NodeJS

Campus Academy

Alan Piron-Lafleur

Rappel ES6 et NPM

Let et var

Ou encore nommé ES6 ou ES2015.

ES6 apporte un lot d'améliorations par rapport à ES5. Voici ce que vous devez savoir, avant tout :

On peut déclarer une variable en utilisant :

- var
- let
- const

La différence entre *let* et *var* ?

C'est la portée !
Avec **let**, la variable est **locale** (accessible uniquement dans le bloc où elle a été déclarée).

Avec **var**, la variable est **globale**.

Let et var

```
var i = 62;
for(var i = 0; i < 10; i++){
    console.log(i);
}

console.log("Et au final : " + i);
```

Alors ? Quel résultat ?

Bonne réponse : 10

Let et var

```
let i = 62;
for(let i = 0; i < 10; i++){
    console.log(i);
}

console.log("Et au final : " + i);
```

Et là ? Quel résultat ?

Bonne réponse : 62

Fonctions fléchées

```
// es5
var addition = function(x){
    return x + 1;
}

// es6
let addition = (x) => {
    return x + 1;
}
```

A votre avis, peut-on faire encore plus simple ?

Fonctions fléchées

```
// es5
var addition = function(x){
    return x + 1;
}

// avec un seul argument, j'écris directement sans parenthèses
let addition = x => {
    return x + 1;
}
```

A votre avis, peut-on faire encore plus simple ?

Fonctions fléchées

```
// es5
var addition = function(x){
    return x + 1;
}

// avec une seule instruction, j'écris sans les accolades
let addition = x => x + 1;
```

A votre avis, peut-on faire encore plus simple ?

Non, quand même pas …

Les template Strings

Vous savez que l'on peut écrire une chaîne de caractères entre guillemets simples ou doubles.

Une nouveauté ES6 nous permet d'utiliser les backquotes pour mettre des variables directement dans une chaîne de caractère… donc pour éviter de concaténer.

```
//es5
let prenom = "Alan";
let message = "Bonjour, je
m'appelle " + prenom + " et
ça va, ça va !";

console.log(message);
```

```
// es6
let prenom = "Alan";
let message = `Bonjour, je
m'appelle ${prenom} et ça
va, ça va !`;
console.log(message);
```

Les timers

Rappel Javascript avec l'utilisation des timers, applicable à NodeJS si l'on veut lancer une fonction après un temps donné. A utiliser au besoin !

```
//La méthode sera executée dans 3s
setTimeout(function setTimeout() {
   return console.log('Hello Geek!');
}, 3000);

console.log("Toujours le dernier .. :(");
```

Les classes

Et le meilleur pour la fin : les class.

```
class Humain{

    constructor(nom){
        this.nom = nom;
    }

    parle(){
        console.log(`Salut, je suis ${this.nom}`);
    }

}
```

😯Attend, il n'y avait pas de class en ES5 ?

Eh bah non. On utilisait les function.

Npm

Le seul l'unique : le gestionnaire officiel de paquets de Node.js

Pour faire simple : plus d'un million de paquets sont disponibles gratuitement.

Un paquet, c'est un dossier dans lequel nous retrouvons tout plein de fichiers qui servent nos applications : uploader des images, hasher un password, récupérer une requête post, …. on ne ré-invente pas la roue.

Egalement, NPM gère la mise à jour d'un paquet avec npm update.

Un paquet qui doit être mis à jour ? Un simple npm update et zoup, c'est fait… et en plus, NPM assure la compatibilité avec notre application !

Configuration de Node

Installer Node.JS

Accédez à NodeJS.org pour télécharger puis installer la dernière version **LTS** de Node.

Laissez tout par défaut.

Cela installe également Node Package Manager ou npm, outil précieux pour l'installation des packages nécessaires à la création de vos projets.

L’IDE

Libre à vous de prendre l’IDE de votre choix.
Personnellement, c’est Intellij.

D’autres préfèrent VSCode ou Cloud9, ...

Le projet

Nous allons nous entraîner avec un projet récupéré en ligne et libre d'utilisation : un magasin en ligne qui vend des trucs, tout plein de truc.

Le frontend peut se faire de plusieurs manières : Angular, React, VueJS, etc …

Etant donné qu'il s'agit ici d'un cours NodeJS, le frontend vous est donné pour que nous nous concentrions sur le backend.

Créez un répertoire : nodeJS-formation (par exemple).
Créez un répertoire nodeJSFronts par exemple (ne le mettez pas nodeJS-formation) pour récupérer le frontend avec :

```
git clone https://github.com/ExtraCode/nodeJS-formation.git
```

Copier/coller le dossier frontend-cours-nodejs dans votre répertoire nodeJS-formation et renommer le "frontend". Gardez le répertoire nodeJSFronts, il nous servira plus tard.

Mise en place du frontend

Une fois le dossier frontend créé, faites un :

```
cd frontend
npm install
npm run start
```

Cela installera toutes les dépendances requises par l'application front-end, et lancera le serveur de développement. Désormais, si vous accédez à http://localhost:4200, vous devriez voir l'interface frontend.

Veillez à laisser tourner un terminal avec npm run start de lancé.

Mise en place du backend

Créez un dossier “backend” puis placez vous dedans en invite de commande.

Lancez : `npm init`

Laissez tout par défaut **sauf** entry point. Mettez : **server.js** au lieu d’index.js.

Pour versionner votre code, faites un `git init`

Créez ensuite un fichier .gitignore dans le dossier backend et renseignez simplement : node_modules dedans.

Enfin, créez le fichier server.js. Il est vide, mais cela ne va pas durer longtemps.

Création du serveur NODE

Dans le fichier server.js, on peut écrire du Javascript simple.
Essayez : `console.log("Ah oui, en effet");`

Puis faites un node server dans votre terminal.

Mais nous allons utiliser nodeJS pour faire des choses un peu plus … puissantes !

(enlevez le console.log).
Nous allons importer le module HTTP de node pour créer un serveur :

```
const http = require('http');
const server = http.createServer((req, res) => {
    res.end('Voilà la réponse du serveur !');
});
```

Retourner une réponse

Chaque appel vers ce serveur déclenchera le callback de createServer.

Dans cet exemple, nous utiliserons la méthode `end` de la réponse pour renvoyer un réponse de type String.

Vous l'avez peut être remarqué : nous utilisons require et non import pour charger le module HTTP.

C'est parce Node utilise un système de module CommonJS, donc pour importer le contenu d'un module JS, on utilise le mot-clé "require". L'utilité, c'est que l'on pourra importer des modules de base de Node sans spécifier de chemin exact du fichier. Node saura où trouver le module lui même !

Configurer un port

Pour l'instant, notre serveur existe mais il n'écoute sur aucun port. Configurons le :

server.listen(process.env.PORT || 3000);

Votre serveur écoutera ainsi :

- soit la variable d'environnement du port grâce à process.env.PORT : si la plateforme de déploiement propose un port par défaut, c'est celui-ci qu'on écoutera ;
- soit le port 3000, ce qui nous servira dans le cas de notre plateforme de développement.

Il ne reste plus qu'à démarrer le serveur : `node server`

Rien (de visuel) ne se passe : votre serveur est en écoute.

Rdv sur http://localhost:3000/ pour tester !

Nodemon

Modifiez la ligne 4 : `res.end('Voilà la réponse du serveur, eh oui !');`
Et relancez votre page http://localhost:3000/… rien ne change.
Il faut relancer le serveur à chaque fois que l'on fait une modification dessus.
Pénible… mais c'est comme ça.

Sinon, il vous suffit d'installer nodemon : `npm install -g nodemon`
Et de faire, dorénavant, un `nodemon server` pour lancer votre serveur.
Ce brave nodemon redémarrera automatiquement le serveur à chaque modification.

Tester du code avec le REPL

Node.js est livré avec REPL (Read Eval Print Loop).

