

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.

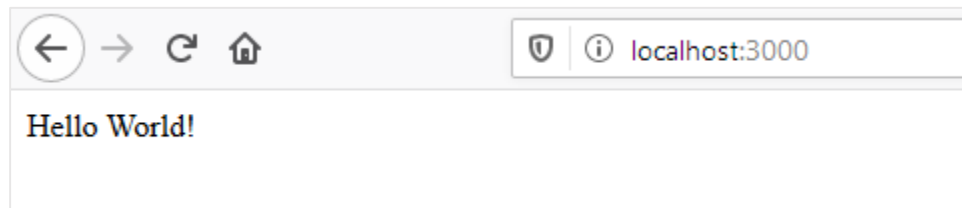
<https://expressjs.com/en/starter/installing.html>

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
- Icones
- Fichiers CSS

Ces ressources ne sont pas modifiées 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

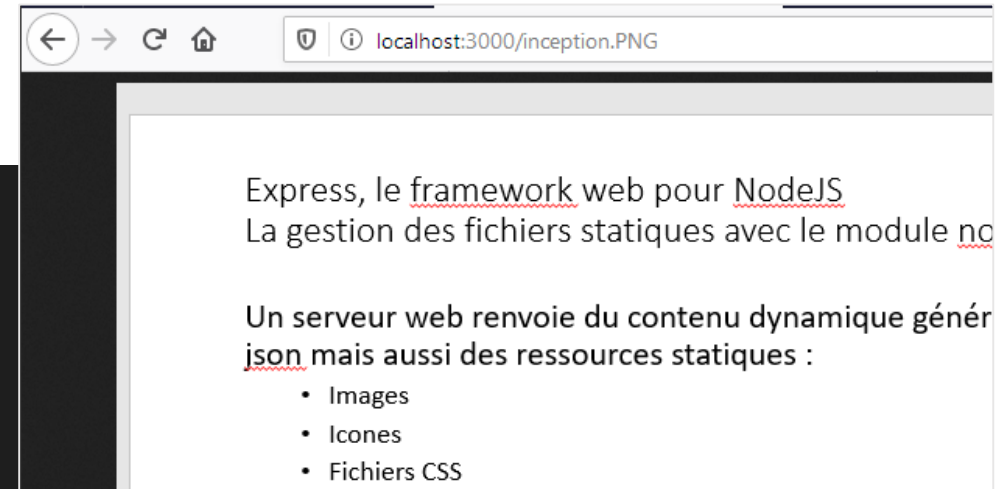
La gestion des fichiers statiques avec le module nodeJS path

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 requetes 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>

