Chapitre 3
Module et développement objet

1. Modules

1.1 Les causes

Jusqu'ici, nous pouvons structurer notre code JavaScript en fichiers indépendants. Supposons que nous ayons besoin régulièrement dans nos projets d'effectuer un type de calcul. Il est donc préférable de concentrer ces calculs dans un fichier JavaScript indépendant puis de le charger au besoin dans notre page HTML.

Par exemple, nous avons ici un fichier JavaScript `calcul.js` contenant un calcul élémentaire d'addition.

```
function calcul(a,b) {
    return a+b;
}
```

Si nous voulons maintenant utiliser notre calcul avec ce code, nous pouvons simplement charger notre calcul à partir de notre site web pour afficher un résultat dans la console du navigateur.

```
<html>
    <head>
        <script src="calcul.js"></script>
        <script>
```

```
            console.log(calcul( 10,20 ) );
        </script>
    </head>
    <body></body>
</html>
```

Ou bien nous pouvons avoir un ensemble de fichiers JavaScript qui vont travailler avec `calcul` en évitant d'ajouter du code directement dans la page HTML. Nous aurons alors un nouveau fichier `app.js` contenant par exemple :

```
function monApp() {
   console.log( calcul( 10, 20 ) );
}
document.addEventListener("DOMContentLoaded", () => {
   monApp();
});
```

Nous avons une fonction `monApp` qui effectue un traitement mais au chargement de notre page HTML (qui affiche « `30` » dans la console). En effet, nous ne voulons généralement pas que le code JavaScript s'exécute avant que notre page HTML soit complètement chargée par le navigateur. Pour cela, nous allons ajouter une instruction `document.addEventListener` qui nous sert à exécuter une fonction (ici `monApp`) lorsque qu'un événement `DOMContentLoaded` (le document est bien chargé et converti en document contenant des nœuds *DOM*). Ainsi, nous n'avons plus besoin d'intégrer de code JavaScript directement dans notre page.

Si cette technique est généralement bonne, elle a un inconvénient si notre projet est important et utilise de nombreux fichiers JavaScript. Nous risquons d'avoir plusieurs fois des méthodes ou des variables de même nom créant ainsi un code difficile à maintenir. Imaginons par exemple que nous importions un nouveau fichier JavaScript `calcul2.js` qui contiendrait le code suivant :

```
function calcul(a,b) {
    return a-b;
}
```

La page HTML finale est donc :

```
<html>
    <head>
        <script src="calcul.js"></script>
        <script src="calcul2.js"></script>
        <script src="app.js"></script>
    </head>
    <body>
    </body>
</html>
```

Nous n'aurons plus le même résultat (ici `-10` dans la console !) car l'interpréteur JavaScript ne va retenir que la dernière méthode de même nom déclarée et donc celle qui effectue une soustraction.

Donc on le constate, ajouter un fichier JavaScript peut compromettre l'exécution d'un code très facilement. Et si cela reste gérable avec un petit projet, si vous avez des centaines, des milliers de fonctions, vous n'arriverez plus à gérer correctement ce problème, votre code deviendra instable.

1.2 Module

Pour régler ces problèmes de collision, la version 2015 de JavaScript (ECMAScript 2015) introduit la notion de module. Un module est déjà un fichier JavaScript où tout le contenu reste visible localement par défaut. Il n'y a plus de visibilité globale comme pour notre méthode `calcul`. Cela sera à vous d'indiquer ce que vous voulez mettre à disposition et ce qui doit rester « caché » dans le module.

1.2.1 Export

Comme tout est en « local » dans un module (donc invisible en dehors du module), nous allons devoir préciser les parties que nous souhaitons rendre utilisables pour les utilisateurs de notre module en les préfixant par export.

Notre premier module de calcul `calcul.js` contient donc maintenant une fonction `calcul` exploitable après importation du module.

```
// Module de calcul
export function calcul(a,b) {
    return a+b;
}
```

Remarque

Nous allons stocker nos modules dans un répertoire de même nom afin d'éviter toute confusion avec les autres formes de scripts.

