NodeJS : Une Todo Liste

(À réaliser en binôme)

Lors du précédent projet, vous deviez faire de la mise en forme de données récupérée via une API publique. Cette fois, vous devez réaliser une Todo List en utilisant NodeJS. Elle sera composée d'une :

- Partie API REST (Serveur)
- Partie Vue (Client)

Des ressources pour le démarrage du projet sont fournies.

Le rendu attendu pour ce travail est :

- Vos sources (avec un fichier readme de type « get started » et bien sûr le code commenté)
- Un rapport détaillant votre projet (Format d'usage : Rappel des objectifs, degré d'aboutissement, répartition des tâches, axes d'amélioration etc.). Pas de capture d'écran de votre code.
- Une courte vidéo de l'interface avec test des cas d'utilisations développés (via OBS Studio par ex)

Bien évidemment pour les étudiants les moins à l'aise, nous n'attendons pas forcément que tout soit fait.

Faites de votre mieux !

Partie 0 : Les bases de nodeJS

Suivez le tutoriel suivant afin d'acquérir les bases et les bonnes pratiques nécessaires à la réalisation du projet :
https://www.youtube.com/playlist?list=PLjwdMgw5TTLV7VsXd9NOeq39soYXORezN

Pour information, le total des vidéos prends un peu plus de 2h, il vous faudra donc au moins 4h pour la réalisation complète du travail proposé dans ces vidéos.

Quelques conseils pour la bonne réalisation de ce tutoriel :

- Faites les manipulations avec l'auteur de la vidéo
- Pour la sixième vidéo, lors de l'inclusion du header, la syntaxe de la vidéo n'est plus à jour, utilisez la documentation pour trouver la bonne syntaxe : https://ejs.co/#docs
- Afin d'optimiser votre temps, ne restez jamais bloqué plus de 10 minutes sans demander de l'aide.

Partie 1 : Création du projet

A la racine du projet, exécutez :

```
npm init
```

Que s'est-il passé ?

Par la suite, nous aurons besoin d'utiliser les modules Nodes suivants :

- express : pour créer un serveur http
- body-parser : pour parser le corps d'une requête http
- promise-mysql : pour manipuler la base de données (c'est une surcouche du module mysql)

Les installer grâce à la commande :

```
npm install express body-parser promise-mysql --save
```

Partie 2 : Connexion à la base de données

Importez le fichier `todolist.sql` dans votre base de données.

Ouvrez le fichier `db.js` et comprenez le code. Renseignez les bonnes informations pour vous connecter à votre base.

Testez la connexion en exécutant le fichier `test-db.js`

Partie 3 : Implémentation de l'API Rest

Lire cette page (les 3 premières règles) expliquant ce qu'est une API Rest, puis celle-ci expliquant avec plus d'exactitude les différents verbes HTTP que nous allons utiliser.

Qu'elle est la différence d'utilisation entre PUT et PATCH ?

Ajouter à l'application 4 routes pour ajouter, modifier, récupérer et supprimer une tâche de notre todolist. Ajouter également une autre route pour récupérer l'ensemble des tâches de la todolist.

Attention :

- Ces 5 routes doivent **correspondre aux normes RestFull** (URLs des points d'entrée, verbes HTTP, codes de réponse, ajout retournant un entête *location* ...).
- Vos requêtes ne doivent pas être créées par concaténation (`"where id="+id`). Utilisez des requêtes préparées avec des points d'interrogation (`"where id=?"`). Référez-vous pour cela à la documentation du module mysql.

Les contenus (body) des requêtes HTTP sont attendus au format json. Ils seront ainsi directement parsés par le module body-parser et on les récupérera automatiquement sous la forme d'un objet JS :

```
req.body.clé_de_l_élément
```

Exemple de l'ajout d'une tâche, avec utilisation de l'id automatique retourné par la bdd :

```
app.post("/taches", (req, res) => {
    db.then( pool =>
          pool.query('INSERT INTO ...(...) VALUES(?)', [req.body....])
    ).then( results => {
          res.status(201);
          res.location("/tasks/"+results.insertId)
          res.send(null);  // Il n'y a pas de corps de réponse
    }) ;
});
```

Dans cet exemple l'ajout ne retourne pas de corps de réponse (on aurait pu imaginer renvoyer l'objet créé) car le fait de retourner un entête *Location* avec l'ID du nouvel enregistrement est suffisant pour faire des appels ultérieurs à l'API et récupérer cet enregistrement si besoin.

Par contre un appel de l'API en GET devra retourner un objet json.

Pour définir un argument variable dans une route, on utilisera la syntaxe :

```
app.get("/taches/:id_tache", (req, res) => {
    console.log("On demande les détails de la tâche "+req.params.id_tache);
    // je récupère "ma_tache" en faisant une requête en base
    // puis je retourne l'objet :
    res.json(ma_tache) ;  // Je retourne une réponse au format json
});
```

Pour tester les appels à l'API que vous venez de créer, utilisez le plugin firefox RestClient, sans oublier d'ajouter un entête « `Content-Type : application/json` » à chaque requête envoyée pour que le module node body-parser s'active. Vous pouvez également utiliser d'autres solutions comme SOAPUI ou Curl sous linux pour les puristes.

Partie 4 : Partie Vue / Utilisation d'EJS

Jusqu'à présent, les requêtes vers notre service web ne produisaient que du json. Nous souhaitons cette fois produire du HTML pour créer la page web qui constituera l'ihm dans notre navigateur.

Pour générer du HTML côté serveur, nous allons utiliser le module EJS (c'est un middleware de express). Il permet, via un système de balisage dans le code HTML (<% ... %>), d'utiliser des fonctions JS dans la partie vue de l'application (séparation MVC).

Commencez par ajouter EJS à votre projet :

```
npm install ejs --save
```

Modifiez ensuite la route racine ("/").
Elle doit :

- Récupérer les données de la table « taches »
- Les passer à la vue « index.ejs » qui produira le HTML.

Cela se fait de cette façon :

```
app.get("/", (req, res) => {
    db.then( pool =>
          pool.query('SELECT * from taches')
    ).then( results => {
          res.render('index.ejs', {todolist: results});
    }) ;
});
```

Le second paramètre de la fonction `render()`est un objet qui défini quelles données seront passées du contrôleur à la vue. Les clés de l'objet correspondent aux noms des variables que l'on utilisera dans la vue pour utiliser les valeurs associées.

Par défaut sans configuration particulière, les vues EJS sont cherchées dans le répertoire `views/`à la racine de notre projet.

Complétez le fichier `index.ejs` pour qu'un clic sur le bouton « ajouter » appelle l'API réalisée précédemment et ajoute une tâche à la todolist.

Vos appels ajax depuis la vue (votre page HTML) devrons être faits avec l'API Fetch :

```
fetch('/taches/12', {
  headers: {
    'Accept': 'application/json',
    'Content-Type': 'application/json'
  },
  method: 'patch',
  body: JSON.stringify( { "statut": "A_FAIRE" } )
}).then(
    ...
);
```

Faites ensuite en sorte qu'un clic sur une case à cocher modifie le statut de la tâche correspondante.

Lorsque l’application est fonctionnelle, peaufinez le CSS puis implémenter le drag&drop pour donner la possibilité d’ordonner les tâches entre elles (créez un nouveau champ « ordre » en base).