Développement de serveur web nodeJS

Partie 2

Table des matières

Déstructuration d'objet et de tableau

L'une des nouvelles fonctionnalités les plus pratiques de javascript est la destruction d'objet et tableaux.

En utilisant l'opérateur de spreading "…" et les caractères "{}" il est possible de décomposer un objet ou un tableau. C'est à dire que chaque variable ou valeur sera affecté dans un nouvel objet.
Ainsi il est possible de copier toutes ou certaines valeurs d'objet ou de tableau simplement comme le montre la saisie d'écran ci-dessous.

```
JS destructuration.js > ...
1   const monTableau = [1, 2, 3, 4, 5];
2   //affichera "Array [1, 2, 3, 4, 5]"
3   console.log(monTableau);
4   //affichera "1 2 3 4 5"
5   console.log(...monTableau);
6
7   const monSecondTableau = [...monTableau, 6, 7, 8, 9, 10];
8   //affichera "Array [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]"
9   console.log(monSecondTableau);
10  //affichera "1 2 3 4 5 6 7 8 9 10"
11  console.log(...monSecondTableau);
12
13  const monObjet = { couleur: 'vert', poids: 10, valeur: 15 };
14  //affichera "Object { couleur: 'vert', poids: 10, valeur: 15 }"
15  console.log(monObjet);
16  const { couleur, valeur } = monObjet;
17  //affichera "Mon objet est vert et vaut 15 euros"
18  console.log(`Mon objet est ${couleur} et vaut ${valeur} euros`);
19
20  const monSuperObjet = { ...monObjet, origine: 'France', type: 'aliment' };
21  //affichera "Object { couleur: 'vert', poids: 10, valeur: 15, origine: 'France', type: 'aliment' }"
22  console.log(monSuperObjet);
```

Si l'opérateur = fait une affectation par référence, l'opérateur de spread lui fait une recopie par valeur. Ainsi si l'on modifie une valeur d'un objet affectée par l'opérateur spread, la valeur de l'objet d'origine ne sera en rien modifié.

```
const monSecondObjet = { ...monObjet };
const monTroisiemeObjet = monObjet;
monObjet.couleur = rouge;
//affichera rouge, vert, rouge
console.log(monObjet.couleur, monSecondObjet.couleur, monTroisiemeObjet.couleur);
```

Asynchronicité

Définition

Asynchrone est l'antonyme de synchrone dont la définition est : *Se dit de phénomènes, de machines, de tâches, de signaux ou d'informations dont les rythmes propres sont égaux, multiples ou sous-multiples. (Larousse)*

Lorsqu'une page HTML est généré, un navigateur reçoit les fichiers nécessaires de façon asynchrone. Cela permet de ne pas attendre qu'un fichier soit fini de télécharger afin de commencer à télécharger le prochain.

De la même manière, **pour qu'un serveur puisse être interrogé simultanément par plusieurs clients, il est indispensable que les tâches du serveur soient asynchrones.**

Javascript et la notion d'asynchrone

Lorsque la notion d'asynchrone est apparu en Javascript il n'était pas possible de l'utiliser da la même façon sur tous les navigateurs. La solution la plus courante était d'utiliser un framework tel jQuery ou Prototype JS pour éviter de devoir gérer plusieurs versions de navigateurs.

```
1 | $.when( $.ajax( "test.aspx" ) ).then(function( data, textStatus, jqXHR ) {
2 |   alert( jqXHR.status ); // Alerts 200
3 | });
```

En utilisant jQuery, le code ci-dessus permet en trois lignes de télécharger le fichier *test.aspx* et d'afficher le code de résultat HTTP.
Même si cela peut sembler trivial, sans de nombreuses lignes de code étaient nécessaires afin

Bien heureusement, les navigateurs respectent désormais les normes W3C vis vis des appels asynchrones, il n'est donc plus nécessaire d'utiliser ce type de framework pour gérer cela ou de gérer différents navigateurs au sein de votre code.

setTimeout et setInterval

Une façon de basique de lancer du code non bloquant de manière non-séquentielle est d'utiliser les fonctions setTimeout et setInterval
Ces fonctions permettent de lancer du code à un instant précis sans en attendre la fin de son exécution.

Dans l'exemple ci-dessous nous afficherons quelque chose dans la console toutes les secondes.

```
for (let i = 0; i < 10; i++) {
  // setTimeout sert à lancer une fonction au bout de N milliseconds
  // Son premier argument est la fonction à appeler
  // Le second argument correspond au temps à attendre
  setTimeout(() => console.log(i), 1000 * i);
}
console.log("Chronomètre lancé!")
```

Note: ce code est disponibles dans le fichiers exemples.js

```
Chronomètre lancé!
0
1
2
3
4
5
6
7
8
9
```

Comme le montre le résultat ci-contre, nous affichons la valeur de *i* après avoir affiché “Chronomètre lancé”

Si setTimeout permet de lancer du code de façon unitaire, setInterval permet d’exécuter le même code plusieurs fois à un intervalle donné.