Ce petit utilitaire permet de tester du JavaScript. Très utile pour déboguer du code et comprendre certains problèmes.

Pour y accéder, il suffit de taper dans le terminal : node

A partir de là, on peut taper du JS (valide) pour le tester.

```
Tapez x = 10
puis x - 5
Cela affichera 5… youpi.
```

Le REPL est plus utilisé pour copier/coller des gros blocs de logique à tester.

Mise en place d’Express

Node et Express

Quelle différence entre Node et Express ?

Node : il permet d'écrire toutes nos actions côté serveur, en JS.
On écrira la logique métier, l'écriture de nos données en base, on gérera la sécurité avec Node… etc

Express, c'est le framework qui repose sur Node et qui permet de faciliter l'écriture… plutôt que d'écrire en Node pur.

Installer express

Je vous le disais, coder des serveurs web en Node pur, c'est possible… mais c'est long et laborieux, vraiment.

Il faudrait analyser manuellement chaque requête entrante et faire un traitement. Express va nous simplifier la tâche, vous allez voir !

Via votre terminal, assurez vous d'être dans le dossier backend et faites l'installation d'express :

```
npm install express
```

Créer l’application avec app.js

Créons un fichier app.js dans notre dossier backend.

Importons express :

const express = ?? que mettriez-vous ici ?

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

Ensuite, nous devons créer notre app :

```
const app = express();
```

et l’exporter pour qu’elle soit utilisable partout, dans tous nos fichiers :

```
module.exports = app;
```

Global objects

❓Une petite seconde…

```
module.exports = app;
```

Il vient d'où lui ?

“module” est ce qu'on appelle un “global objects”. Pas besoin de le require, il existe déjà, partout dans le code.
Il en existe plusieurs, évidemment.

Lier notre server Node avec notre app Express

Retour sur notre server.js

Plutôt que de renvoyer un message, on voudra utiliser notre application Express lorsqu'une requête arrive sur le serveur.

Pour ça, il faut importer notre app qui se trouve dans app.js directement dans notre fichier server.js

Comment peut-on faire selon vous ?

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

Il faut également préciser à express le port utilisé (oui, lui aussi a besoin de le savoir, tout comme le serveur Node) :

```
app.set('port', process.env.PORT || 3000);
```

Lier notre server Node avec notre app Express

Ensuite, il faut que le server nodeJS utilise notre app :

```
const server = http.createServer(app);
```

Lier notre server Node avec notre app Express

Testez notre serveur en allant sur votre navigateur.

Une erreur 404 est générée.. sauriez vous dire pourquoi ? (réponse toute bête)

C'est parce que notre application n'a aucun moyen de répondre… c'est tout 😁

On va l'aider :

```
app.use((req, res) => {
   res.json({ message: "Votre requête a bien été reçue !" });
});
```

Testez en allant sur votre navigateur… la réponse est renvoyée.
Testez en ajoutant /test (ou tout autre route) à l'url… la réponse est renvoyée car aucune route n'est paramétrée. C'est la même réponse, dans tous les cas.

Les middlewares

Un middleware, c’est une fonction dans une application express qui reçoit une requête et une réponse.

Est-ce que l’on a déjà créé un middleware dans notre app ?

Oui, regardez app.use <<<--- c’est un middleware !

Une app Express complète se composera de plusieurs middleware. Pour l’instant, nous avons juste besoin de savoir qu’un middleware reçoit une requête, une réponse ET la méthode next.. pour passer au middleware suivant. Nous allons tester cela.

Les middlewares

Créez ces 3 middlewares et lancez votre navigateur :

```
app.use((req, res, next) => {
   console.log("Réponse reçue");
});

app.use((req, res, next) => {
   res.json({ message: "Votre requête a bien été reçue !" });
});

app.use((req, res, next) => {
   console.log("Réponse envoyée");
});
```

Les middlewares

Le code précédent crash.. en effet, il existe 3 middlewares mais aucun n'envoie vers l'autre, la route est donc bloquée pour Express.

Ajoutez next(); après le console.log() du premier middleware et après la réponse json du second => ça fonctionne (le code sera exécuté deux fois, c'est normal).

next(); sert donc à passer d'un middleware à l'autre.

server.js amélioré

Je vous passe le code d’un fichier server.js amélioré, et on regarde cela ensemble.

- La fonction errorHandler recherche les différentes erreurs et les gère de manière appropriée. Elle est ensuite enregistrée dans le serveur ;
- Un écouteur d'évènements est également enregistré, consignant le port ou le canal nommé sur lequel le serveur s'exécute dans la console.
- La fonction normalizePort renvoie un port valide, qu'il soit fourni sous la forme d'un numéro ou d'une chaîne ;

server.js amélioré

```
const http = require('http');
const app = require('./app');

const normalizePort = val => {
    const port = parseInt(val, 10);

    if (isNaN(port)) {
        return val;
    }
    if (port >= 0) {
        return port;
    }
    return false;
};
const port = normalizePort(process.env.PORT || '3000');
```

Traiter les requêtes GET et POST

Notre première route

Rdv sur la partie front : http://localhost:4200/part-one.

Ouvrons la console pour voir l'erreur : *Access to XMLHttpRequest at 'http://localhost:3000/api/stuff'*

Il nous est clairement indiqué que la route /api/stuff de notre backend est inaccessible. Normal, nous ne l'avons pas créée.😉

Notre première route

Dans app.js, enlevez les middlewares et créez en un nouveau :

```
app.use('/api/stuff', (req, res, next) => {
    // ici, nous mettrons un tableau d'objet, faisons le ensemble
});
```

Rdv sur localhost:3000/api/stuff pour voir la réponse de notre serveur.

L’erreur CORS

Retour sur notre frontend via le navigateur.. ça ne fonctionne pas mieux.. erreur de CORS.

Le “Cross Origin Resource Sharing”, c’est un système de sécurité qui, par défaut, bloque les appels HTTP entre des serveurs différents, ce qui empêche donc les requêtes malveillantes d'accéder à des ressources sensibles. Dans notre cas, nous avons deux origines : localhost:3000 et localhost:4200

Nous, nous voulons que notre API soit accessible par tout le monde, de n’importe où.

L’erreur CORS

Elle se gère dans app.js en créant un middleware global (sans route) :

```
app.use((req, res, next) => {
  res.setHeader('Access-Control-Allow-Origin', '*');
  res.setHeader('Access-Control-Allow-Headers', 'Origin,
X-Requested-With, Content, Accept, Content-Type, Authorization');
  res.setHeader('Access-Control-Allow-Methods', 'GET, POST, PUT,
DELETE, PATCH, OPTIONS');
  next();
});
```

Retour sur le frontend, il fonctionnera !

L’erreur CORS

Le premier header permet d’accéder à notre API depuis n’importe quelle origine.

Le second demande à ceux qui font les requêtes vers notre API d’ajouter les headers Origin, X-Requested-With, etc…

Le dernier permet d’accepter les requêtes de type get, post, put, etc…

Poster des données

Pour gérer la requête POST venant de l'application front-end, on a besoin d'en extraire le corps JSON.
Express nous met a dispo un middleware très simple :

```
app.use(express.json());
```

❗ (Dans des versions précédentes, on utilisait le module BodyParser à la place)

Avec ceci, Express prend toutes les requêtes qui ont comme Content-Type application/json et met à disposition leur body directement sur l'objet.
Créons maintenant le middleware pour le POST sur la route /api/stuff

```
app.post('/api/stuff', (req, res, next) => {
  console.log(req.body);
  res.status(201).json({
    message: 'Objet créé !'
  });
});
```

Poster des données

On réécrit le middleware pour le GET :

`app.get` au lieu de app.use

Testons le de remplir le formulaire du menu “Vendre un objet” : ça fonctionne (enfin.. on récupère les données côté serveur) !

Stateless ou pas stateless ?

