Création des Web Services Rest

Dans ce TP, nous allons créer ensemble une **API REST** pour que nous puissions comprendre chaque élément qui la constitue.

Créer un serveur Express

Node JS API est avant tout un serveur web à l'écoute des requêtes HTTP entrantes. Pour démarrer ce serveur web, nous allons utiliser le ***framework Express***.

Démarrage du projet Node JS API

1. Créer votre répertoire de votre future API et naviguez à l'intérieur
2. Saisir la commande « **npm init -y** »
3. Créer un fichier « **index.js** »

Nous aurons maintenant dans notre répertoire un fichier « **package.json** », qui va reprendre différentes informations du projet et qui contiendra les dépendances qu'on va y installer.

La commande « **npm init** » génère le fichier **package.json**, qui est le squelette de l'API et ses dépendances.

Ajout d'Express à notre Node JS API

Retourner maintenant à notre terminal et taper la commande suivante :

```
npm install express
```

Cette commande a pour but de télécharger depuis la registry NPM puis d'installer la librairie express ainsi que l'ensemble des librairies dont express a besoin pour fonctionner dans votre répertoire de travail, dans le répertoire *node_modules*. **NPM va également l'ajouter dans le package.json** dans l'objet *dependencies*.

Création du serveur Express dans notre fichier index.js

Maintenant qu'Express est disponible dans notre projet, nous pouvons créer le serveur. Commençons par intégrer la librairie express dans notre fichier index.js :

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

Le **require('express')** est une façon d'importer la librairie express et ses fonctions dans notre code. La constante **app** est l'instanciation d'un objet Express, qui va contenir notre serveur ainsi que les méthodes dont nous aurons besoin pour le faire fonctionner.

Pour le moment, le serveur est préparé mais pas encore lancé. Si nous nous rendons sur **localhost:8080** depuis le navigateur, nous devrions avoir une erreur.

Pour que notre serveur puisse être à l'écoute il faut maintenant utiliser la méthode **listen** fournie dans **app** et lui spécifier un port. Le plus souvent en développement nous utilisons 8080, 3000 ou 8000. Ça n'a pas d'importance tant que nous n'avons pas d'autres applications qui tournent localement sur ce même port.

```
app.listen(8080, () => {  console.log('Serveur à l'écoute)
})
```

En lançant la commande **node index.js** dans le terminal, nous verrons qu'il affichera que notre serveur est à l'écoute. Cela veut dire que tout fonctionne bien. S'il y a une erreur, nous aurons droit à un message d'erreur sur le terminal.

Le serveur Node est à l'écoute mais ne sait pas quoi faire pour l'instant.

Si nous nous rendons sur le navigateur à l'adresse **localhost:8080** (ou l'autre port que nous aurions choisi), Le serveur répond au navigateur. N'ayant pour l'instant aucune route de configurée, il vous retourne cette erreur **Cannot GET** / mais il est bel et bien fonctionnel.

Définir une ressource et ses routes

Maintenant que le serveur est fonctionnel, il est temps de définir **le cœur de l'API : ses ressources**.

Définition des ressources de notre Node JS API

Pour notre exemple, nous prendrons le cas d'une société exploitant des parkings de longue durée et qui prend des réservations de la part de ses clients. Nous aurons besoin des fonctionnalités suivantes :

- Créer un parking
- Lister l'ensemble des parkings
- Récupérer les détails d'un parking en particulier
- Supprimer un parking
- Prendre une réservation d'une place dans un parking
- Lister l'ensemble des réservations

- Afficher les détails d'une réservation en particulier
- Supprimer une réservation

Ces opérations sont plus communément appelées CRUD, pour CREATE, READ, UPDATE, DESTROY. Dans notre exemple, **Node JS API dispose de deux ressources : le Parking et la Réservation**.

Création des routes

Le standard d'API REST impose que nos routes soient centrées autour de nos ressources et que la méthode HTTP utilisée reflète l'intention de l'action. Dans notre cas nous aurons besoin des routes suivantes :

- GET /parkings
- GET /parkings/:id
- POST /parkings
- PUT /parkings/:id
- DELETE /parkings/:id

Les réservations étant une sous-ressource de la ressource parking, nous aurons à créer les routes suivantes :

- GET /parkings/:id/reservations
- GET /parking/:id/reservations/:idReservation
- POST /parkings/:id/reservations
- PUT /parking/:id/reservations/:idReservation
- DELETE /parking/:id/reservations/:idReservation

Pour que notre Node JS API fonctionne, nous avons besoin de données échantillon.

Le but est de comprendre le fonctionnement d'une API. Nous n'allons pas connecter de vraie base de données dans ce guide. **Nous allons à la place utiliser un fichier JSON contenant un échantillon de données pour manipuler notre API (parkings.json)**.
Commençons par définir la route GET /parkings. Cette route a pour but de récupérer l'ensemble des parkings dans nos données. Allons modifier notre fichier **index.js** :

```
const express = require('express')const app =
express()app.get('/parkings', (req,res) => {
res.send("Liste des parkings")})app.listen(8080, () => {
console.log("Serveur à l'écoute")})
```

La méthode **.get** d'express permet de définir une route GET. Elle prend en premier paramètre une *String* qui a défini la route à écouter et une *callback*, qui est la fonction à exécuter si cette route est appelée. Cette callback prend en paramètre l'objet **req**, qui reprend toutes les données fournies par la requête, et l'objet **res**, fourni par express, qui contient les méthodes pour répondre à la requête qui vient d'arriver.

