ECE PARIS
ÉCOLE D'INGÉNIEURS

Création d'un service Web

2019/2020

APERCU

Dans ce TP vous serez amenés à aborder les bases de NodeJS afin de créer un service Web simplifié.

OBJECTIFS

1. Mettre en place une application NodeJS avec Express
2. Découvrir les routes
3. Communiquer avec la base de données

1. Mettre en place une application NodeJS avec Express

1.1 Première application

Express est un framework minimaliste pour **NodeJS**, il permet de rapidement mettre en place une application web qui écoutera des requêtes HTTP sur un port particulier. Pour mettre en place votre première application **Express** avec **NodeJS**, ouvrez votre fichier **index.js** et insérez le code:

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

app.get('/', function(req, res) {
    res.send('Hello World!');
});

app.listen(3000, function() {
    console.log('Example app listening on port 3000!');
});
```

Sauvegardez votre fichier puis lancez votre application NodeJS avec la commande:

```
node index.js
```

Vous pouvez désormais accéder à votre application à l'adresse http://localhost:3000/

Le code ci-dessus commence avec la commande **require** qui permet d'importer la librairie **express** dans la constante "express". Une application "express" est ensuite créée dans la constante "app".

La fonction **get** de l'application "app", permet de créer une **route**, et d'y assigner une action. Une route correspond à une URL surveillée par l'application qui exécutera une fonction associée à cette route. Ici la fonction associée à la route permet d'envoyer le message "Hello World!" au client qui se connecte sur l'application sur la route "/".

Finalement, la fonction **listen** est ce qui permet de réellement lancer l'application en surveillant les requêtes HTTP sur le port 3000. La fonction qui est juxtaposée au port est une fonction de **callback** qui est appelée une fois que l'application est correctement lancée.

1.2 Réponse JSON

Dans le cadre de notre service Web, il n'est pas très intéressant de retourner un simple texte car nous ne voulons pas avoir une interface graphique mais plutôt échanger de la donnée. Pour cela, nous allons donc utiliser le format texte JSON dans les réponses des actions. Heureusement, Javascript permet une implémentation très simple de ce genre de réponse:

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

app.get('/', function(req, res) {
    let response = { "page": "home" };
    res.send(JSON.stringify(response));
});

app.listen(3000, function() {
    console.log('Example app listening on port 3000!');
});
```

Ici la fonction **stringify** de la classe **JSON** permet de transformer en texte JSON n'importe quelle variable sérialisable.

2. Découverte des routes

2.1 Routes statiques

Comme vu précédemment, il est possible d'associer une fonction à une route afin d'effectuer différentes actions en fonction de l'URL. Pour cela, nous allons tout d'abord mettre en place plusieurs routes qui seront statiques:

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

app.get('/', function(req, res) {
    let response = { "page": "home" };
    res.send(JSON.stringify(response));
});

app.get('/users', function(req, res) {
    let response = { "page": "users" };
    res.send(JSON.stringify(response));
});

app.get('/users/messages', function(req, res) {
    let response = { "page": "messages" };
    res.send(JSON.stringify(response));
});

app.listen(3000, function() {
    console.log('Example app listening on port 3000!');
});
```

Ici nous avons donc trois routes (**/**, **/users** et **/users/messages**), chacune reliée à une fonction différente. Dans ce cas ci, les routes sont donc statiques, c'est à dire que l'URL attribuée sera toujours la même pour accéder à cette page. Pour y accéder:
http://localhost:3000/
http://localhost:3000/users
http://localhost:3000/users/messages

2.2 Routes dynamiques

Dans le cadre d'un service Web, il est très peu intéressant d'avoir des routes statiques car la requête à faire pour le service se trouve dans l'URL et donc cet URL change en permanence.

C’est donc pour cela qu’est utilisé le principe de **paramètre de routage** qui représente une variable de routage dans l’URL. Ce paramètre est indiqué dans la route avec “**:**”.