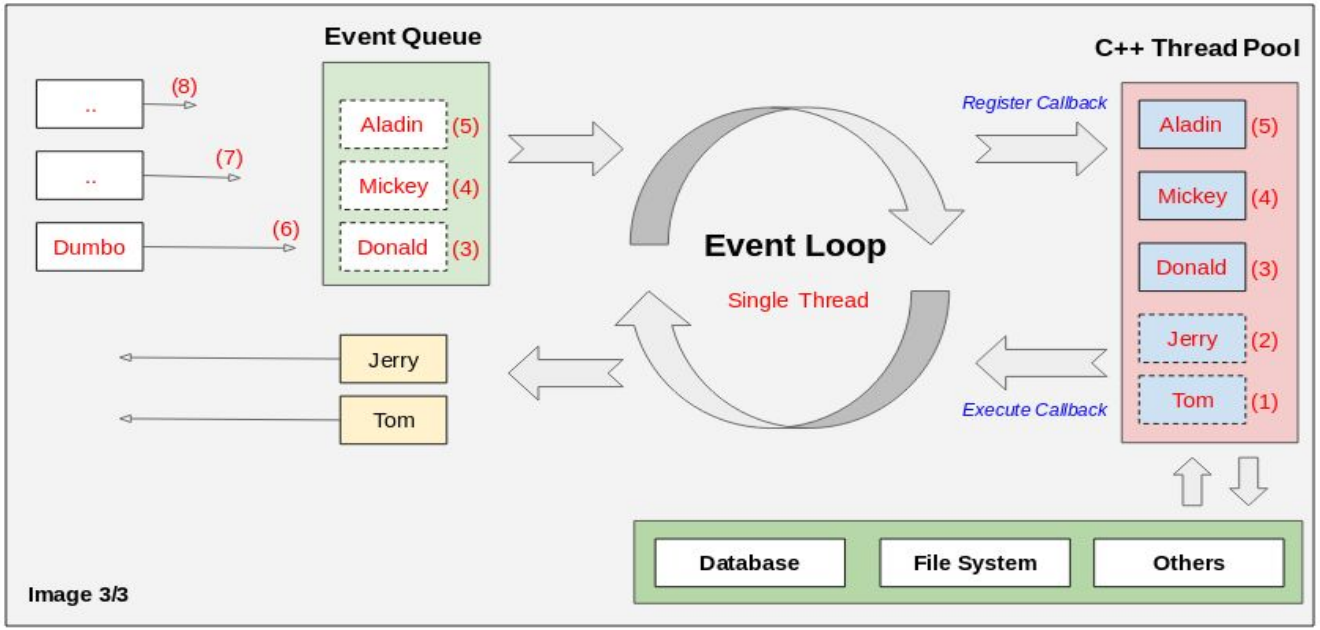
On dit qu'une application est stateless quand il n'y a pas d'informations stockées sur ce qui s'est passé avant. Chaque transaction est individuelle. Dans une application stateless, si la transaction est interrompue, on doit la reprendre dès le début.

Dans une application stateful, les données sont sauvegardées; c'est l'inverse : on reprend là où on s'est arrêté.

Alors ? Stateless ou pas notre application ?

L’event LOOP

Un autre point vocabulaire : l’event LOOP… Rien de mieux qu’une illustration.



L’event LOOP

NodeJS est une application qui utilise ce qu’on appelle le multi-thread pour effectuer des tâches en même temps.

Chaque demande est traitée comme un événement. Elles sont placées dans le Event Queue.
Les events font donc la queue… puis sont transmis au C++ Thread Pool.
What ? C++ ? Eh oui ! NodeJS, c’est écrit en grand partie en C++ !

Le Thread Pool enregistre les callback des events appelés, puis les déclenchent et renvoie à l’event LOOP de NodeJS la réponse…. qui est servie au client.

QUIZ 1

Le CRUD

Base de donnée SQL vs NoSQL

Les BDD en SQL sont des bases de données relationnelles.
On y retrouve des tables qui respectent des schémas stricts, par exemple :
User(id, first_name, last_name, email).

Les relations entre les différentes tables sont très importantes. Ainsi, on pourra trouver une table Commande avec un champ (id_user) pour lier User et Commande.

Ces BDD sont parfaites pour des données relationnelles ayant des définitions (plusieurs champs) fortes. Cependant, dans le cas de projets plus petits dont on ne sait pas encore leur taille finale, on préfèrera du NoSQL.
Car il est très difficile d'agrandir une base de données SQL au-delà d'une certaine limite.

C'est là où le NoSQL montre son utilisé, en particulier MongoBD.

MongoDB

C’est une BDD NoSQL où les données sont stockées sous format JSON. Il n’y a pas de schéma strict de données. Autrement dit, on peut faire ce qu’on veut, où on veut. Il n’y a pas non plus de relation concrète entre les différentes données.

Les avantages sont : évolutivité et flexibilité. Sur le site officiel de MogoDB, on lit : “construit pour des personnes qui construisent des applications Internet et des applications métier qui ont besoin d'évoluer rapidement et de grandir élégamment”.

MongoDB

Rendez-vous sur le site “MongoDB” et cliquez sur “Essayez gratuitement”.

Créez un compte, validez l’email, et choisissez un serveur gratuit.
Laissez les options par défaut, cela créera un cluster chez AWS.

Vous arrivez ensuite sur le Security Quickstart. Choisissez un username et un password, c’est l’utilisateur qui accèdera à la BDD. Faites “Create User”.

Cliquez ensuite sur “Network Access” (menu de gauche) -> add IP Address -> allow access from anywhere -> confirm.

Connectons notre API à MongoDB Atlas

Dans l'onglet Database, cliquez sur "Connect" -> "Connect your application" puis copiez la String de connexion donnée, elle commence par "mongodb".

De retour sur l'api, nous allons installer l'ORM Mongoose. Il va faciliter les interactions avec notre MongoDB (valider les données, gérer les relations entre documents, communiquer directement avec notre BDD).

Le tout en une commande : `npm install mongoose`

Connectons notre API à MongoDB Atlas via Mongoose

Importez l'ORM mongoose dans votre app.js (vous savez faire !).

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

Puis, en dessous de la déclaration de la constante app, connectons notre API avec Mongoose (je vous donne le code). Le message "connexion réussie" devra apparaître :

```
mongoose.connect('votrechainedecaractere',
  { useNewUrlParser: true,
    useUnifiedTopology: true })
  .then(() => console.log('Connexion à MongoDB réussie !'))
  .catch(() => console.log('Connexion à MongoDB échouée !'));
```

Créer notre premier schéma

Nous allons créer un premier schéma que nous allons appeler Thing. Pour tout objet stocké dans notre application.
Créons un dossier models et un fichier Thing.js à l'intérieur.

Thing.js a besoin :

- d'importer Mongoose
- de déclarer le schéma
- d'exporter le modèle

On le fait ensemble.

