Récupérer des données d'un capteur analogique

La plateforme de prototypage que vous avez montée sur Arduino vous a permis de convertir des mesures physiques de luminosité et de température en des valeurs numériques que vous avez envoyées à un serveur Python. Grâce à ce TD, vous allez maintenant interroger le serveur Python avec un client *Node.js* et enregistrer les mesures dans une base de données orientée documents : *MongoDB*.

Présentation du code de départ

Dans le répertoire *debut*, vous trouverez deux autres répertoires :

- le répertoire *client* qui contient les vues utilisées dans l'application finale ;
- le répertoire *server* qui contient le script Python servant à simuler les données à récupérer.

Conception de l'application

Étape 1 : démarrage de l'application

Créez un répertoire de travail avec le nom de votre application et démarrez un nouveau projet *Express* avec `express-generator` :

```
$ mkdir client && cd client
$ npx express-generator
```

Installez le module *Pug* ainsi que les dépendances :

```
$ npm install
$ npm install --save pug
```

Configurez ensuite le script de démarrage dans le fichier *package.json* pour que la commande `npm start` lance l'application *app.js* avec la commande `nodemon` :

```
"scripts": {
    "start": "nodemon ./bin/www"
}
```

Définissez à présent le module *Pug* comme moteur de rendu de vos templates, en modifiant votre application *app.js* :

```
app.set('view engine', 'pug');
```

Vous pouvez maintenant remplacer les vues du répertoire *views* par celles présentes dans le répertoire de départ.

Si vous lancez l'application, vous devriez voir une page… ben en fait, non, vous obtenez une page d'erreur.

```
$ npm start
```

Étape 2 : lister les routes nécéssaires

L'application que vous configurez ne s'occupera que de quelques tâches de base :

- enregistrer une mesure de température ;
- enregistrer une mesure d'intensité lumineuse ;
- afficher les mesures de température ;
- afficher les mesures de luminosité ;
- afficher les deux types de mesures.

Pour chacune de ces tâches, définissez une route :

- (**GET**) `/light` : enregistrer une mesure d'intensité lumineuse ;
- (**GET**) `/temperature` : enregistrer une mesure de température ;
- (**GET**) `/data` : lister toutes les mesures ;
- (**GET**) `/data/lights` : afficher les mesures de luminosité ;
- (**GET**) `/data/temperatures` : afficher les mesures de température.

Prévoyez également que la page d'index renvoie vers la route `/data`.

Étape 3 : configurer la base de données *MongoDB*

Configurer le service en nuage

Si l'utilitaire *MongoDB* est installé sur votre machine, n'hésitez pas à l'utiliser en adaptant les commandes qui vont suivre. Pour les autres, rendez-vous sur le site MongoDB.com pour utiliser la base de données en tant que service gratuit directement dans le nuage.

Commencez par ouvrir un compte puis, une fois dans votre tableau de bord, créez un nouveau *cluster* en ne sélectionnant que les options gratuites dont celle qui recourt aux *Amazon Web Services* (AWS). Pendant que votre *cluster* se met en place, cliquez sur l'onglet *Database Access* et ajoutez un nouvel utilisateur qui soit autorisé à lire et écrire sur toutes les bases de données. Notez bien le mot de passe ! Ensuite, dans l'onglet *Network Access*, ajoutez une adresse IP en validant l'option qui autorise de se connecter depuis n'importe où.

Dès que le *cluster* est prêt, rendez-vous sur l'onglet *Databases* et appuyez sur le bouton *Connect* puis *Connect your application* pour récupérer l'adresse de connexion.

Paramétrer la connexion au nuage *MongoDB Atlas*

Tout d'abord, installez le paquet *Mongoose* pour dialoguer plus facilement avec *MongoDB* pendant toutes vos opérations:

```
npm install mongoose --save
```

Créez ensuite un répertoire *controllers* où vous éditerez un nouveau fichier *icumoov.js* dans lequel vous appelerez le paquet `mongoose` :

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

Définissez ensuite l'adresse de connexion récupérée précédemment, en adaptant les informations de connexion `<USER>` et `<PASS>` en fonction de l'utilisateur créé sur le nuage :

```
const uri = 'mongodb+srv://<USER>:<PASS>@cluster0.udwqf.mongodb.net/ICUMoov?
retryWrites=true&w=majority';
```

Grâce à la méthode `connect()` de l'objet `mongoose`, établissez une connexion au nuage :

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

Testez maintenant la connexion en incluant le contrôleur dans la route *index.js* :

```
const icumoov = require('../controllers/icumoov');
```

Rechargez la page et abracada… ah ben non en fait, ça ne marche toujours pas. Mais regardez plutôt votre console, vous devriez y lire un message sacrément encourageant : "Connexion à MongoDB réussie !"

