Spoke

Module pattern en JavaScript

12 août 2019

Table des matières

Aujourd'hui, le JavaScript est devenu incontournable surtout pour les développeurs web. Cependant, il reste encore trop méconnu et mal utilisé par la plupart des programmeurs, qui ne prennent pas vraiment le temps de l'étudier. Nombreux sont les projets web où le JavaScript est codé "comme on peut" dans un coin de la page HTML voire directement dans les éléments HTML.

Ce design pattern, que je vais vous présenter, vous permettra d'avoir une approche différente sur la façon de structurer le JavaScript dans vos projets afin d'améliorer l'évolutivité et la maintenabilité de votre code.

Bonne lecture !

i Il est recommandé d'être à l'aise avec JavaScript (et d'avoir de bonnes notions objets) afin d'aborder sereinement ce tutoriel.

1. Rappels

Je vous fais un petit récapitulatif des éléments que vous devez connaitre pour continuer la lecture.

1.0.1. Portée des variables

Les variables déclarées avec le mot-clé **var** dans une fonction, ne sont pas accessibles en dehors de celle-ci.

```
1 function maFonction(){
2     var a = 10;
3
4     //a est accessible ICI
5 }
```

```
6 //a n'est pas accessible ICI
```

Il est vivement conseillé de toujours utiliser le mot-clé **var**, si vous ne le faites pas la variable devient *globale* !

```
1 function maFonction(){
2     a = 10; //ce qu'il ne faut PAS FAIRE !
3 }
4
5 maFonction();
6 console.log(a); // affichera 10 !! a est global
```

Les variables globales sont à éviter. D'une part, elles posent des problèmes de sécurité car rien ne vous garantit qu'elles ne seront pas écrasées ou modifiées par un autre script. Et d'autre part pour des raisons de performances, plus le **scope** (espace de définition) est loin plus le temps d'accès est long.

1.0.2. Fonctions dans des fonctions

En JavaScript les fonctions sont considérées comme des objets **Function**. Le langage supportant les *closures* on peut déclarer des fonctions dans des fonctions comme ceci :

```
1 function maFonction(){
2     //création d'une fonction à portée privée
3     function test(){
4
              console.log('Je suis une fonction contenue dans maFonction');
5     }
6
7     b();
8 }
9 maFonction();
```

Dans cet exemple il est important de noter que la fonction **test()** n'est pas accessible en dehors de maFonction(). Elle est *encapsulée* dans l'objet maFonction.

1.0.3. Fonctions anonymes

JavaScript est un langage objet et donc les variables peuvent pointer sur des objets. Nous avons vu qu'une fonction était un objet, donc il est possible de faire ceci :

```
var bonjour = function(){
    console.log("Je suis une fonction anonyme");
};

bonjour(); //affichera 'Je suis une fonction anonyme' dans la
    console
```

Ce code montre la création d'une variable bonjour qui se voit affecter une référence vers une *fonction anonyme* qui est déclarée tout de suite après. L'appel de la fonction se fait avec le nom de la variable suivi de parenthèses. On peut bien évidemment passer des arguments à la fonction.

Si la variable *bonjour* est détruite, ou mise à **undefined**, et qu'il n'existe aucune autre variable détenant la référence vers la fonction, celle-ci sera nettoyée par le *garbage collector*. Vous noterez que la fonction se termine par un ; car c'est une affectation et non une simple déclaration.

2. Le pattern module

Nous y voilà ! Le pattern module est une manière d'encapsuler du code dans un **package ou namespace** tout en permettant si besoin, un accès extérieur à certaines propriétés/fonctions. Nous ne verrons pas dans ce tutoriel comment instancier ce module.

Sans plus attendre voilà à quoi il ressemble :

```
var MODULE = (function(){
    var self = {};
        var variablePrivee = 10;

    self.attributPublic = "bonjour";

        function methodePrivee() {
            console.log('Je suis encapsulée !');
        }

        self.methodePublique = function(){
        console.log('Je suis accessible !');
        };

        return self;
})();
```

Quelques explications s'imposent...