1.2.2 Import

Maintenant, pour que notre module application `app.js` puisse fonctionner, nous allons devoir aussi « importer » ce qui nous intéresse.

```
// Module application
import { calcul } from './calcul.js' function monApp() {
    console.log( calcul( 10, 20 ) );
}
monApp();
```

Remarque

Les fichiers `app.js` et `calcul.js` étant au même niveau, nous nous préfixons les répertoires d'accès aux modules respectifs par "./".

Remarque

Il est aussi important de noter que nous n'avons plus besoin de vérifier si notre page HTML est bien complètement chargée, c'est automatique. Un module ne sera chargé que si la page HTML a déjà été parsée.

Pour notre intégration dans une page HTML (test.html), nous allons faire ces déclarations :

```
<!DOCTYPE html>
<html>
    <head>
      <script src="calcul2.js"></script>
      <script type="module" src="modules/app.js"></script>
    </head>
```

```
    <body>
    </body>
</html>
```

Nous avons laissé explicitement la mauvaise fonction de calcul (celle qui fait la soustraction). Elle est donc toujours chargée à partir de `calcul2.js` et est donc « globale » et visible partout. Mais ce qui va changer, c'est le code de notre application `app.js` qui est un module et pour cela nous allons préciser que le type de notre fichier JavaScript correspond à celui du module par l'attribut `type="module"`.

Notre module `app.js` est exécuté automatiquement après le chargement/parsing de la page HTML. Nous l'avons placé explicitement dans un sous-répertoire `modules` pour éviter tout mélange de code.

Si vous chargez votre page HTML dans un navigateur, vous risquez d'avoir un message d'erreur dans la console de type :

```
....html:1 Access to script at '…/app.js' from origin 'null' has
been blocked by CORS policy.
```

Ceci vient du fait que vous pouvez charger un module n'importe où, le navigateur veut éviter qu'un accès ne se fasse en local pour des questions de sécurité (un code malveillant pourrait capturer/transmettre les valeurs d'un champ). Donc pour tester votre page, il va falloir exécuter votre code avec un serveur web.

Il existe plusieurs solutions pour cela, mais pour rester dans la thématique et être cohérent, nous allons passer par l'application *Node.js* qui autorise la création d'un serveur web local en JavaScript.

Tout d'abord, vous pouvez installer *Node.js* à partir du site https://nodejs.org/fr, l'installation ne pose pas de difficulté. Ensuite, nous allons installer un mini serveur Web avec cette commande dans un terminal (disponible sous Windows avec `cmd.exe`) et en vous positionnant dans votre répertoire de travail (commande `cd`) :

```
npm install -g http-server
```

Remarque

Node.js est disponible sur différentes plateformes, vous aurez des commandes similaires quel que soit votre système d'exploitation.

`npm` est un gestionnaire de paquets propre à *Node.js* et qui vous sert à installer et utiliser un module *JavaScript*. Grâce à cela, vous pourrez maintenant exécuter un serveur local avec la commande `http-server`. Par défaut, l'accès à vos pages web se fera par le port 8080, donc en utilisant cette URL http://127.0.0.1:8080/test.html

Une fois exécutée, vous devriez voir dans la console la réponse `30`. Donc `calcul2.js` qui contient pourtant une fonction `calcul` est totalement ignorée et ne gêne plus l'exécution de notre code.

1.2.3 Export/import multiple

Nous pouvons exporter autant d'éléments que nous le souhaitons. Cela concerne les fonctions, variables ou classes. Par exemple, nous allons ici étendre notre module `calcul.js` pour proposer une classe `Poisson` (nous aborderons cette syntaxe par la suite).

```
// Module de calcul et classe Poisson

export function calcul(a,b) {
    return a+b;
}

export class Poisson {
    constructor(espece) {
        this.espece = espece
    }
    getEspece() {
        return this.espece;
    }
}
```

À l'usage, si nous voulons utiliser notre nouvelle classe `Poisson` dans notre module `app.js`, nous aurons alors :

```
import { calcul,Poisson } from './calcul.js'
function monApp() {
    const p = new Poisson( "carpe" );
    console.log( calcul( 10, 20 ) );
    console.log( p.getEspece() );
}
monApp();
```

Nous avons bien précisé entre accolades vouloir la fonction `calcul` et la classe `Poisson`.