Promesses

L'objet **Promise** (pour « promesse ») est utilisé pour réaliser des traitements de façon asynchrone. Une promesse représente une valeur qui peut être disponible maintenant, dans le futur voire jamais. (source: MDN)

Users > trambz > git > imie > code-examples > JS promises.js > ...

```
//renvoie YOU WIN si le résultat du random est superieur à 0.5
//sinon la promesse est rejetée
const maRandomPromesse = new Promise((resolve, reject) => {
  // réaliser une tâche asynchrone et appeler :
  const randomNumber = Math.random();
  if (randomNumber > 0.5) {
    resolve({ message: 'YOU WIN!!!', res }); // si la promesse est tenue
  } else {
    reject({ message: 'YOU LOSE...', res }); // si elle est rompue
  }
});
```

Comme on peut le voir sur la saisie d’écran on peut voir qu’une promesse permet de gérer le résultat et le retour de l’appelle asynchrone.

Ainsi on utilise le mot *resolve* si le résultat du traitement est bon, sinon nous pourrons rejeter la promesse et le code placé dans la partie *then* de l'appelant ne sera pas appelé.

Si la promesse est résolu le code appelant exécutera la fonction then, sinon si la promesse est rejeté on exécutera le code déclaré dans *catch* comme montré ci-dessous.

```
const res = maRandomPromesse()
  .then(res => console.log('o/', res))
  .catch(res => console.log(':(', res))
  .finally(() => console.log('thanks for playing'));
```

On notera que si la promesse soit résolue ou rejetée, il est possible d'exécuter systématiquement du code en utilisant *finally*.

Bases de données et NodeJS

NodeJs est communément utilisé avec des bases de données puisque c'est un langage serveur.
Il est extrèmement simple de connecter n'importe quelle type de base de données

Mais il est encore plus simple de connecter Express avec une base de données.
La liste de type de base de données disponible est décrite sur cette page:
https://expressjs.com/fr/guide/database-integration.html

Une fois MAMP, LAMP ou WAMP installé nous allons créer une base de données simple et utiliser un connecteur nodeJS afin de pouvoir y accéder depuis express.

Pour ce faire nous allons installer le paquet npm myqsl

```
npm install mysql
```

Puis, une fois installé nous allons nous connecter à la base avec le code suivant dans notre fichier *App.js*.

```
export const mySqlConnection = mysql.createConnection({
  //ONLY FOR MAMP AND MAC OS X USERS!!!
  socketPath: '/Applications/MAMP/tmp/mysql/mysql.sock', //path to mysql sock in MAMP
  host: 'localhost',
  user: 'trambz',
  password: 'toto',
  database: 'cats'
});

mySqlConnection.connect(err => {
  if (err) throw err;
  else {
    app.listen(3000, () => {
      console.log('Server listening...');
    });
  }
});
```

Nous allons créer une connexion avec différents paramètres précis comme le mot de passe, le nom de la base de données ou encore la méthode de connexion.

Il y a un paramètre ci-dessus spécifique à Mac OS X, si vous utilisez windows ou Linux il vous faudra supprimer la ligne socketPath ou spécifier le numéro de port utilisé par votre serveur mySql

Afin de pouvoir l'utiliser dans d'autre fichier nous exportons cette connexion.

Une fois créé nous nous connectons à base de données.
Si la connexion échoue, le paramètre err sera défini, dans ce cas nous passerons dans le if et n'exécutrons pas le code qui suit.

Sinon: nous pouvons lancer le serveur!

Nous allons maintenant utiliser cette fonction dans une route POST et insérer un élément en base de données.

Si la fonction se passe correctement nous retournerons le numéro de l'ID du nouvel enregistrement.

Il faudra dans un premier temps importé l'objet de connexion créé dans le fichier app.js.
Une fois importé nous utiliserons la fonction *query.*

Pour créer l'enregistrement nous utiliserons les données passés dans le body de la requête post au format JSON.

```
catsRouter.post('/', (req, res, next) => {
  const { name, url } = req.body;
  if (!name || !url) {
    res.status;
  }
  mySqlConnection.query(
    `INSERT INTO \`cats\` (\`name\`, \`url\`) VALUES ('${name}', '${url}');`,
    (err, rows, fields) => {
      if (err) throw err;
      res.json({ id: rows.insertId });
      res.end();
    }
  );
});
```

Dans un premier temps nous récupérons les informations contenues dans le body.
Attention il vous faudra installer le paquet body-parser et le configurer comme montré ci-dessous pour pouvoir utiliser cette fonctionnalité.

Cette configuration doit être faite lors de la création de l'application. Seule une ligne suffira pour cela (en ayant importé le module bien sur)

```
// parse application/json
app.use(bodyParser.json());
```

Références

Déstructuration d'objet et de tableau
https://dev.to/quantumsheep/all-you-need-to-know-about-destructuring-in-javascript-1hla

Promesses
https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Promise

Asynchronicité en JS
https://www.pluralsight.com/guides/introduction-to-asynchronous-javascript
https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Promise