Premièrement, vous pouvez remarquer qu'une fonction est créée et qu'elle est passée dans une variable nommée **MODULE**. Rien de nouveau, nous l'avons vu plus haut.

Oui, mais... pourquoi toutes ces parenthèses autour de la fonction ?

On appelle ça une **self-invoking function**, ce qui peut se traduire en français par *fonction qui se lance automatiquement (auto-exécution)*. En résumé, la fonction est appelée automatiquement au moment de sa création. Cela permet d'initialiser entièrement les attributs qu'elle contient.

À l'intérieur de notre fonction on peut voir qu'une variable **self** est déclarée. Cette variable est TRÈS importante, elle est la clef du problème d'encapsulation que vous avez soulevé !

Le nom '**self**' est un nom que j'utilise dans mes projets, vous pouvez bien évidemment nommer cette variable comme vous le souhaitez !

En fait, **self** est un objet (ou ensemble) qui ne contient rien à sa déclaration. Pendant la création du module, on y met plusieurs choses : *attributPublic* et *methodePublique*. Ces deux éléments appartiennent à l'objet **self**.

A la fin de la création du module, on retourne **self**.

Mais du coup, la variable MODULE ne contient pas vraiment une fonction mais le contenu de la variable self ?!

Oui ! Enfin, pas tout à fait, elle contient la référence vers la variable **self**. Et c'est pourquoi on peut faire ceci :

```
1 var MODULE = /* code ci-dessus */
2
3 MODULE.methodePublique(); // affiche 'Je suis accessible'
4
5 console.log(MODULE.attributPublic); // affiche 'bonjour'
```

Tout ce qui est ajouté à **self** est disponible en dehors du module, c'est *public*. Tout ce qui n'y est pas est local au module et n'est pas accessible en dehors, c'est *privé*.

Mais, le reste du module est détruit alors ?

Pas du tout, comme la variable **MODULE** contient une *référence* vers **self** et que **self** appartient au module, tout le contenu du module persiste en mémoire. Rien n'est détruit par le *garbage collector* ! On peut donc utiliser des fonctions privées (ou publiques) dans des fonctions publiques par exemple et inversement :

```
var MODULE = (function(){
    var self = {};

    function methodePrivee() {
            console.log('Je suis encapsulée !');
        }

        self.methodePublique = function(){
        methodePrivee();
        self.methodePubliqueDeux()
        };

    self.methodePubliqueDeux = function(){
        console.log('Je suis publique !');
    };

        return self;
})();

MODULE.methodePublique(); // affiche 'Je suis encapsulée' puis 'Je
    suis publique'
```

3. Une fonctionnalité, un module

D'une manière générale, il faut éviter au maximum de mettre tout votre code dans l'espace global ou même dans un seul module. C'est comme si les maisons n'avaient qu'une seule grande pièce et que tout était mis un peu n'importe où. Découper son espace en sous-espace est une bonne pratique qu'il faut respecter et cela, quelque soit le langage.

C'est pourquoi il est conseillé de créer un module par fonctionnalité ou ensemble de fonctionnalités qui ont un lien entre elles. Voilà un exemple de ce qui peut se faire :

```
//PREMIER MODULE

// Gestionnaire de logs
var Logger = (function(){
    var self = {};
    var logger = new Array(); // attribut privé

    //méthode privée
    function displayLog(log){
        console.log(log.module + " : " + log.message);
    }