Ce qui n'est pas explicitement exporté reste local au module (donc invisible en dehors).

Nous n'importons que notre module `calcul` ici, mais nous pouvons aussi importer n'importe quel module supplémentaire, l'important étant de pouvoir spécifier la localisation de chaque module.

1.3 Modules CommonJS

Les modules *CommonJS* sont un peu les ancêtres des modules JavaScript. Ils sont principalement utilisés par l'application *Node.js*. Ils ne sont pas plus compliqués à comprendre que les modules JavaScript.

Il existe deux manières d'exporter une variable/constante, une fonction, un objet ou une classe.

La première forme d'exportation est plus simple et permet à l'importation d'avoir directement la bonne valeur avec `module.exports`. Dans l'exemple ci-dessous, nous reprenons notre fonction de `calcul` que nous exportons grâce à l'objet `module` et la propriété `exports`.

```
function calcul(a,b) {
    return a+b;
}

// Export de la fonction calcul
module.exports = calcul;
```

Si maintenant nous voulons utiliser notre module dans un module *CommonJS* `app.js`, nous passerons par la fonction `require` avec en argument le chemin vers notre module de calcul.

```
const calcul = require( "./calcul.js" );
console.log( calcul( 20, 10 ) );
```

La fonction de calcul est la valeur retournée par la fonction `require`, nous n'avons plus qu'à effectuer un test de calcul et l'afficher dans la console (qui avec *Node.js* est en réalité le terminal Windows).

Pour cela, nous n'avons plus qu'à demander à `Node.js` d'exécuter notre module principal avec :

```
node app.js
```

Ce système est pratique si nous n'avons qu'une seule valeur à exporter mais dans le cas contraire, nous allons simplement indiquer grâce à la propriété `module.exports` (qui n'est qu'un objet vide) chaque valeur à disposition.

Dans l'exemple ci-dessous, nous avons une variante de notre module de calcul dans laquelle nous « exportons » trois valeurs : une fonction `calcul`, une classe `Poisson` et une constante `PI`.

```
function calcul(a,b) {
   return a+b;
}

class Poisson {

}

const PI = 3.14;

module.exports.calcul = calcul;
module.exports.Poisson = Poisson;
module.exports.PI = PI;
```

Remarque

Il est possible de ne pas mettre `module` mais d'utiliser `exports` directement, cependant cela peut créer aussi des conflits de nom.

Maintenant, pour importer toutes nos valeurs dans un module d'application, nous pourrions écrire :

```
const { calcul, PI, Poisson } = require( "./calcul2.js" );

const carpe = new Poisson();
console.log( calcul( PI, PI ) );
```

La fonction `require` retourne un objet dont les propriétés correspondent aux éléments exportés. On utilise pour simplifier une déstructuration qui consiste à pouvoir associer à des propriétés de même nom des constantes et même variables (en général moins sous cette forme). Dans notre module d'application, nous récupérons bien les valeurs associées pour `calcul`, `PI`, `Poisson`.

2. Développement Objet

2.1 Concepts Objets

2.1.1 L'objet

Un objet est un concept abstrait pour désigner une valeur existant, caractériser des propriétés (son état) et des méthodes qui s'y appliquent (ses services). C'est une forme de structure de données qui garantit une homogénéité du code au sein de briques élémentaires et réutilisables. L'idée étant de ne plus avoir d'un côté les données et de l'autre les méthodes, mais de ne créer que des blocs cohérents combinant des données et des méthodes.

Remarque

Un objet se base sur le même principe qu'un objet dans tout environnement, c'est-à-dire le fait qu'il ait un état et que des actions soient possibles sur ce dernier. Prenons l'exemple d'un Rubik's cube, il possède un état constitué du positionnement de chaque case de couleur sur chaque face, mais il possède également des actions pour faire diverses rotations.