Un serveur Web d’inscription à un colloque, 2e version

Grâce au module *Connect*, vous allez revoir le script élaboré au précédent TD afin de concevoir une ébauche d’application fonctionnelle. Encore une fois, le résultat sera loin d’être pleinement opérationnel, mais il permet de comprendre les mécanismes mis en place par le cadriciel *Express.js* que vous utiliserez à partir du prochain TD.

Présentation du code de départ

Dans le dossier *debut*, vous trouverez l’ensemble des fichiers sur lesquels vous travaillerez. Les deux fichiers HTML constituent le mini site Web :

- *index.html*, le formulaire d’inscription qui envoie à l’adresse `/inscription` ;
- *confirmation.html*, la page de confirmation de la prise en compte de l’inscription au colloque.

Les deux autres fichiers sont :

- Un fichier au format CSV (*Comma-Separated Values*) qui contient la liste des inscrit·es au colloque avec un enregistrement par ligne au-delà de la ligne d’en-tête. Les informations recueillies sont celles du formulaire, dans l’ordre : nom, prénom, email, statut et affiliation.
- Un fichier *server.js* dans lequel vous interviendrez.

Conception du serveur

Les missions dévolues à votre serveur ne changent pas vraiment par rapport au premier TD. Une fois configuré et prêt à l’écoute, il devra :

1. afficher le formulaire ;
2. comprendre les données envoyées et mettre à jour le fichier *inscriptions.csv* ;
3. rediriger vers la page de confirmation.

Étape 1 : configurer un serveur prêt à écouter

Pour cette première étape, configurez un serveur en écoute sur le port 3000 à l’aide du module *Connect*. Il vous faudra tout d’abord l’installer dans votre répertoire de travail :

```
$ npm install connect
```

Cette commande a pour effet de créer un répertoire *node_modules* avec un sous-répertoire pour chaque module dont dépend *Connect*. Vous pouvez maintenant paramétrer votre serveur facilement dans le fichier *server.js* :

```
// modules requis
const connect = require('connect');
// app
const app = connect();
// port d'écoute
app.listen(3000);
```

Pour l'instant, ce serveur ne fait rien à part se mettre à l'écoute du port 3000, mais vous pouvez déjà penser la suite en ajoutant des *middlewares* pour chacune de ses missions :

```
// charger la page d'accueil
app.use('/home', …);
// mettre à jour le fichier des inscriptions
app.use('/inscription', …);
// charger la page de confirmation
app.use('/inscription', …);
```

À cette étape, il peut être intéressant de faire preuve d'anticipation en prévoyant la création d'un module personnalisé *utils* qui contienne :

- une fonction `loadFile()` pour charger un fichier HTML ;
- une fonction `register()` pour mettre à jour les données.

Modifiez en ce sens le script de votre serveur :

```
const connect = require('connect');
const utils = require('utils');

const app = connect();

// middlewares
app.use('/home', utils.loadFile('./index.html') );
app.use('/inscription', utils.register('./inscription.csv'));
app.use('/inscription', utils.loadFile('./confirmation.html'));

app.listen(3000);
```

Étape 2 : un *middleware* pour afficher un fichier HTML

Avant toute chose, créez un sous-répertoire *utils* dans le répertoire *node_modules*, puis ajoutez à l'intérieur un fichier `index.js` dans lequel vous écrirez le code de votre fonction `loadFile()` qui sera chargée d'afficher un fichier HTML. Prévoyez que cette fonction accepte un paramètre `filename` :

```
function loadFile(filename) {
    // instructions
};
```

N'oubliez pas qu'elle devra retourner une fonction de *callback* à l'application avec les deux objets `request` et `response` :

```
function loadFile(filename) {
    return (req, res) => {
        // instructions
    };
};
```

Pour le reste, votre fonction s'attache à créer un flux en lecture vers le fichier `filename` et à le connecter grâce à un pipeline à un flux en écriture :

```
const fs = require('fs');
const stream = require('stream');

function loadFile(filename) {
  return (req, res) => {
    const file = fs.createReadStream(filename);
    stream.pipeline(file, res, (error) => {
      if (error) console.log(error);
    });
  };
};
```

En dernier lieu, comme vous travaillez à l'intérieur d'un module, vous devrez exporter votre fonction :

```
module.exports.loadFile = loadFile;
```

Étape 3 : un *middleware* pour mettre à jour le fichier des inscriptions

Sur le même principe, ajoutez une fonction `register()` dans le fichier *index.js* :

```
function register(filename) {
    return (req, res, next) => {
        // instructions
        next();
    };
};
```

La seule différence, comme vous le remarquez, c'est que le *middleware* ne s'arrête pas là : une instruction `next()` est adressée au serveur afin de lui indiquer de se mettre en attente d'autres instructions. Si vous observez le code de votre script *server.js*, vous comprenez que c'est exactement ce qui est prévu lorsque le client visite l'URL */inscription* : d'abord vous enregistrez ses informations dans le fichier des inscriptions puis vous lui affichez le message de confirmation.

Pour en revenir à la fonction `register()`, le code est finalement le même que celui du TD 1 :

```
function register(filename) {
    return (req, res, next) => {
        req.on('data', (chunk) => {
        let data = String();
        let search = String();
        for (let param of chunk.toString().split('&')) {
          const q = param.split('=');
          search += `${ decodeURIComponent(q[1]) }\t`;
        }
        data = `${ search.slice(0, -1) }\n`;
        fs.appendFile('./inscriptions.csv', data, (error) => {
          if (error) console.log(error);
        });
      });
      next();
```

```
    };
};
```

Il ne vous reste plus qu’à exporter la fonction :

```
module.exports.register = register;
```

Et à tester en lançant le serveur :

```
$ node server.js
```

Vous trouverez le code final des documents HTML et JavaScript dans le dossier *fin* de ce second TD.