```
app.get('/users/:id', function(req, res) {
    let response = { "page": "users", "parameters": req.params };
    res.send(JSON.stringify(response));
});
```

Ici nous avons créé un paramètre **id** dans la route, il sera disponible dans l’objet **req.params**.
Exemple: http://localhost:3000/users/420

Remarque: Il n’est plus possible d’accéder simplement à **/users** car tout **paramètre de routage** devient obligatoire. Il faudra donc créer une autre route avec **/users** et sa fonction associée.

Mais le problème de ces URLs est que les variables ne sont pas blindées et qu’il est donc possible de mettre ce que l’on veut comme variable, que ce soit du texte ou des nombres. Ce qui n’est évidemment pas utile dans le cadre de notre service web où nous souhaitons spécifier le type de saisie en fonction de la requête.

2.3 Conditions de routes

Afin de mieux paramétrer les routes, il est possible de mettre en place des conditions sur les variables de routage afin que l’action ne soit qu’appeler si l’URL remplit ces conditions . Nous aborderons différents paramétrage de conditions:

- Expression régulière
- Méthode HTTP

2.3.1 Expression régulière

Les expressions régulières sont un moyen très utile d’extraire de l’information dans du texte. Si vous souhaitez en apprendre plus, je vous invite à lire https://regexone.com/ (en anglais). Par la suite, les expressions régulières qui vous seront présentées seront expliquées.

Afin de blinder un paramètre de routage en tant que chiffre, il est possible de faire ceci avec les expressions régulières:

```
app.get('/users/:id(\\d+)', function(req, res) {
    let response = { "page": "users", "parameters": req.params };
    res.send(JSON.stringify(response));
});
```

Ici, nous obligeons le paramètre **id** a être un chiffre avec l'expression régulière entre parenthèse: **(\\d+)**. L'expression régulière qui permet cela est le **\d+**, il y a un backslash "\" supplémentaire afin d'échapper le premier backslash "\". Dans cette expression, le **\d** correspond à une décimale et le **+** correspond à une occurrence ou plus. Donc cela veut dire que la route est match s'il y a dans la chaîne de caractère de l'id un chiffre ou plus.

Remarque: Toute expression régulière liée à un paramètre de routage devra se placer entre parenthèses à la suite du paramètre. Attention à bien échapper les caractères spéciaux.

2.3.2 Méthode HTTP

Il est aussi possible de blinder une route pour une méthode HTTP particulière, et c'est déja ce que vous faisiez. La méthode **get** de la variable **app** permettait uniquement d'accéder à la route par méthode HTTP **GET**:

```
app.get('/users/:id(\\d+)', function(req, res) {
    let response = { "page": "users", "parameters": req.params };
    res.send(JSON.stringify(response));
});

app.post('/users', function(req, res) {
    let response = { "page": "users", "parameters": req.params };
    res.send(JSON.stringify(response));
});

app.put('/users/:id(\\d+)', function(req, res) {
    let response = { "page": "users", "parameters": req.params };
    res.send(JSON.stringify(response));
});

app.delete('/users/:id(\\d+)', function(req, res) {
    let response = { "page": "users", "parameters": req.params };
    res.send(JSON.stringify(response));
});
```

Ici nous avons donc quatres routes pour les méthodes HTTP **GET**, **POST**, **PUT** et **DELETE**.

Pour tester ces routes, vous devez impérativement utiliser Postman car votre navigateur ne permet que la méthode HTTP **GET** par défaut.

3. Communiquer avec la base de donnée

Maintenant qu'il est possible d'accéder à plusieurs parties de notre service Web grâce aux différentes routes créées, nous allons voir comment communiquer avec notre base de données SQL afin de faire nos différentes requêtes pour récupérer les données requises.

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

let db = mysql.createConnection({
    host: "localhost",
    user: "root",
    password: "root",
    database: "courses",
    port: "3306"
});

app.get('/', function(req, res) {
    db.query("SELECT * FROM test", function(err, result, fields) {
        if (err) throw err;

        let response = { "page": "home", "result": result };
        res.send(JSON.stringify(response));
    });
});

app.listen(3000, function() {
    db.connect(function(err) {
        if (err) throw err;
        console.log('Connection to database successful!');
    });

    console.log('Example app listening on port 3000!');
});
```

Dans ce code, il a été ajouté la librairie javascript **mysql** qui permet de mettre en place une connexion avec la base de données. Tout d'abord, il faut créer une nouvelle connexion afin de préparer les informations de connexion; ceci est fait avec la variable **db** ci-dessus. Remplacez la configuration avec vos informations de connexion à votre base de données. Puis une connection est établie avec la méthode **connect** lors du lancement de l'application. Le callback est donc

appelé lorsque la connexion est établie ou non. **Remarque:** en cas d'erreur, l'application sera quittée avec un message d'erreur grâce au mot-clé **throw**.

Ensuite, tout se passe dans la fonction attachée à la route "/". La requête est effectuée avec la méthode **query**, dans laquelle est transmise la requête SQL. Le callback est appelé une fois que la requête est effectuée. En cas d'erreur, l'application sera aussi quittée avec un message d'erreur grâce au mot-clé **throw**. Sinon le résultat sera retourné dans la réponse JSON.