```

```
14      //methodes publiques
15
16          self.log = function(moduleName, msg){
17          var log = {module: moduleName, message: msg};
18          displayLog(log);
19          logger.push(log);
20          };
21
22      self.showAll = function(){
23          for(var i = 0; i < logger.length; i++)
24              displayLog(logger[i]);
25      };
26
27          return self;
28  })();
29
30
31  // DEUXIEME MODULE
32
33  // Gestionnaire d'un div particulier (inscription, connexion,...)
34  var DivManager = (function(){
35      var self = {};
36      var div = undefined;
37
38      // si l'on souhaite retarder la récupération de l'élément on
            peut
39      // faire les opérations dans une fonction qui sera appelée dans
            le code.
40      // C'est un choix personnel, mais vous pouvez tout à fait
            déclarer directement
41      // la variable var div = document.getElementById(...) au
            dessus.
42      self.init = function(){
43          div = document.getElementById('monDiv');
44          div.onclick = onClick;
45      };
46
47      function onClick(){
48          //utilisation d'un autre module :)
49          Logger.log("DivManager", "Le bloc a été cliqué");
50      }
51
52
53          return self;
54  })();
55
56
57  DivManager.init(); //on appelle la fonction du module pour
        initialiser le click
```

On constate qu'il y a deux modules, un qui se charge uniquement de la gestion des logs (enregistrement, affichage etc.) et un autre qui est dévoué à la gestion d'un élément HTML (dans notre cas de la récupération d'une div et de l'ajout d'un écouteur d'événement).

4. Sous-modules

Pour chaque fonctionnalité il est conseillé de créer un module, mais généralement le module peut grandir très vite et devenir rapidement difficile à relire pour des programmeurs externes. C'est pourquoi il est primordial d'utiliser des **sous-modules** quand le cas se présente.

Les sous-modules sont des *sous-ensembles* du module principal. L'écriture est la suivante :

```
1  var ModuleParent =  (function(){
2      /* module */
3  })();
4
5
6  ModuleParent.SousModule = (function(){
7      /* sous module enfant */
8  })();
```

On déclare un nouvel élément dans ModuleParent (qui est un objet ou ensemble) qui contient lui-même un module et ses propriétés. Le sous-module n'a pas accès aux fonctionnalités de l'élément parent, car les deux ont leur propre *espace de noms*. Si l'enfant doit communiquer avec le parent, il faut que les méthodes et attributs soient publics. **Ce n'est pas de l'héritage !** C'est uniquement un découpage du code en fonctionnalités.

Un exemple parlant de communication parent/enfant :

```
1  // Gestionnaire d'un div particulier (inscription, connexion,...)
2  var DivManager = (function(){
3      var self = {};
4      self.div = undefined; //notez le changement, div appartient à
           self.
5
6      self.init = function(){
7          self.div = document.getElementById('monDiv');
8          DivManager.Events.init();
9      };
10
11         return self;
12 })();
13
14
15 DivManager.Events = (function(){
16     var self = {};
```

```
17
18      self.init = function(){
19          // on récupère div qui est public et on lui affecte un
                écouteur (click)
20          DivManager.div.onclick = onClick;
21
22          //ici je peux ajouter d'autres écouteurs
23      };
24
25      function onClick(){
26          console.log("J'ai été cliqué !");
27      }
28
29      // je peux ajouter ici les fonctions spécifiques à de nouveaux
            écouteurs
30
31
32      return self;
33  })();
34
35
36
37  DivManager.init(); //on appelle la fonction du module pour
        initialiser l'ensemble
```

On peut voir qu'on a externalisé le code qui gère les évènements sur la div. On évite ainsi de surcharger de fonctionnalités DivManager et le code s'en trouve plus lisible. Il est également plus facile de modifier le code, même 6 mois plus tard, car les noms sont explicites et précis.

Veillez à choisir correctement les noms des modules et des sous-modules. Ils sont essentiels pour facilement vous y retrouver. Également, prenez l'habitude de mettre une majuscule pour nommer les modules, cela vous permettra de différencier rapidement un module d'une fonction.

Attention à ne pas déclarer un sous-module **AVANT** un module. Le module parent (la variable) serait non défini.

Pour finir cette partie, je vous conseille de mettre chaque module (et ses sous-modules) dans des **fichiers indépendants** nommés par le nom du module. D'une part cela vous évitera d'avoir trop de code au même endroit et d'autre part vous verrez facilement si un nom de module existe déjà. Ceci est valable en développement, en production il peut être intéressant, selon vos besoins, de réduire le nombre de fichiers et de les compresser.