Créer notre premier schéma

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

const thingSchema = mongoose.Schema({
  title: { type: String, required: true },
  description: { type: String, required: true },
  imageUrl: { type: String, required: true },
  userId: { type: String, required: true },
  price: { type: Number, required: true },
});

module.exports = mongoose.model('Thing', thingSchema);
```

Enregistrer en BDD

Nous devons importer le nouveau modèle dans app.js pour l'utiliser :

```
const Thing = require('./models/Thing');
```

puis remplacer la logique de la route POST pour récupérer les données et les enregistrer, on le fait ensemble.

Enregistrer en BDD

```
app.post('/api/stuff', (req, res, next) => {

  // regardons ce qu'il a dans la requête

  console.log(req.body);
  return false;

});
```

Enregistrer en BDD

```
app.post('/api/stuff', (req, res, next) => {

  // Pour créer un nouvel objet, on fait :
  const thing = new Thing({

    // on pourrait faire :
    title: req.body.title,
    etc

    // on préfèrera faire :
    …req.body

  });
```

Enregistrer en BDD

```
app.post('/api/stuff', (req, res, next) => {

  // On doit enlever l'_id passé par le frontend car il sera
auto généré chez nous.
  delete req.body._id;
  const thing = new Thing({
    ...req.body
  });



});
```

Enregistrer en BDD

```
app.post('/api/stuff', (req, res, next) => {

   delete req.body._id;

   const thing = new Thing({
       ...req.body
   });
   thing.save()
       .then(() => res.status(201).json({ message: 'Objet
enregistré !'}))
       .catch(error => res.status(400).json({ error }));
});
```

Lire en BDD

Rappelez vous : notre frontend fait un GET sur notre api /api/stuff pour récupérer les objets.
Pour l'instant, nous lui renvoyons simplement nos deux objets créés à la main.

Allons lire en base de données, c'est très simple : je vous montre.

Lire en BDD

```
app.get('/api/stuff', (req, res, next) => {
   Thing.find()
        .then(things => res.status(200).json(things))
        .catch(error => res.status(400).json({ error }));
});
```

Lire un objet précis en BDD

Sur le frontend, quand on clic sur un objet, on voudrait voir ses infos.

Encore une fois, c'est très simple… je vous montre la route, ensuite vous essaierez de compléter par vous même !

Essayez de faire fonctionner la route.

But : quand on clic sur un objet, on veut voir ses infos.

Indication :

- la doc est là : mongoose doc query
- Il faut récupérer l'objet en base grâce à son id
- Pour récupérer l'id dans la requête, c'est req.params.id

Lire un objet précis en BDD

```
app.get('/api/stuff/:id', (req, res, next) => {
    Thing.findById(req.params.id)
        .then(thing => res.status(200).json(thing))
        .catch(error => res.status(400).json({ error }));
});
```

Modifier un objet précis en BDD

Pour la modification des objets, c’est avec PUT et UPDATEONE ! Je vous montre.

Modifier un objet précis en BDD

```
app.put('/api/stuff/:id', (req, res, next) => {
   Thing.updateOne({ _id: req.params.id }, { ...req.body, _id: req.params.id })
        .then(() => res.status(200).json({ message: 'Objet modifié !'}))
        .catch(error => res.status(400).json({ error }));
});
```

Voir prochaine diapo pour l'explication d'updateOne

Modifier un objet précis en BDD

Arrêtons nous sur :

```
Thing.updateOne(
    { _id: req.params.id },
    { ...req.body, _id: req.params.id }
)
```

Ligne 2 : on récupère le Thing en fonction de l'id que l'on récupère dans les paramètres de la requête

Ligne 3 : on met les nouvelles infos de l'objet à la place des anciennes puis on précise l'ID car celui-ci n'est pas envoyé par le frontend.

Supprimer un objet précis en BDD

Enfin, pour supprimer un objet, c’est avec DELETE et DELETEONE.
Comme d’habitude, je vous montre (la doc).

Supprimer un objet précis en BDD

```
app.delete('/api/stuff/:id', (req, res, next) => {
   Thing.deleteOne({ _id: req.params.id })
       .then(() => res.status(200).json({ message: 'Objet supprimé !'}))
       .catch(error => res.status(400).json({ error }));
});
```

TP 1

Le modèle MVC

Optimiser notre backend

Selon vous, qu'est-ce qui ne va pas trop dans notre App.js ?

Il y a TOUT à l'intérieur : la logique globale, la logique de routing, etc etc..

Nous allons donc créer un dossier routes dans le dossier backend puis un fichier stuff.js pour y placer la logique de nos routes stuff :

```
const express = require('express');
const router = express.Router();
module.exports = router;
```

C'est notre routeur express qui va maintenant gérer nos routes ! 😍

Optimiser notre backend

Couper / coller toutes nos routes de app.js vers stuff.js

Il faut remplacer toutes les occurrences de **app** par **router** car les routes sont maintenant enregistrées par celui-ci.

❗ Bien laisser l'export du router à la fin du fichier.

Egalement, il faudra mettre : `const Thing = require('./models/Thing');` de **app.js** dans **stuff.js** ❗ Il faudra changer le chemin ❗

Chaque route commençant par /api/stuff, nous allons pouvoir alléger notre routeur.

Supprimons /api/stuff de chaque route.

/api/stuff ⇒ /

/api/stuff/:id => /:id

Des commentaires

Il est temps de mettre des commentaires à nos routes.

```
// GET
// Liste tous les objets
router.get('/', (req, res, next) => {
.....
// GET
// Liste les informations d'un objet précis
.....
// POST
// Ajoute l'objet en base
```

etc etc

Importer le routeur dans app.js

Rien de plus simple :

`const stuffRoutes = require('./routes/stuff');` ← en haut du fichier, avec les autres require

`app.use('/api/stuff', stuffRoutes);` ← après les headers, sinon, on aura l'erreur CORS

Testez le frontend, ça fonctionnera tout pareil !

Optimisons encore notre backend

Dans MVC (Modèle / Vue / Controller) :

- nous avons fait le M
- nous avons fait le V (enfin, c'est la team des frontmen qui l'a fait)
- nous avons fait le C.. ah non, ça, on l'a pas fait.

Pour le moment, toute la logique métier (Create, Read, Update, Delete) de notre backend se situe… dans le routeur (aie 😱).

Créons de ce pas un dossier **controllers** dans notre dossier backend puis créons y un **stuff.js**

Créer un controller

Il suffira de couper / coller la logique métier (exemple :)

```
// GET
// Liste tous les objets
router.get( path: '/', handlers: (req : … , res : Response<ResBody, Locals> , next : NextFunction ) => {
    Thing.find() …
        .then( onfulfilled: things => res.status( code: 200).json(things)) …
        .catch(error => res.status( code: 400).json( body: { error }));
});
```

Créer un controller

Et de la glisser coller dans controllers/stuff.js :

```
exports.getAllThings = iciJeColleLaLogiqueMetier
```

Ce qui donne :

```
exports.getAllThings =  (req, res, next) => {
    Thing.find()
        .then(things => res.status(200).json(things))
        .catch(error => res.status(400).json({ error }));
}
```

Créer un controller

Faites pareil pour toutes les routes.
Nommez vos fonctions intelligemment (getAllThings, getOneThing, createThing, etc).

Ensuite, dans routes/stuff.js, il suffira de récupérer le contrôleur :

```
const stuffCtrl = require('../controllers/stuff');
```

puis d'utiliser les fonctions :

```
router.get('/', stuffCtrl.getAllStuff);
```

❗ Il faut que controllers/stuff importe Thing.
On peut supprimer l'import de Thing du routeur, plus besoin !

Créer un controller

Testez votre frontend, tout doit fonctionner.

❗ Testez l'ajout surtout. S'il ne fonctionne pas, c'est certainement parce que vous avez mis le code qui permet de récupérer les données postées :
(`app.use(express.json());`)

après le code qui utilise ces données (`app.use('/api/stuff', stuffRoutes);`)

Notre code fait strictement la même chose.
Mais nous avons **grandement** gagné en lisibilité (c'est ça le modèle MVC 😎).

Sécuriser notre API

Bienvenue les utilisateurs

Dans les chapitres suivants, nous implémenterons l'authentification par e-mail et mot de passe pour notre API. Cela implique de stocker des mots de passe utilisateur dans notre base de données.

Créer un model User avec les champs suivants :

email (String, requis)

password(String, requis)

Créer un controller

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

const userSchema = mongoose.Schema({
    email: { type: String, required: true },
    password: { type: String, required: true }
});

module.exports = mongoose.model('User', userSchema);
```

Un peu de sécurité

Nous voulons éviter qu'un utilisateur puisse s'enregistrer avec le même email :

```
email: { type: String, required: true,  unique: true }
```

Si un utilisateur avec la même adresse mail tente de s'enregistrer, il aura une erreur… vraiment pas très jolie.

