TP12 – Express+Prisma

1. Tools

1.1. Installer node.js

Avant tout, installez la dernière version LTS de node.js : https://nodejs.org/en/download/

1.2. Node modules list

```
nodemon
express
express-generator
serialport
prisma
```

1.3. Mise en place

1.3.1. Commandes de depart

Créer un dossier, aller dedans, ouvrir un terminal, puis :

```
npm i -g nodemon

npm i express express-generator serialport prisma

npx express --view=twig .

npm i
```

1.3.2. Scripts

Ajouter dans le `package.json` :

```
  "scripts": {
    "start": "node ./bin/www",
    "dev": "nodemon -e js,json,twig"
  },
```

La commande suivante servira de raccourci à `"nodemon -e js,json,twig"` :

```
npm run dev
```

1.3.3. Nettoyage facultatif de départ

Supprimer les fichiers et variables inutiles :

- `/public/stylesheets/styles.css`
- `/routes/users.js`
- Dans le fichier `app.js` :

```
- var usersRouter = require('./routes/users');
- app.use('/users', usersRouter);
```

2. Lecture des données

2.1. Créer une interface de base

Dans `/views/index.twig` :

```
{% extends 'layout.twig' %}

{% block body %}
<h1>Valeur lue</h1>
<p>Mesure : <span id="mesure-lue"> {{ mesure }}</span></p>
{% endblock %}
```

2.2. Données Arduino

2.2.1. Données envoyées

Injecter un code simple dans un Arduino, avec un potentiomètre sur A0 (celui d'un MFS est justement sur A0) :

```
#define pot A0
int mesure;
void setup() {
  Serial.begin(9600);
}

void loop() {
  delay(1000);
  mesure = analogRead(pot);
  Serial.println(mesure, DEC);
}
```

2.2.2. Récupérer les données

Voir : https://serialport.io/docs/guide-usage

Dans le fichier `/routes/index.js` :

```
const { SerialPort } = require("serialport");
const { ReadlineParser } = require("@serialport/parser-readline");

// Ouverture d'une communication avec le port COM3 à 9600 bauds
const port = new SerialPort({
    path: "COM3",
    baudRate: 9600,
});

// La lecture série se fera jusqu'à rencontrer un retour à la ligne
const parser = port.pipe(new ReadlineParser({ delimiter: "\n" }));

let lastMesure;

parser.on("data", data => {
    console.log(`Data: ${data}`);
    lastMesure = data;
});
```

À ce stade, on récupère une valeur toutes les secondes dans la console, côté serveur.

2.2.3. Afficher sur la page web

Dans le fichier `/routes/index.js`, changer le `res.render` de façon à passer la variable `lastMesure` à la vue `index` :

```
router.get("/", function (req, res, next) {
    res.render("index", { mesure: lastMesure });
});
```

La fonction `res.render()` permet d'envoyer des variables serveurs aux fichiers `.twig`.

Malheureusement, la page ne se met pas à jour toute seule ; il faut à chaque fois la rafraîchir. Passer `lastMesure` de cette façon au client n'est pas très efficace ; supprimons-le.

```
    res.render("index", { mesure: lastMesure });
```

2.3. Fichier de refresh AJAX

Pour avoir une mise à jour sans avoir à recharger la page, il faut faire une requête de type POST au serveur. Ces requêtes se font depuis un fichier JavaScript.

Créons un fichier `/public/javascripts/refresh.js`. Tout à la fin du `{% block body %}` du fichier `index.twig`, rajouter :

```
<script defer src="/javascripts/index.js"></script>
```

Tant que vous laisser l'attribut defer, vous pouvez même le mettre dans le `<head>` du fichier. Pour ça, il faudrait insérer un bloc du `{% block scripts %}` dans le `<head>` de `layout.twig`.

Pour effectuer la requête POST et recevoir la mesure depuis le serveur, on peut utiliser la fonction `jQuery.ajax()` :

```
// setInterval() se lance toutes les xxx millisecondes
setInterval(() => {
    $.ajax({
        type: "post",
        url: "/api/mesure",
        dataType: "json",
        success: function (response) {
            $("#mesure-lue").text(response);
        },
    });
}, 1000);
```

L'url demandé, `"/api/mesure"`, n'existe pas encore, créons-le dans `/routes/index.js` :

```
router.post("/api/mesure", (req, res) => {
    res.json(lastMesure);
});
```

À présent, la valeur est correctement mise à jour sans qu'on ait à rafraichir la page !

3. Écrire dans la db

3.1. Enregistrer les premières mesures

```
npx prisma init
```

Ajouter/changer dans prisma.schema :

```
datasource db {
  provider = "sqlite"
  url      = env("DATABASE_URL")
}

model Mesure {
  id     Int @id @default(autoincrement())
  mesure Int
}
```

On génère ensuite les modèles dans les fichiers qui créeront la db :

```
npx prisma generate
```

Après la génération, on crée le fichier de db et les tableaux doivent s'y trouver :

```
npx prisma db push
```

Changer la fonction `parser.on` pour qu'elle écrive dans la db en plus de mettre à jour l'affichage :

```
const { PrismaClient } = require("@prisma/client");
const prisma = new PrismaClient();

parser.on("data", async data => {
    console.log(`Data: ${data}`);
    lastMesure = data;
    const mesure = await prisma.mesure.create({
        data: {
            mesure: Number(lastMesure),
        },
    });
    console.log(mesure);
});
```

On aurait pu aussi réaliser l'enregistrement dans la db dans la fonction passée à `router.post("/api/mesure", ...)`. Dans ce cas-là, l'enregistrement dans la db se serait arrêté dès qu'on ferme la page du navigateur. La fonction `parser.on` s'exécute indépendamment de l'état du navigateur, ce qui permet de continuer de prendre des mesures tant que le serveur est up.