NodeJS API - A1 - Hello World

NodeJS est un écosystème pour faire fonctionner du javascript en dehors du navigateur web classique. Pour faire du web, il n'est aujourd'hui quasiment jamais utilisé seul. On utilisera souvent des framework comme surcouche pour faciliter le développement.

Ex :

- Express (le plus populaire)
- Nest.JS
- Fastify
- ...

Si aujourd'hui, de plus en plus de framework commence à gérer nativement le TypeScript, tous utilisent Javascript sous le capot.

Ce cours se portera sur Express mais tout ce qui est appris ici peut être en général retranscrit dans tous les frameworks (avec une adaptation ^^).

Pour faire fonctionner ce dernier, il faudra donc installer NodeJS et npm tout deux fourni dans une seule installe Node. Prenez la version LTS.

Coté IDE, je vous recommande Visual Studio Code.

01 - Mise en place

Créez votre dossier de travail sur votre machine. Ouvrez le dossier en question avec VSCode. Ouvrez un terminal dans votre IDE, celui-ci devrait être automatiquement dans le bon dossier.

Une fois dans votre terminal exécutez la commande suivante

```
npm init
```

Puis laissez vous guider par la config (en général, tapez la touche entrée).

Une fois fini, il est temps d'installer notre dépendance principale :

```
npm install express
```

Inspectez votre fichier `package.json`, vous devriez y trouver des infos importante.

```
"scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "node server.js"
  },
```

Cela défini que vous pouvez lancer les commande `npm start` et `npm test`. La seule qui nous intéressera sera start qui permet de lancer le serveur en exécutant un certain fichier server.js. Il est donc grand temps de créer ce fichier

```
//server.js d'exemple
const express = require('express')
const app = express()
const port = 3000

app.get('/', (req, res) => {
  res.send('Hello World!')
})
app.listen(port, () => {
  console.log(`Example app listening on port ${port}`)
})
```

Si tout s'est bien déroulé, dans le terminal (qui servira de console à votre serveur) devrait s'afficher `Example app listening on port 3000`.
Puis tentez d'aller depuis votre navigateur sur l'url `http://localhost:3000` et voir un magnifique Hello world !

Votre première API est donc fonctionnelle !

Pour tester votre API, vous pouvez aller via votre navigateur. Mais cela va être vite limité quand nous utiliseront d'autres verbes HTTP comme POST/PUT/DELETE.

Je vous invite donc à utiliser une extension VSC comme Postman ou bien le logiciel stand alone Bruno pour gérer vos différentes routes.

02 - Premier contrôleur simple avec MVC

L'objectif en plus d'avoir une API qui fonctionne, c'est d'avoir une API bien organisé. Cela aura l'avantage d'être plus pratique pour retrouver son code et ajouter de nouvelles fonctionnalités.

Plein de structures différentes existent mais nous allons rester simple. En utilisant le pattern Model View Controller, nous allons avoir besoin de plusieurs dossiers :

- controllers
- models
- routes

Votre archi devrait ressembler à cela :

```
📦 mmi-nodejs-api
 ┣ 📁 controllers
 ┣ 📁 models
 ┣ 📁 node_modules
 ┣ 📁 routes
 ┣ 📜 package-lock.json
 ┣ 📜 package.json
 ┣ 📜 README.md
 ┗ 📜 server.js
```

Contrôleur

Nous allons donc dans le dossier `controllers` créer un fichier `hello.controller.js`

Celui ci contiendra pour le moment une seule fonction à exporter :

```
exports.HelloWorld = (req, res) => {
    res.status(200).json({ message: "Hello World" });
}
```

Ceci est une fonction fléché. Son nom sera donc `HelloWorld` car elle est exporté sous ce nom. Elle aura 2 paramètres, `req` pour la requête entrante, et `res` pour la requête sortante.

Comme Node.js fonctionne avec un système de middleware qui s'enchainent, les 2 paramètres existent tout au long du cycle de vie de la requête.

Nous définissons alors le `status_code` à 200 qui veut dire succès. Voir les status code

Puis nous créons un format JSON pour la réponse avec une seule clé message et sa valeur.

Route

Nous allons donc dans le dossier `routes` créer un fichier `hello.js`

Celui ci n'aura qu'une seule route pour le moment. Nous avons besoin d'importer express pour notre routeur pour en utiliser sa méthode `Router`

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

Puis nous importons notre contrôleur

```
const helloController = require('../controllers/hello.controller');
```

Nous continuons par indiquer au routeur que notre route `/` utilisera la fonctionne de notre contrôleur

```
router.get('/', helloController.HelloWorld);
```

Puis il nous reste plus qu'à exporter notre routeur dont nous auront besoin dans le serveur

```
module.exports = router;
```

Il nous reste encore à exploiter le hello routeur dans notre application. Rendez-vous dans le fichier server.js puis importez votre routeur. Enfin il faut dire à notre application d'utiliser ce routeur à la route souhaité (`/hello` me semble convenir)

```
const helloRouter = require('./routes/hello');
...
app.use('/hello', helloRouter);
```

Si tout à bien fonctionné, vous devriez pouvoir visiter `http://localhost:3000/hello` et voir le message JSON Hello World (qui diffère du visuel pur texte du premier point)

03 - Requête GET avec paramètre simple

Rajoutez dans le contrôleur une opération qui répond à la requête `/hi` qui, elle, possède un paramètre `name` (une chaîne de caractères) en renvoyant une chaîne `Hi <name> !`. Si le paramètre n'existe pas, alors donnez lui une valeur par défaut.