On utilisera donc un petit plugin dans notre modèle :

```
npm install mongoose-unique-validator
```

Un peu de sécurité

Il suffit d'importer le module :

```
const uniqueValidator = require('mongoose-unique-validator');
```

et de l'utiliser, après la const userSchema :

….

```
userSchema.plugin(uniqueValidator);
```

S’inscrire et se connecter

Avant de passer à la logique métier “s’inscrire” et “se connecter”, nous voulons :

- créer deux nouvelles routes /api/auth/signup et /api/auth/login dans un fichier user.js
- créer un controller qui contiendra nos deux fonctions signup et login, appelées par les routes précédemment créées.
  Pour l’instant elles seront vides :

```
(req, res, next) => {
      // du vide
}
```

- enregistrer les routes au sein de notre app.js, bien évidemment

A vous de jouer !

S’inscrire et se connecter

Il fallait donc, dans controllers/user.js :

```
exports.signup = (req, res, next) => {

};

exports.login = (req, res, next) => {

};
```

S’inscrire et se connecter

Il fallait donc, dans routes/user.js :

```
const express = require('express');
const router = express.Router();
const userCtrl = require('../controllers/user');

router.post('/signup', userCtrl.signup);
router.post('/login', userCtrl.login);

module.exports = router;
```

S’inscrire et se connecter

Dans app.js :

```
const userRoutes = require('./routes/user');

app.use('/api/auth', userRoutes);
```

Gérer l’inscription

Nous avons besoin de crypter les mots de passe avec bcrypt : `npm install bcrypt`

Ensuite, il faut implémenter signup dans le controller, je vous montre.

Gérer l’inscription

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

exports.signup = (req, res, next) => {
   bcrypt.hash(req.body.password, 10)
       .then(hash => {
           const user = new User({
               email: req.body.email,
               password: hash
           });
           user.save()
               .then(() => res.status(201).json({ message : 'Utilisateur créé !'
}))
               .catch(error => res.status(400).json({ error }));
       })
       .catch(error => res.status(500).json({ error }))
}
```

Un vrai événement cette inscription

Nous pourrions avoir envie d'envoyer un mail lors de l'inscription d'un utilisateur.

Pour notre exemple, nous n'enverrons pas de mail car il faudrait configurer le serveur.. nous nous contenterons juste de console.log() pour simuler.

Où mettriez vous le code pour envoyer l'email après une inscription réussie ?

Normalement, vous avez pensé : "dans la fonction signUp()"...
Oui, mais que se passe-t-il si on veut envoyer un mail après … l'ajout d'un objet ?

Il faudrait remettre la logique d'envoi d'email dans la fonction createThing()... etc

Nous allons créer un écouteur d'événement qui sera appelé à chaque fois que l'on en aura besoin.

Créez un dossier events et un fichier sendMail.js, je vous montre la suite

Un vrai événement cette inscription

Dans sendMail.js

```
const event = require('events');

let evenement = new event.EventEmitter();

evenement.on('sendMail', function(params){
    console.log("J'envoi un mail à " + params.email);
});

module.exports = evenement;
```

Un vrai événement cette inscription

Dans le controller user.js :

```
const event = require('../events/sendMail');
```

On modifie le user.save() ainsi :

```
user.save()
    .then(() => {
        res.status(201).json({message: 'Utilisateur créé !'});
        event.emit('sendMail', {email: user.email});
    })
    .catch(error => res.status(400).json({ error }));
```

Gérer la connexion

Il suffit maintenant de récupérer l'email / mot de passe envoyé par le front puis de regarder en base de donnée si l'on retrouve un user avec cet email / ce mot de passe.

Je vous montre.

Gérer la connexion

```
exports.login = (req, res, next) => {
   User.findOne({ email: req.body.email })
       .then(user => {
           if (!user) {
               return res.status(401).json({ message: 'Paire login/mot de passe incorrecte'});
           }
           bcrypt.compare(req.body.password, user.password)
               .then(valid => {
                   if (!valid) {
                       return res.status(401).json({ message: 'Paire login/mot de passe incorrecte' });
                   }
                   res.status(200).json({
                       userId: user._id,
                       token: 'TOKEN'
                   });
               })
               .catch(error => res.status(500).json({ error }));
       })
       .catch(error => res.status(500).json({ error }));
};
```

Testez votre backend

Rdv sur la partie 3 en front pour enregistrer et se connecter avec un user, cela doit fonctionner !
Si ce n’est pas le cas, corrigez :)

Avant de passer à la partie suivante, il faut supprimer via le frontend, tous les articles en vente de la section Parties1+2 étant donné que les id_user des objets sont factices.

Créer le token d'authentification

L'objectif est que notre utilisateur ne se connecte qu'une seule fois à son compte.
Pour cela, au moment de se connecter, ils recevront un token.
Le frontend est déjà configuré pour qu'à chaque requête à notre API, le token soit envoyé dans le corps de celle-ci.

Utilisons le package jsonwebtoken :

```
npm install jsonwebtoken
```

Et importons le dans le controller user :

```
const jwt = require('jsonwebtoken');
```

Utilisons le ensuite dans la fonction login, je vous montre comment faire.

Créer le token d'authentification

A ligne 34, on modifie :

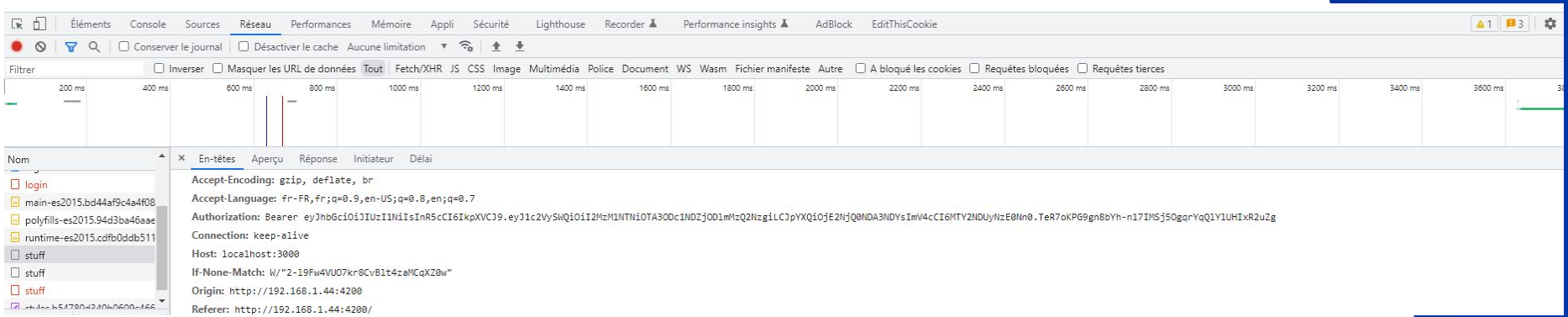
```
res.status(200).json({
    userId: user._id,
    token: jwt.sign(
        { userId: user._id },
        'RANDOM_TOKEN_SECRET',
        { expiresIn: '24h' }
    )
});
```

Note : le RANDOM_TOKEN_SECRET est à remplacer par une chaîne aléatoire beaucoup plus longue pour la production.

Créer le token d’authentification

Déconnectez vous si vous êtes connecté puis ouvrez l’inspecteur Chrome -> onglet Réseau.

Connectez vous puis regardez la première requête “stuff”. En scrollant ses informations, vous verrez le Authorization Bearer avec le token.



Sécuriser nos routes

Nous allons à présent créer le middleware qui va vérifier que l'utilisateur est bien connecté et transmettre les informations de connexion aux différentes méthodes qui vont gérer les requêtes… pour ajouter à un objet, l'user_id de l'utilisateur qui le crée.

Créons un dossier middlewares et un fichier auth.js à l'intérieur.

👉 *Le fait de checker qu'un user soit bien connecté et de récupérer l'user_id dans le token de la requête n'est pas un controller car il n'y a pas de route spécifique. Il s'agit d'un traitement que l'on voudra faire à chaque requête où l'utilisateur doit être connecté.*

Regardons ensemble le code de auth.js

Sécuriser nos routes

