 404.

En mode simple :

```
app.use(function (req, res, next) {  
    res.status(404).send("Sorry can't find that!")  
})
```

Souvent, on va vouloir renvoyer une page HTML personnalisée :



The screenshot shows a browser window with the URL `localhost:3000/nimportequoi`. The page content is a simple message: `Oh no! Something went wrong!`. This demonstrates how a custom 404 page can be rendered.

```
app.use(function (req, res, next) { // attention, cette fonction doit être défini après toutes les autres routes  
    res.status(404).render(generated404Template);  
})
```

Attention : cette fonction doit être définie en dernier car le serveur va tester toutes les routes déjà définies avant de se rabattre sur celle-là.

Express, le framework web pour NodeJS

La gestion des erreurs - 500

Lorsque quelque chose ne s'est pas bien passé (par exemple un appel à la base de données, ou la lecture d'un fichier), le serveur doit renvoyer une erreur 500.

Dans le cas d'une fonction synchrone, il suffit d'utiliser la fonction *throw* de javascript pour que le serveur renvoie une erreur :

```
app.get('/breakit', function (req, res) {
  | throw new Error('BROKEN') // Express will catch this on its own.
})
```

The screenshot shows a POSTMAN interface. At the top, it says "GET" and "http://localhost:3000/breakit". Below that, there are tabs for "Params", "Authorization", "Headers (7)", "Body", "Pre-request Script", "Tests", and "Settings". Under "Headers (7)", there is a table with one row: "Key" and "Value". In the "Body" tab, it says "Status: 500 Internal Server Error". Below that, there are tabs for "Pretty", "Raw", "Preview", "Visualize BETA", "HTML", and a copy icon. The "Pretty" tab shows the following HTML code:

```
1 <!DOCTYPE html>
2 <html lang="en">
3
4   <head>
5     <meta charset="utf-8">
6     <title>Error</title>
7   </head>
8   <body>
9     <h1>Error</h1>
10    <p>Something went wrong. Please try again later.</p>
11  </body>
12</html>
```

<https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Instructions/throw>

Express, le framework web pour NodeJS

La gestion des erreurs - 500

Dans le cas d'une fonction asynchrone, comme la lecture d'un fichier, par exemple ^^, on va utiliser un troisième paramètre du callback d'express, la fonction du middleware de routing *next()* :

```
// app.get('/nofile', function (req, res, next) {
//   fs.readFile('/file-does-not-exist', function (err, data) {
//     if (err) {
//       next(err) // Pass errors to Express.
//     } else {
//       res.send(data)
//     }
//   })
// })
```

The screenshot shows the Postman application interface. At the top, there is a header bar with 'GET http://localhost:3000/nofile'. Below it, the main window has tabs for 'Params', 'Authorization', 'Headers (7)', 'Body', 'Pre-request Script', 'Tests', and 'Settings'. Under 'Headers (7)', there is a table with one row containing 'Content-Type' and 'application/json'. In the 'Body' tab, there is a 'Raw' section with the following JSON:

```
{ "error": "Internal Server Error" }
```

At the bottom of the interface, the status bar displays 'Status: 500 Internal Server Error'.

<http://expressjs.com/en/guide/error-handling.html>

Express, le framework web pour NodeJS

TP

Express, le framework web pour NodeJS

TP : API d'écriture et de modification d'un fichier cities.json

Nous allons créer une API pour écrire, mettre à jour et supprimer des noms de villes contenues dans un fichier json *cities.json* qui aura la structure suivante :

```
{} cities.json ×
{} cities.json > ...
1  {
2    "cities": [
3      { "id": "042ffd34-d989-321c-ad06", "name": "Toulouse" },
4      { "id": "823ffd34-e789-321c-gf88", "name": "Albi" }
5    ]
6 }
```

1. Créer un fichier cities.json avec le contenu ci-dessus.
2. Créer une route GET avec le chemin */cities* qui retourne le contenu du fichier cities.json. Lorsque cette route est appelée, le serveur doit :
 1. Vérifier si un fichier cities.json existe sur le serveur
 2. Si oui, le serveur retourne le contenu du fichier
 3. Si non, le serveur retourne une erreur.
3. Mettre en place une route POST avec le chemin suivant */city* qui contient un body avec un nom de ville
 1. Lorsque cette route est appelée, le serveur doit :
 1. Vérifier si un fichier cities.json existe sur le serveur, si non, le créer.
 2. Vérifier que la nouvelle ville n'existe pas dans le fichier, sinon retourner une erreur 500
 3. Si elle n'existe pas, la nouvelle ville doit être ajouté dans la liste, avec un id (généré à la volée, unique, en utilisant la librairie uuid par exemple : <https://www.npmjs.com/package/uuid>)
4. Mettre en place une route PUT avec le chemin */city/:id* qui contient un body avec un id de ville existant et un nouveau nom. Par exemple :
`{"id": "042ffd34-d989-321c-ad06", "name": "Toulouse, la ville rose" }`
 1. De la même manière que précédemment, ajouter les contrôles nécessaires, et mettre à jour la ville correspondante
5. Ajouter une route DELETE avec le chemin */city/:id* qui supprime la ville dont l'id a été passé en paramètre
6. Dans le git de votre TP, mettre des screenshots de tous vos appels Postman
7. **Bonus** : modifier la route GET */cities* pour qu'elle renvoie un joli template HTML de toutes les villes (on pourrait même imaginer les positionner sur une carte ... ^^). Faire un screenshot du résultat et le mettre dans le git également.