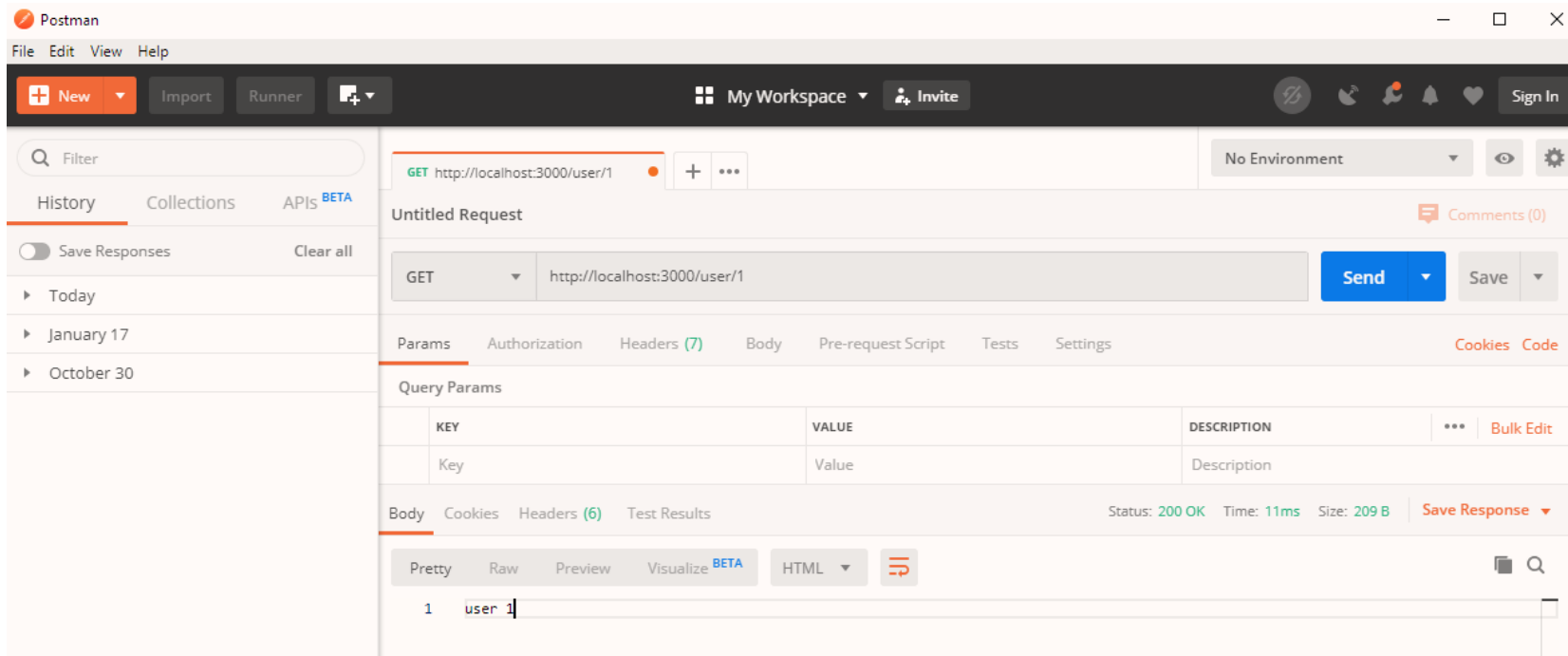
Express, le framework web pour NodeJS

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):



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

Mini TP

NodeJS, le moteur Javascript

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
- Dans le git de votre TP, mettre des screenshots de tous vos appels 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

Les templates

Express, le framework web pour NodeJS

Les templates

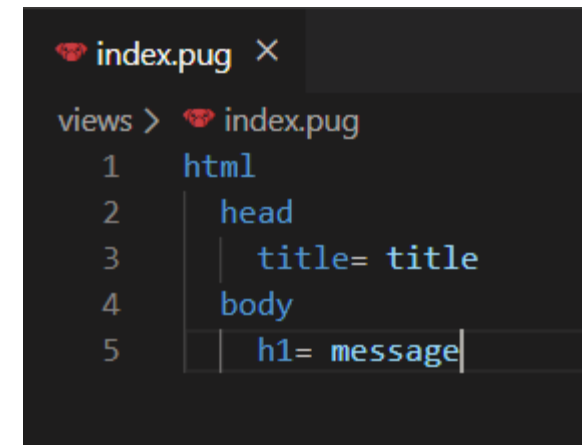
On peut continuer à utiliser le moteur de template pug avec express, mais :

- 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.

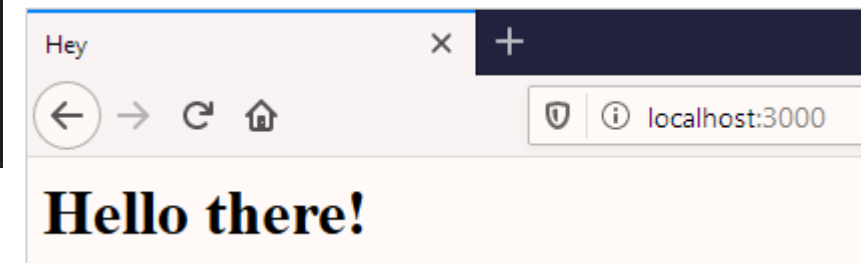
Exemple d'une route `/` qui renvoie un template `index.pug` :

```
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>



```
index.pug ×
views > index.pug
1  html
2    head
3      title= title
4    body
5      h1= message
```



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 taches 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>

Express, le framework web pour NodeJS

Quelques middlewares

Exemple d'un middleware qui va logger toutes les requetes 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

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

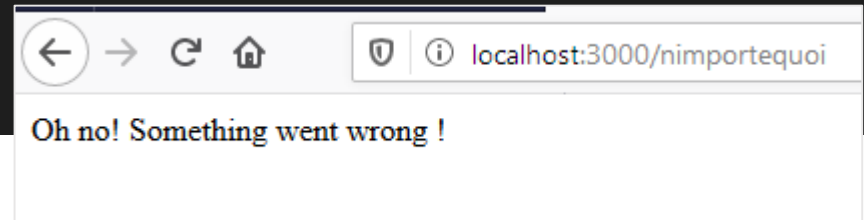
Lorsque le serveur reçoit une requête pour laquelle il n'a pas de ressources, il renvoie une erreur HTTP 404.

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 customisée :