```
const jwt = require('jsonwebtoken');

module.exports = (req, res, next) => {
    try {
        const token = req.headers.authorization.split(' ')[1];
        const decodedToken = jwt.verify(token, 'RANDOM_TOKEN_SECRET');
        const userId = decodedToken.userId;
        req.auth = {
            userId: userId
        };
        next();
    } catch(error) {
        res.status(401).json({ error });
    }
};
```

Sécuriser nos routes

Enfin, nous devons appliquer le middleware à toutes nos routes sur lesquelles l'user doit être authentifié, autrement dit, nos routes stuff.

Dans le routeur stuff :

```
const auth = require('../middleware/auth');
```

Et pour chaque route, on rajoutera notre middleware :

```
router.get('/', auth, stuffCtrl.getAllStuff);
```

Maintenant, toute application qui tentera de se connecter à notre API sans avoir un utilisateur authentifié recevra une erreur 401.

Sécuriser nos routes

Une petite seconde.

Vous avez vu ? Nous venons d'importer un module que nous avons nous même créé !

```
const auth = require('../middleware/auth');
```

Contrairement à : `const express = require('express');` où nous importions un module déjà existant.

Ce que nous avons fait : nous avons créé un module natif à notre application Node.js
Ce module, très simple, est écrit avec la toute dernière norme : N-API qui permet que, quelque soit la version de Node utilisée, notre module fonctionne.
N-API assure une compatibilité des données dans le future et Node changeant souvent de version… c'est pas mal !

QUIZ 2

Utiliser postman

Nous pouvons utiliser des services comme postman pour poster des données à nos API et ainsi les tester.

Une fois le compte créé, il ne reste plus qu'à poster une requête :

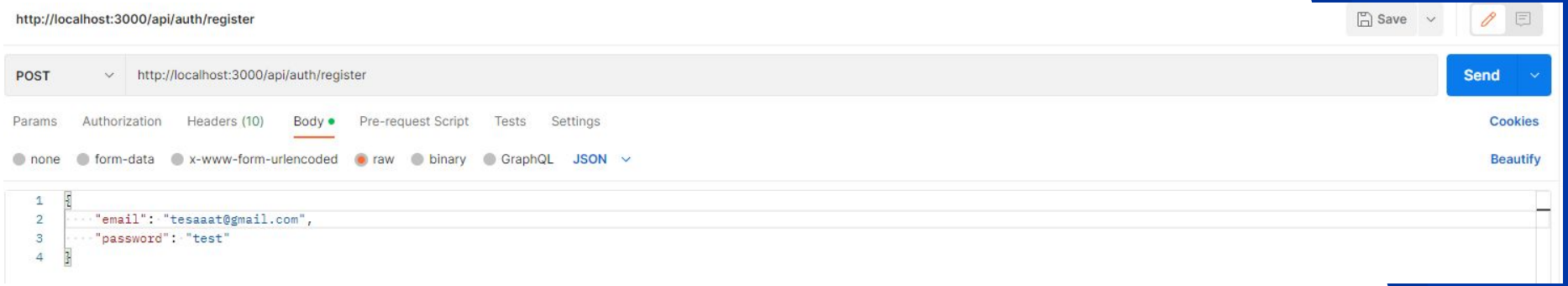
GET :

Utiliser postman

N’oubliez pas le header :

Key = Content-Type
Value = application/json

Pour un post :



TP 2

Gestion des fichiers statiques

Uploader des fichiers

Nous allons implémenter des envois de fichiers pour que nos users puissent uploader des images d'objets à vendre.

Un package gère ça : multer

```
npm install multer
```

Les images seront enregistrées dans un dossier images, créez le.

Créez un fichier multer-config.js … où à votre avis ?

Dans middleware, car c'est un bout de code que l'on appellera lorsque l'on aura besoin de gérer l'upload.
Regardons ensemble le code.

Uploader des fichiers

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

const storage = multer.diskStorage({

      // ici, on mettra deux fonctions

});
```

Fonction 1 : destination

```
destination: (req, file, callback) => {
   callback(null, 'images');
},
```

Destination sert à indiquer à multer où se trouve le dossier où stocker l'image.
L'écriture est un peu lourde mais req et file sont obligatoires, on pourrait en avoir besoin.

Uploader des fichiers

Fonction 2 : filename

```
filename: (req, file, callback) => {
   const name = file.originalname.split(' ').join('_');
   // créer un tableau MIME_TYPES d'abord (slide suivante)
}
```

1. On récupère le filename original, on le met sous forme de tableau en splittant les espaces puis on reforme une string à partir du tableau en liant les mots avec des underscores.
2. Malheureusement, multer ne récupère pas l'extension… mais on a le MIME_TYPES dans file. (prochaine slide).

Uploader des fichiers

Fonction 2 : filename

```
const MIME_TYPES = {
    'image/jpg': 'jpg',
    'image/jpeg': 'jpg',
    'image/png': 'png'
};
```

Ici, on fait un simple tableau MIME_TYPES qui prend en clé un MIME_TYPE et en valeur une extension d'image.

A mettre avant la fonction diskStorage(), évidemment.

Uploader des fichiers

Fonction 2 : filename

```
filename: (req, file, callback) => {
    const name = file.originalname.split(' ').join('_');
    const extension = MIME_TYPES[file.mimetype];
    callback(null, name + Date.now() + '.' + extension);
}
```

De retour sur filename, il ne reste plus qu'à récupérer l'extension puis à créer le callback qui retournera le nom du fichier avec la date actuelle en milliseconde (Date.now()) et l'extension.

Un export pour finir et c'est gagné.

```
module.exports = multer({storage: storage}).single('image');
```

Modifier la route POST

Nous avons un middleware Multer.
Comment feriez-vous pour l'utiliser ?

Il faut l'ajouter à la route post de stuff.js :

```
const multer = require('../middleware/multer-config');

router.post('/', auth, multer, stuffCtrl.createThing);
```

❗ On met multer après l'authentification et avant de créer un objet en base car pour créer l'objet en base, on a besoin de l'url de l'image, donnée par multer.

Modifier la fonction createThing

Sans l'upload d'image, nous faisions un :

```
const thing = new Thing({
    ...req.body
});
```

Car le frontend nous envoyait les données (après que l'user ait submit le formulaire) sous format JSON. Ainsi, req.body est un tableau JSON… parfait pour nous.

Seulement voilà.. avec l'upload de fichier, le frontend ne peut plus envoyer un tableau JSON.. car on ne recevrait pas l'image. Il est obligé d'envoyer une requête **form-data**.

Alors, nous devons modifier createThing()... je vous montre !

Modifier la fonction createThing

```
exports.createThing = (req, res, next) => {
   const thingObject = JSON.parse(req.body.thing);
   delete thingObject._id;
   delete thingObject._userId;
   const thing = new Thing({
       ...thingObject,
       userId: req.auth.userId,
       imageUrl: `${req.protocol}://${req.get('host')}/images/${req.file.filename}`
   });

   // le reste ne change pas

};
```

Renvoyer une image

Testez l'ajout via le frontend (prenez la partie 4) : l'objet s'ajoute mais l'image ne s'affiche pas.

Normal : lorsque le frontend essaie de charger l'image, il ne peut pas faire quelque chose comme : objet.getImageUrl() pour la récupérer car l'image est stockée…. sur l'API, et non sur la partie front !

Le front doit donc effectuer une **requête** à notre API où il indiquera l'url de l'image… pour que l'API lui serve l'image !