Étape 4 : enregistrer les mesures envoyées par la plateforme Arduino

Pré-requis : le serveur Python doit être lancé ! Si vous souhaitez obtenir des données simulées, lancez le script *demo.py* dans le répertoire *server* :

```
python demo.py
```

Votre application *Node* va s'attacher à exécuter deux actions :

1. lancer une commande valide au serveur ;
2. enregistrer le résultat de la commande.

Établir une communication avec le serveur

Pour communiquer avec le serveur, vous devrez établir une connexion TCP sur un port du serveur. Le serveur Python livré avec ce TD définit les paramètres suivants :

- `host` : `localhost` ;
- `port` : `9999` .

Commencez par importer le module *Net* inclus dans *Node.js* :

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

Définissez ensuite une méthode `getData()` qui accepte un paramètre `cmd` dont les valeurs seront `light` ou `temperature` (cf. documentation du serveur). La première, `light`, demandera l'acquisition d'une valeur de luminosité, tandis que la seconde demandera une mesure de température. N'oubliez pas que cette méthode doit renvoyer une fonction de *callback* avec les objets `req` et `res` (et éventuellement `next`) :

```
const getData = () => {
    return (req, res) => {
        const cmd = req.params.cmd;
        // instructions
```

```
    };
};
```

Exportez-la en fin de script :

```
module.exports = {
    getData
};
```

Dans le corps de la méthode `getData()`, connectez-vous au serveur via un *socket* :

```
const socket = net.connect({ port: 9999 });
```

Utilisez ce canal pour transmettre la commande au serveur :

```
socket.write(cmd);
```

Vous pouvez maintenant couper la communication :

```
socket.end()
```

Il vous reste à traiter les informations obtenues en écoutant l'événement `data` :

```
socket.on('data', (data) => {
    const value = data.toString().substr(4);
    res.render('data', { type: cmd, data: value });
});
```

Votre contrôleur est prêt, vous pouvez l'utiliser en définissant une route dans le fichier *routes/index.js* pour envoyer une commande au serveur :

```
router.get('/:cmd', icumoov.getData());
```

Consultez la page *light* ou la page *temperature* pour admirer le résultat de votre commande !

Enregistrer les mesures dans la base de données

Comment enregistrer les données dans la base *MongoDB* ? Si vous venez du monde des bases de données relationnelles, vous serez les premiers temps déconcerté·es par les spécificités de *MongoDB*. Ce TD n'a d'autre objectif que d'aller à l'essentiel afin de vous mettre le pied à l'étrier !

Un peu de vocabulaire tout d'abord : une collection (une table au sens des SGBDR) est composée de documents (les enregistrements). Pour le moment, vous avez simplement créé une base de données *ICUMoov* dans *MongoDB Atlas*. Comment ? Vous ne vous rappelez pas l'avoir fait ? En fait, dans l'adresse de connexion (`'mongodb+srv://<USER>:<PASS>@cluster0.udwqf.mongodb.net/ICUMoov?retryWrites=true&w=majority';`) figure le nom d'une base de données par défaut. Le simple fait de se connecter à une base de données permet de la créer ! C'est comme lorsque vous lancez en *shell* une commande `vim test.txt` sur un fichier *test.txt* qui n'existe pas.

Dans le même ordre d'idée, nul besoin de créer les collections. Vous avez en revanche besoin de décrire un modèle pour les documents. En d'autres termes, quels seront les champs à intégrer et de quelle nature sont-ils ?