5. Un module parent

La multiplicité des bibliothèques, frameworks ou extensions JavaScript au sein d'un même projet peut entraîner un problème très important : l'écrasement des espaces de noms. Ceci arrivera inévitablement si votre module se nomme pareil qu'une bibliothèque que vous utilisez (ou pire encore, vous écraserez la bibliothèque). Pour éviter cette situation, il est vivement conseillé d'utiliser un module parent qui contiendra l'ensemble de vos modules. Celui-ci aura un nom unique qui pourra facilement être modifié s'il entre en conflit avec d'autres extensions.

Le fonctionnement est similaire aux *sous-modules*. L'idée est simplement de créer un module parent qui englobera l'ensemble des modules enfants. Le chargement de vos fichiers pouvant intervenir dans un ordre aléatoire, il faut faire attention à bien vérifier l'existence du parent avant de tenter d'y ajouter un enfant. Voilà comment procéder pour un parent se nommant **Application** :

```
1 var Application = Application || {};
```

Ce code signifie : *Si la variable **Application** existe tu la récupères, sinon tu déclares un nouvel ensemble vide*. Il doit être mis en haut de chaque fichier de module. Il permet de s'assurer qu'on ne va pas tenter de créer un sous-ensemble sans que le parent ne soit initialisé. De plus, il évite d'écraser Application si celui-ci existe déjà et qu'il contient d'autres modules qui ont été chargés avant.

```
1  // fichier div-manager.js
2
3  var Application = Application || {};
4
5  // on peut créer le module enfant
6  Application.DivManager = (function(){
7      /* ... */
8  })();
9
10
11 // et des sous modules...
12
13 Application.DivManager.Events = (function(){
14     /* ... */
15 })();
```

6. Extensions de modules

Le problème des modules c'est qu'ils sont déclarés dans un seul fichier. Dans le cas de très gros projets avec beaucoup de développeurs, il peut être intéressant de pouvoir ajouter des

fonctionnalités sans modifier le fichier principal. Il existe une façon **d'étendre** les modules, et cela avec assez peu de changements.

Je ne parle pas de sous-modules mais bien de *l'ajout de fonctionnalités* dans un module.

```
1  var Application = Application || {};
2
3  Application.DivManager = (function(self){
4
5      self.nouvelleMethode = function(){
6          // fonctions à ajouter au module
7      };
8
9
10     return self;
11 })(Application.DivManager || {});
```

On peut voir sur cet exemple que l'on passe directement en paramètre le module à étendre (ou un ensemble vide si celui-ci n'existe pas). Ce paramètre est directement utilisé par le module à la place du **self** que l'on déclarait au départ.

En procédant ainsi, si le module n'existe pas, il est créé !

Si vous utilisez ce modèle de conception, tous les modules doivent être déclarés de la sorte et self ne doit plus être déclaré dans le corps du module (au risque d'écraser toute extension passée).

Le module qui a été étendu par l'exemple ci-dessus doit, par conséquent, être déclaré de la sorte :

```
1  var Application = Application || {};
2
3  Application.DivManager = (function(self){
4
5      /* toutes les fonctions du module */
6
7      return self;
8  })(Application.DivManager || {});
```

Cette façon de procéder permet un chargement parallèle des fichiers JavaScript. Le module n'a plus besoin d'être créé en amont, il sera créé soit dans le fichier principal soit par la première extension qui tentera de l'étendre.

Enfin, on peut bien évidemment étendre les sous-modules de la même manière en faisant attention à bien vérifier l'existence de leur parent !

J'espère que vous avez pris plaisir à lire ce cours et surtout que vous avez apprécié découvrir ce modèle de conception.

Merci beaucoup à Arius pour sa relecture ainsi qu'aux bêta-lecteurs.