Qui dit faire une requête dit, pour nous les backendmen, une .. ?

… nouvelle route ! Directement dans app.js

```
app.use('/images', ????);
```

Voyons ensemble ce qu'il faut mettre à la place des "???"

Renvoyer une image

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

1. Comme le dit la doc d'express.js : Pour servir des fichiers statiques tels que les images, les fichiers CSS et les fichiers JavaScript, utilisez la fonction de logiciel intermédiaire intégré express.static dans Express.
2. `path.join()` permet d'accéder au path de notre serveur (au dossier)
3. `path.join(__dirname, 'images')` permet d'accéder précisément au dossier "images"

Il faudra, bien évidemment, importer le path :

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

Modifier la fonction updateThing

Sur le même principe, on va devoir modifier updateThing avec une subtilité :

- si l'user transmet une nouvelle image, la requête sera une form-data
- si l'user n'en transmet pas, la requête sera une JSON

Nous devrons donc gérer ces deux possibilités.

Tout d'abord, ajoutons multer comme middleware à notre route PUT.

Puis modifions updateThing ensemble.

Modifier la fonction updateThing

Premièrement, on va récupérer l'objet modifié.

On test si req.file existe pour savoir si une image a été uploadée.

```
const thingObject = req.file ? {
    ...JSON.parse(req.body.thing),
    imageUrl: `${req.protocol}://${req.get('host')}/images/${req.file.filename}`
} : { ...req.body };
```

Dans les deux cas, on récupère un thingObject au format JSON.

A la suite, nous supprimons le _userId envoyé par le frontend

```
delete thingObject._userId;
```

Modifier la fonction updateThing

A la suite : on récupère le Thing que l'on a récupéré en BDD.
Si l'userId du thing récupéré en BDD correspond à l'id de l'user connecté, c'est bon, on peut update !

```
Thing.findOne({_id: req.params.id})
    .then((thing) => {
        if (thing.userId !== req.auth.userId) {
            res.status(401).json({ message : 'Not authorized'});
        } else {
            Thing.updateOne({ _id: req.params.id}, { ...thingObject, _id: req.params.id})
                .then(() => res.status(200).json({message : 'Objet modifié!'}))
                .catch(error => res.status(401).json({ error }));
        }
    })
    .catch((error) => {
        res.status(400).json({ error });
    });
```

Modifier la fonction deleteThing

Aujourd'hui, n'importe quelle application pourrait appeler notre route DELETE pour supprimer un objet de l'api car on ne contrôle pas que seul l'utilisateur qui a créé l'objet puisse le faire.

Modifiez donc la fonction deleteThing().
Je vous laisse faire.

Modifier la fonction deleteThing

Votre fonction doit ressembler à ça :

```
exports.deleteThing = (req, res, next) => {
   Thing.findOne({_id: req.params.id})
       .then((thing) => {
           if (thing.userId !== req.auth.userId) {
               res.status(401).json({message: 'Not authorized'});
           } else {
               Thing.deleteOne({_id: req.params.id})
                   .then(() => res.status(200).json({message: 'Objet supprimé !'}))
                   .catch(error => res.status(400).json({error}));
           }
       })
       .catch((error) => {
           res.status(400).json({error});
       });
}
```

Modifier la fonction deleteThing

Et ça fonctionne… mais il manque une seule petite chose pour faire propre.

Supprimer l'image du serveur également !

Pour cela, nous aurons besoin du package "fs" pour File System, qui nous permettra de récupérer l'image serveur.

A mettre en haut du controller :

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

Je vous montre la suite.

Modifier la fonction deleteThing

```
const filename = thing.imageUrl.split('/images/')[1];
fs.unlink(`images/${filename}`, () => {

    // ici, c'est le code de deleteOne que nous avons déjà

});
```

Logique intensive

Faisons une petite pause de dev (une petite) pour revenir sur le concept de Thread dans JS.

JavaScript (et Node.js au début), c’est un Thread. C'est-à dire un tunnel, un seul processus qui exécute plein d'événements de manière asynchrone.

Avec la popularité grandissante de Node, les entreprises qui ont commencé à migrer leurs systèmes sur cette techno ont vite rencontré un problème : que se passe-t-il lorsqu’un événement est super lourd en ressource ?

Le programme rame, voir plante si le code exécuté est trop lourd.

Worker thread à la rescousse

Le worker thread a été créé pour permettre d'exécuter un code en parallèle d'un autre en gérant des processus fils.

Testons cela ensemble dans un nouveau dossier : workers.

Créons un fichier dans ce nouveau dossier : worker.js

Je vous donne le code et on le regarde ensemble.

Worker thread à la rescousse

```
const { Worker } = require('worker_threads')

module.exports = (req, res, next) => {

    return new Promise((resolve, reject) => {
        const worker = new Worker(req)

        // Une fois le worker actif
        worker.on('online', () => {
            console.log('DEBUT : Execution de la tâche intensive en parallèle')
        })

        // Si un message est reçu du worker
        worker.on('message', workerMessage => {
```

Worker thread à la rescousse

Notre worker.js lancera un processus fils secondaire.

Nous allons créer un fichier tache-intensive.js dans workers pour simuler une tâche intensive :

```
const { parentPort } = require('worker_threads')

let count = 0
for (let i = 0; i < 10000000000; i++) {
    count += 1
}
console.log(`FIN: ${count}`)
const message = `Tâche intensive terminée, total : ${count}`

// On renvoie un message depuis le worker récupérer lors du worker.on('message'...)
parentPort.postMessage(message)
```

Worker thread à la rescousse

Il suffira maintenant d'importer le worker dans stuff.js :

```
const worker = require('../workers/worker');
```

Puis de l'utiliser. Mettons le dans createThing, une fois qu'un objet a été sauvé en base :

```
console.log("C'est parti");
worker('./workers/tache-intensive.js');
res.status(201).json({message: 'Objet enregistré !'})
console.log("Et c'est terminé !");
```

QUIZ 3

Les tests

Tests avec Mocha et Chai

Comment peut-on tester son application ?

L'idée sera de pouvoir tester une route, prenons par exemple la route /api/stuff qui retourne tous les stuffs.

Créons un dossier tests et mettons un fichier stuff.test.js

Nous aurons besoin de divers package :

```
process.env.NODE_ENV = 'test';
let server = require('../app');

let chai = require('chai');
let chaiHttp = require('chai-http');
let should = chai.should();
chai.use(chaiHttp);
```

Tests avec Mocha et Chai

Chai et chai-http n'existant pas, on les installe avec la commande --save-dev

npm install –save-dev chai
npm install –save-dev chai-http

Je vous montre pour la suite.

Tests avec Mocha et Chai

```
describe('/GET stuff', () => {
    it('it should GET all the stuff', (done) => {
        chai.request(server)
            .get('/api/stuff)
            .end((err, res) => {
                res.should.have.status(200);
                res.body.should.be.a('array');
                done();
            });
    });
});
```

Tests avec Mocha et Chai

Si on lance le test, cela ira voir la route /api/stuff et l'exécutera.

Pour lancer un test, on utilisera mocha :

```
npm install –save-dev mocha
```

puis, dans package.json, on complète “scripts”

```
"scripts": {
 "test": "mocha tests/stuff.test.js"
},
```

Tests avec Mocha et Chai

Pour lancer le test, il suffira de stopper le serveur (s'il tourne) et de faire un :

```
npm test
```

Pourquoi ai-je l'erreur :

Uncaught AssertionError: expected { Object (_events, _eventsCount, ...) } to have status code 200 but got 401 ?

On enlève le middleware auth, on stop le test (ctrl + C) et on relance !.. et là ça passe !

On retrouvera toute la liste des possibilités de test sur : https://www.chaijs.com/api/bdd/

Documenter avec Swagger

Swagger

Une manière de documenter rapidement son API : Swagger.