Express, le framework web pour NodeJS

La sécurité

Express, le framework web pour NodeJS

La sécurité

Nous allons faire le tour des recommandations de sécurité. Cela mériterait un cours à part entière donc on ne va pas pouvoir rentrer dans tous les détails.

N'hésitez pas à aller voir les liens, et retrouvez **la checklist sécurité complète** à la fin de la section.

Si vous pensez un jour travailler avec des serveurs NodeJS/Express, gardez cette checklist dans un coin. Elle est absolument essentielle 😊

Express, le framework web pour NodeJS

La sécurité

Ne pas utiliser de version d'Express déprécié ou vulnérable

- Express 2.x et 3.x ne sont plus maintenus.
- Suivre les dernières versions vulnérables sur : <http://expressjs.com/en/advanced/security-updates.html>

Express, le framework web pour NodeJS

La sécurité

Utiliser TLS

- Sécuriser la connexion et les données en encryptant avant l'envoi au serveur pour éviter, entre autres, le sniffing et les attaques man-in-the-middle.
- Pour avoir un certificat TLS gratuit, utiliser [let's encrypt](#)

https://en.wikipedia.org/wiki/Transport_Layer_Security

Express, le framework web pour NodeJS

La sécurité

Utiliser le middleware Helmet

- Helmet protège votre application en rajoutant des headers HTTP contre des vulnérabilités bien connues
- En particulier, Helmet ajoute
 - Le header content-security-policy
 - Le header X-content-type-options pour éviter le sniffing MIME
 - Le header X-XSS-Protection pour mettre le filtre XSS

```
$ npm install --save helmet
```

Then to use it in your code:

```
// ...  
  
var helmet = require('helmet')  
app.use(helmet())  
  
// ...
```

<https://www.npmjs.com/package/helmet>

Express, le framework web pour NodeJS

La sécurité

Attention avec les cookies !

- Pour éviter des vulnérabilités liées au cookies, utiliser les middlewares express-session ou cookie-session
- NE PAS utiliser le nom du cookie par défaut de la session
- Mettre les options de sécurité (*secure*, pour envoyer uniquement en HTTPS, *expires* pour la date d'expiration...etc)

Un exemple avec le middleware cookie-session :

```
var session = require('cookie-session')
var express = require('express')
var app = express()

var expiryDate = new Date(Date.now() + 60 * 60 * 1000) // 1 hour
app.use(session({
  name: 'session',
  keys: ['key1', 'key2'],
  cookie: {
    secure: true,
    httpOnly: true,
    domain: 'example.com',
    path: 'foo/bar',
    expires: expiryDate
  }
}))
```

<https://www.npmjs.com/package/express-session>
<https://www.npmjs.com/package/cookie-session>

Express, le framework web pour NodeJS

La sécurité

Prévenir les attaques brute force :

- Bloquer les autorisations en se basant sur :
 - Le nombre de requêtes consécutives en erreur du même utilisateur et de la même IP
 - Le nombre de requête en erreur à partir d'une même IP sur une longue période de temps. Par exemple, bloquer une adresse IP si il y a eu 100 requêtes en erreur sur une journée.
- Rate-limiter-flexible est une librairie pour mettre en place ce type de blocage.

<https://github.com/animir/node-rate-limiter-flexible/wiki/Overall-example#login-endpoint-protection>

Express, le framework web pour NodeJS

La sécurité

S'assurer que toutes les librairies utilisées sont sécurisées :

- Utiliser *npm audit* pour vérifier la vulnérabilité de ces paquets npm
- Utiliser snyk

<https://docs.npmjs.com/cli/audit>
<https://snyk.io/>

Express, le framework web pour NodeJS

La sécurité

Prévenir les vulnérabilités connus :

- Suivre les mises à jour de la security checklist de nodeJS : <https://blog.risingstack.com/node-js-security-checklist/>
- Suivre les recommandations de la fondation OWASP : <https://owasp.org/www-project-web-security-testing-guide/>

Express, le framework web pour NodeJS

La sécurité

Comment sécuriser un serveur Express en production? (cf <http://expressjs.com/en/advanced/best-practice-security.html>).

Checklist de sécurité :

- Ne pas utiliser de version d'Express déprécié ou vulnérable
- Utiliser TLS
- Utiliser le middleware Helmet
- Utiliser les cookies de manière sécurisé
- Prévenir les attaques brute force
- S'assurer que les toutes les librairies utilisées sont sécurisées
- Prévenir les vulnérabilités connus

<http://expressjs.com/en/guide/error-handling.html>