```
const compiledFunction = pug.compileFile('template404.pug'); // compilation du template  
  
app.use(function (req, res, next) { // attention, cette fonction doit être défini après toutes les autres routes  
  const generated404Template = compiledFunction();  
  res.status(404).send(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.

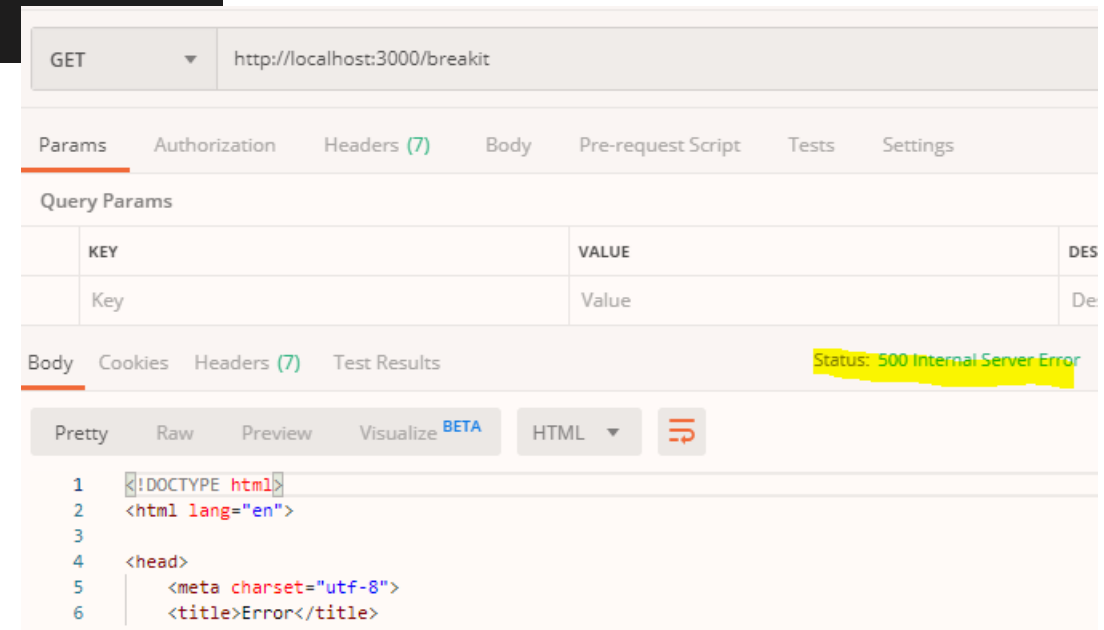
Express, le framework web pour NodeJS

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.  
})
```



<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 a REST client interface with a request to `http://localhost:3000/nofile` using the `GET` method. The response status is `500 Internal Server Error`, with a time of `6ms` and a size of `1.46 KB`. The response body is displayed in HTML format, showing the following structure:

```
1 <!DOCTYPE html>  
2 <html lang="en">  
3  
4 <head>  
5   <meta charset="utf-8">  
6   <title>Error</title>
```

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

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://letsencrypt.org/)

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 requetes 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 requetes 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 toutes les librairies utilisées sont sécurisées
- Prévenir les vulnérabilités connus

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

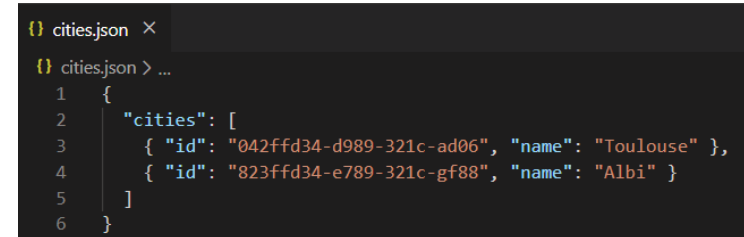
Express, le framework web pour NodeJS

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 x
{} 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 ajoutée 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.