Pour récupérer un paramètre de votre requête voici un exemple:

```
GET /route?color1=red&color2=blue
```

```
app.get('/route', (req, res) => {
    req.query.color1 === 'red'  // true
    req.query.color2 === 'blue' // true
})
```

A vous de jouer !

Une fois cela fonctionnel, essayer d'ajouter une route qui utilisera un paramètre dans le chemin de base dans l'url plutôt que dans les query params.

```
GET /hi/monparam
```

Pour cela votre routeur devrait avoir une route du style `/hi/:name`. Il faudra alors utiliser `req.params` au lieu de `req.query`.

04 - Recharger !

Actuellement, à chaque changement dans votre code, vous devez shutdown le serveur et le relancer. Cela est certainement fort pénible. Il est grand temps d'améliorer cela !

Pour cela, nous avons un package qui fait très bien le taf ! Installez `nodemon`. Puis, à l'aide de la doc essayez de lancer votre serveur ! Une fois fonctionnel, améliorer votre `package.json` pour que le script start prenne en compte `nodemon`.

05 - Requête POST avec un objet en paramètre.

Les routes GET servent souvent à récupérer les données, et les requêtes POST à créer des données. Nous allons simuler ce comportement pour le moment.

Pour que notre serveur puisse gérer l'envoie d'un corps de requêtes POST, il faut préciser quel format l'on accepte de traiter. Voici un code qui permettra de gérer l'envoi des données au format `raw/JSON` et au format `x-www-url-encoded`.

```
// Dans votre server.js
// Parse URL-encoded bodies (Envoyé par les HTML forms)
// Body x-www-url-encoded
app.use(express.urlencoded({ extended: true }));
// Parse JSON bodies (Envoyé par les clients API)
// Body Raw format JSON
app.use(express.json());
// Cela popule le req.body avec les infos envoyé
```

Nous allons aussi créer un modèle dans notre dossier 📁`models`. Créez une classe Person (donc un fichier `Person.js`) qui dans son constructeur prendra en paramètre un `name` et un `age`. Puis exportez la.

Dans le contrôleur, ajoutez une nouvelle fonction qui aura pour but de récupérer les infos depuis le body (hint : `req.body`) puis d'instancier une `Person` puis de la retourner au format JSON.

Il n'y aura pas de sauvegarde pour le moment !

Testez avec votre client API dans les 2 format et assurez vous que cela fonctionne.

06 - Affichage d'une image ou d'un fichier

Il est totalement possible d'afficher une image ou même un fichier (par exemple de l'html) avec son API.

Créez un dossier `📂images` puis ajouter un fichier dedans.

Ajoutez une fonction dans votre contrôleur qui aura pour but d'afficher l'image. Comme le dossier ou se trouve notre fichier n'est pas au même endroit, dans un soucis de sécurité par express, il nous faudra résoudre le chemin de l'image pour l'avoir en dur (à partir d'un chemin relatif).
Importez le module `path` puis utilisez sa méthode résolve qui a partir d'un chemin relatif de votre image (il part depuis l'emplacement de votre `server.js`). Puis vous retourne le chemin absolu en tant que string.
Il ne vous reste plus qu'a utiliser `res.sendFile()` et ajouter votre route GET.

Testez depuis votre navigateur pour voir votre image s'afficher (Thunder gratuit ne le permet pas)

07 - Environnement

Dans l'idée d'avoir des variables partagées à toute l'application, facilement paramétrables depuis l'extérieur (pas besoin de lire le code source par exemple), il existe les variables d'environnement.

Géré nativement depuis Node.js 20, il vous suffira de créer un fichier au format `.env` a la racine de votre application.

Créez donc un fichier `props.env`, puis ajouté une variable nomme `IMGPATH` contenant le chemin avant le nom du fichier utilisé avant :

```
IMGPATH="./images"
```

Modifiez votre package.json pour qu'au lancement de l'application, celle-ci prenne en compte votre `.env`. Pour cela, ajoutez l'option `--env-file=props.env`.

```
Attention, bien mettre l'option avant le nom du fichier js lancé
```

Pour tester si cela fonctionne, affichez la variable `IMGPATH` dans la console grâce à la ligne suivante dans votre server.js:

```
console.log(process.env.IMGPATH);
```

Si le contenu de votre variable s'affiche dans la console, bien joué ! Il ne vous reste plus qu'a vous en servir dans le code de la partie 6.

08 - Découplage du code : Service !

Dans cette partie, nous allons découper notre code de notre contrôleur.

Cette couche est l'endroit où votre logique métier doit vivre. C'est juste un ensemble de classes, suivant les principes SOLID appliqués à node.js.

Objectifs :

- Éloignez votre code du routeur express.js
- Ne transmettez pas l'objet req ou res à la couche service
- Ne renvoyez aucun élément lié à la couche de transport HTTP, tel qu'un code d'état ou des en-têtes de la couche de service.

Pour commencez, créez un dossier `services` puis ajoutez un fichier `person.service.js` contenant une class `PersonService`. Celle-ci aura un constructeur avec un paramètre `basePath` qui remplira l'attribut `this.basePath`. Puis une fonction `resolveFromFileName(filename)`. Celle-ci retournera juste le résultat de la fonction `path.resolve`

Puis réalisez le combo routeur-contrôleur pour créer une route `/imgfilename?filename=myfile.ext`. Votre fonction du contrôleur devras utiliser le service. La route devrait alors affichez l'image comme celle de la partie 6. Mais vous pouvez maintenant changer le nom de l'image directement depuis l'url. Bonus en prime, votre code se rapproche des principes SOLID !

Bien joué d'être arrivé en vie ! Rendez vous au 2ème sujet !