Définir un modèle de données

Commencez par créer un répertoire *models* à la racine de votre application. Importez ensuite le module *mongoose* dans un fichier *light.js* qui servira de modèle pour la collection *Light* :

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

La classe `Schema()` du module vous permettra ensuite de décrire les champs d'un document *Light* :

```
const lightSchema = mongoose.Schema({
    light: Number,
    date: Date
});
```

Convertissez maintenant ce schéma en modèle avec la méthode du même nom :

```
const Light = mongoose.model('Light', lightSchema);
```

Terminez en exportant le modèle :

```
module.exports = Light;
```

Répétez les étapes pour concevoir un modèle *Temperature*.

Exploiter le modèle pour enregistrer les données

Maintenant, importez les modèles *Light* et *Temperature* dans votre contrôleur *icumoov.js* :

```
const Light = require('../models/light');
const Temperature = require('../models/temperature');
```

Programmez une fonction `newData()` qui accepte deux paramètres : les données et le type envoyé (*light* ou *temperature*). Selon le type reçu, instanciez un nouvel objet `Light` ou `Temperature` qui respecte le modèle retenu et enregistrez-le avec la méthode `save()` :

```
function newData(data, type) {
    if (type == 'temperature') {
        var obj = new Temperature({
          temperature: data,
          date: Date.now()
        });
    } else {
        var obj = new Light({
          light: data,
          date: Date.now()
        });
    }
    obj.save();
}
```

Appelez cette fonction dans la méthode `getData()`, au niveau de l'écoute de l'événement `data` :

```
socket.on('data', (data) => {
    const value = data.toString().substr(4);
    newData(value, cmd);
    res.render('data', { type: cmd, data: value });
});
```

Voilà ! Vos routes enregistrent les données dans le nuage *MongoDB Atlas* ! Il reste à les afficher dans votre application.

Étape 5 : afficher la liste des mesures pour chaque modèle

Dans votre contrôleur *icumoov.js*, écrivez une méthode `listLights()` pour récupérer l'ensemble des mesures de luminosité. Pour information, le modèle *Light* intègre un ensemble de méthodes pour dialoguer avec les documents. L'une d'elles se nomme `find()` et permet simplement de renvoyer tous les documents :

```
const listLights = () => {
    return (req, res) => {
        Light.find()
            .then(lights => res.status(200).json(lights))
            .catch(error => res.status(400).json({ error }));
    }
}
```

Faites de même pour le modèle *Temperature* et exportez ces deux nouvelles méthodes :

```
module.exports = {
    listLights,
    listTemperatures,
    getData
};
```

Pour utiliser ces deux méthodes, créez un nouveau routeur `data.js` dans le répertoire *routes* dans lequel vous importerez, au côté des pré-requis, le contrôleur *icumoov.js* :

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

Définissez deux routes : une pour lister les températures et l'autre pour lister les mesures de luminosité.

```
router.get('/temperatures', icumoov.listTemperatures());
router.get('/lights', icumoov.listLights());
```

Exportez le routeur :

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

Et importez-le dans le fichier principal `app.js` :

```
const dataRouter = require('./routes/data');

app.use('/data', dataRouter);
```

Désormais, les adresses *data/lights* et *data/temperatures* renvoient un document JSON avec la liste des mesures enregistrées.

Étape 6 : exploiter les deux collections dans une vue

Dernière phase de ce TD, vous allez récupérer les deux catégories de mesure pour les afficher dans la vue *index.js*. Écrivez une méthode `listData()` qui récupère de manière asynchrone les documents des deux collections pour les renvoyer dans la vue aux clés `lights` et `temperatures` :

```
const listData = () => {
    return async (req, res) => {
        const temperatures = await Temperature.find();
        const lights = await Light.find();
        res.render('index', {
            lights: lights,
            temperatures: temperatures
        });
    }
};
```

Exportez cette nouvelle méthode :

```
module.exports = {
    listData,
    listTemperatures,
    listLights,
    getData
};
```

Maintenant, dans le routeur *index.js*, appelez cette méthode pour la route `/` :

```
router.get('/', icumoov.listData());
```

Votre application est prête !

Pour aller plus loin

Si vous le souhaitez, essayez d'afficher un graphique des mesures grâce à une libraire comme *Chart.js*.

Vous trouverez le code final de ce TD spécial dans le dossier *fin*.