Dans ce code, à l'arrivée d'une requête GET sur l'URL **localhost:8080/parkings**, le serveur a pour instruction d'envoyer la *String* « Liste des parkings ».

Couper le serveur node s'il tourne encore (avec la commande ctrl+c dans le terminal) et relancez la commande **node index.js** pour prendre en compte les modifications.

L'API peut maintenant répondre aux requêtes HTTP

Maintenant que notre route fonctionne et est capable de recevoir la requête entrante, nous allons pouvoir renvoyer la donnée des parkings au lieu d'avoir simplement une chaîne de caractères :

```
const express = require('express')const app =
express()const parkings =
require('./parkings.json')app.get('/parkings', (req,res)
=> {    res.status(200).json(parkings)})app.listen(8080,
() => {    console.log("Serveur à l'écoute")})
```

Nous avons remplacé la méthode **send** par la méthode **json**. En effet notre API REST va retourner un fichier JSON au client et non pas du texte ou un fichier html. Nous avons également ajouté le statut 200, qui correspond au code réponse http indiquant au client que sa requête s'est terminée avec succès.

Notre route GET /Parkings est maintenant terminée. Il faut maintenant mettre en place les routes suivantes en utilisant les méthodes JavaScript pour répondre à nos besoins.

La route GET /parkings/:id est la suivante. Nous avons besoin de récupérer l'id de la route depuis l'URL pour n'afficher que le JSON de ce parking dans la réponse. Cet id se trouve dans les *params*, dans l'objet **req**, envoyé par le navigateur.

Reprenons notre fichier *index.js* :

```
const express = require('express')const app =
express()const parkings =
require('./parkings.json')app.get('/parkings', (req,res)
=> {
res.status(200).json(parkings)})app.get('/parkings/:id',
(req,res) => {    const id = parseInt(req.params.id)
```

```
const parking = parkings.find(parking => parking.id ===
id)    res.status(200).json(parking)})app.listen(8080, ()
=> {    console.log("Serveur à l'écoute")})
```

Nous récupérons l'*id* demandé par le client dans les *params* de la requête. Comme ma route a défini '/:id', la valeur passée dans le *param* sera sous forme d'objet contenant la clé « *id* » . La valeur de req.params.id contient ce qui est envoyé dans l'URL, sous forme de String. Comme l'id de chaque parking est sous forme de Number, il faut d'abord transformer le *params* de String en Number. Ensuite, il faut rechercher dans les parkings pour trouver celui qui a l'*id* correspondant à celui passé dans l'URL.

Passons à la route POST /parkings pour pouvoir créer un nouveau parking.

Pour créer un nouveau parking via votre Node JS API, il va falloir envoyer au serveur les données relatives à ce nouvel élément, telles que son nom, son type etc. **Dès qu'il s'agit d'envoyer de la donnée, il faut utiliser une requête POST**.

Pour récupérer les données passées dans la requête POST, nous devons ajouter un *middleware* **à notre Node JS API afin qu'elle soit capable d'interpréter le body de la requête.** Ce middleware va se placer à entre l'arrivée de la requête et nos routes et exécuter son code, rendant possible l'accès au body.

Voici notre fichier *index.j s*:

```
const express = require('express')const app =
express()const parkings = require('./parkings.json')//
Middlewareapp.use(express.json())app.get('/parkings',
(req,res) => {
res.status(200).json(parkings)})app.get('/parkings/:id',
(req,res) => {    const id = parseInt(req.params.id)
const parking = parkings.find(parking => parking.id ===
id)    res.status(200).json(parking)})app.listen(8080, ()
=> {    console.log("Serveur à l'écoute")})
```

Il n'y a plus qu'à ajouter la route POST et à tester notre nouvelle route :

```
const express = require('express')const app =
express()const parkings = require('./parkings.json')//
Middlewareapp.use(express.json())app.get('/parkings',
(req,res) => {
res.status(200).json(parkings)})app.get('/parkings/:id',
(req,res) => {    const id = parseInt(req.params.id)
const parking = parkings.find(parking => parking.id ===
id)
res.status(200).json(parking)})app.post('/parkings',
(req,res) => {    parkings.push(req.body)
res.status(200).json(parkings)})app.listen(8080, () => {
console.log("Serveur à l'écoute")})
```

Pour tester notre route POST, nous allons utiliser l'outil Postman qui nous permet de manipuler facilement des API.

Notre requête POST sur l'URL **localhost:8080/parkings** contient dans son body un objet JSON contenant l'id, le nom, le type et la ville de notre nouveau parking.

Dans un cas réel de Node JS API, votre base de données aurait généré l'id. Dans notre cas nous allons le passer à la main pour simplifier.

Passons à **la route PUT /parkings/:id** pour pouvoir modifier un parking.

```
app.put('/parkings/:id', (req,res) => {    const id =
parseInt(req.params.id)    let parking =
parkings.find(parking => parking.id === id)
parking.name =req.body.name,    parking.city
=req.body.city,    parking.type =req.body.type,
res.status(200).json(parking)})
```

Il reste maintenant à terminer cette ressource avec la route `**DELETE /parkings**`

```
app.delete('/parkings/:id', (req,res) => {    const id =
parseInt(req.params.id)    let parking =
parkings.find(parking => parking.id === id)
parkings.splice(parkings.indexOf(parking),1)
res.status(200).json(parkings)})
```

La route DELETE permet d'effacer un élément de la ressource grâce à votre Node JS API

Votre Node JS API est maintenant capable de gérer la ressource Parking. Pour mettre en place la sous-ressource Réservation, il faut répliquer la logique.