Il suffira d'installer le package : `npm i swagger-ui-express -S`

Et de créer un fichier à la racine (dossier backend) : swagger.json

Dans app.js, on mettra en haut du fichier :

```
const swaggerUi = require('swagger-ui-express');
swaggerDocument = require('./swagger.json');
```

Dans app.js, on mettra avant le module.exports = app :

```
// Documentation SWAGGER
app.use(
   '/api-docs',
```

Swagger

Les infos de base sont les suivantes, mais voyons les ensemble :

```
{
 "swagger": "2.0",
 "info": {
   "version": "3.0.0",
   "title": "API CRUD D'OBJETS",
   "description": "Une API qui permet d'ajouter toutes choses et d'autres"
 },
 "host": "localhost:3000",
 "basePath": "/",
 "tags": [
   {
     "name": "La formation",
     "description": "Une API pour apprendre nodeJS"
```

Swagger

Votre api est accessible : http://localhost:3000/api-docs

Ajoutons lui maintenant les routes… appelées “paths”

Les autres routes

Nous allons voir ensemble les routes une par une.

Si vous souhaitez spécifier des paramètres individuellement (et non via une définition), il faudra écrire :

```
"parameters": [
 {
   "name": "email",
   "in": "query",
   "description": "Email de la personne",
   "type": "string"
 }
]
```

EJS

EJS

NodeJS peut également être utilisé pour faire de l'affichage.
Un module utile : EJS, permet de générer facilement du HTML.

```
npm install ejs
```

Il suffira dans app.js de mettre : `app.set('view engine', 'ejs');`

EJS

Il n’y a plus qu’à créer un dossier views dans lequel on mettra un dossier pages et un fichier index.ejs

Dans index.ejs, mettez `<h1>Test</h1>`.

EJS

Créons une route example.js dans route.

Toutes nos routes auront le préfixe /example.

Le route fera, dans un premier temps, un simple chargement d’index:

```
router.get('/', (req, res) => res.render('pages/index'));
```

EJS

Rdv sur localhost:3000/example, la page s’affiche !

EJS

EJS utilise un système de partials pour inclure les bouts de code qui se répètent dans notre HTML.

Créons ensemble un dossier partials dans views puis créons les fichiers head.ejs, header.ejs et footer.ejs.

head.ejs

```
<meta charset="UTF-8">
<title>EJS Is Fun</title>

<!-- CSS (load bootstrap from a CDN) -->
<link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.0.2/dist/css/b
ootstrap.min.css" rel="stylesheet"
integrity="sha384-EVSTQN3/azprG1Anm3QDgpJLIm9Nao0Yz1ztcQTwFsp
d3yD65VohhpuuCOmLASjC" crossorigin="anonymous">
<style>
   body { padding-top:50px; }
</style>
```

header.ejs

```
<nav class="navbar navbar-expand-lg navbar-light
bg-light">
    <a class="navbar-brand" href="/">EJS Is Fun</a>
    <ul class="navbar-nav mr-auto">
        <li class="nav-item">
            <a class="nav-link" href="/">Home</a>
        </li>
        <li class="nav-item">
            <a class="nav-link" href="/about">About</a>
        </li>
    </ul>
</nav>
```

footer.ejs

```
<p class="text-center text-muted">© Copyright 2020 The Awesome People</p>
```

about.ejs

```
<!DOCTYPE html>
<html lang="en">
<head>
    <%- include('../partials/head'); %>
</head>
<body class="container">

<header>
    <%- include('../partials/header'); %>
</header>
```

EJS

Poussons un peu plus l'affichage avec un listing des Thing qui sont dans notre base, directement grâce à notre API.

Créons un fichier example.js dans controllers et déclaration **getAllThings** qui devra récupérer toutes les Thing (la même que celle de stuff). La seule différence sera qu'au lieu de renvoyer une réponse JSON, nous créerons un affichage.

Je vous montre.

EJS

```
const Thing = require("../models/Thing");
exports.getAllThings = (req, res, next) => {
    Thing.find()
        .then(things => {

            res.render('pages/index', {
                things: things
            });

        })
        .catch(error => res.status(400).json({error}));
}
```

EJS

On modifie la route pour qu’elle appelle getAllThings plutôt que de faire le render() directement et le tour est joué, il ne reste plus qu’à modifier la vue.

EJS

A ajouter après le paragraphe :

```
<ul>
   <% things.forEach(function(thing) { %>
        <li>
            <strong><%= thing.title %></strong> au prix de <%=
thing.price %>€
        </li>
   <% }); %>
</ul>
```

Gestion des fichiers

Créer un fichier

Lorsque l'on update un objet, nous allons créer un fichier avec un message à l'intérieur.

Créons un dossier "fichiers".

Le reste se passe dans stuff.js
Une fois qu'un Thing a été update, on veut écrire dans un fichier : "L'objet <title> a été modifié".

Je vous montre.

Créer un fichier

```
fs.writeFile('fichiers/logModification.txt', L'objet ${thing.title}
a été modifié`
, err => {

    if(err){
        throw err;
    }

    console.log("Fichier créé !");

} )
```

Créer un fichier

OK, mais mauvaise pratique… laquelle ?

La logique de création de fichier est placée dans le controller… mais que se passe-t-il si l'on veut créer d'autres fichiers plus tard ?.. on répète et c'est pas DRY.

A vous de jouer : créez un fichier gestionFichier.js dans evenement et faites en sorte qu'un évenement "creerFichier" soit lancée lorsque l'on veut créer un fichier.

Créer un fichier

Dans events/gestionFichier :

```
let evenement = new event.EventEmitter();


evenement.on('creerFichier', (params) => {


    fs.writeFile('fichiers/' + params.fileName,params.message, err
=> {


        if(err){
            console.log(err);
            throw err;
        }
```

Créer un fichier

Dans controller/stuff :
const event = require('../events/gererFichier');

```
event.emit('creerFichier', {fileName: 'logModification.txt',
message: `L'objet ${thing.title} a été modifié \n`});
```

Ecrire dans un fichier existant

Modifiez plusieurs objets et regardez le fichier : il est écrasé à chaque écriture… bien pour une facture qu'on modifierait mais pas top pour des logs !

C'est la fonction appendFile qui va nous aider, voyons cela ensemble.

Créer un fichier

Je changer writeFile par appendFile et j'ajoute un \n après le message

```
evenement.on('appendFichier', function(params){


    fs.appendFile('fichiers/' + params.nomFichier, params.message
+ "\n", err => {
        if(err){
            throw err;
        }

        console.log("Fichier créé !");


    } )
});
```

D’autres fonctions

Il existe plusieurs autres fonctions de FS que nous ne verrons pas ici mais qui sont très simples à utiliser :

Lire

```
fs.readFile('nomFichier', 'utf8', (err, data) => {

    console.log(data);

})
```

D’autres fonctions

Copier :

```
fs.copyFile('nouveauFichier.txt', 'nouveauFichier_copie.txt',
(err) => {
    if (err) throw err;
    console.log('Fichier copié !');
});
```

D’autres fonctions

Renommer :

```
fs.rename('nouveauFichier.txt', 'fichier.txt', (err) => {
    if (err) throw err;
    console.log('Fichier renommé !');
});
```

Bonus : mysql

Mysql

Il est bien sûr possible d'utiliser une base de données mysql avec nodeJS.
Sans entrer dans les détails, voici comment on fait :

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

const con = mysql.createConnection({
    host: "localhost",
    user: "root",
    password: "",
    database: "mabdd"
});
```

Mysql

Puis, on récupère les données ainsi pour les servir au front :

```
con.connect(function (err) {
    if (err) throw err;
    console.log("Connecté à la base de données MySQL!");
    con.query("SELECT * FROM eleves", function (err, result) {
        if (err) throw err;
        console.log(result);
    });
});
```

TP FINAL