Tutoriel : Bonnes pratiques JavaScript

Table des matières

Bonnes pratiques JavaScript

Un script JavaScript peut rapidement devenir une vraie boucherie impossible à débugger. Pour éviter ces situations, le meilleur moyen est de coder proprement. Voici donc une série de bonnes pratiques utilisées par des professionnels du web, connus et reconnus.

Si les concepts sont encore un peu abstraits (fonctions anonymes, DOM, etc.) ou inconnus (JSON, etc.) pour vous, ce n'est pas dramatique. Le plus important est de prendre connaissance de ces quelques directives et au moins de retenir les choses à ne pas faire, le reste viendra petit à petit.

Deux parties : les erreurs à éviter absolument et les bonnes pratiques proprement dites.

Où placer son code JavaScript ?

Les scripts JavaScript doivent être placés dans un fichier externe à la page HTML. Habituellement on place la balise d'insertion des fichiers dans le `<head>`. Pour une question de performances, préférer la placer à la fin de la page, juste avant `</body>`.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Inclusion de fichier JavaScript dans une page</title>
  </head>
  <body>
    <!--
        tout le contenu de la page
    -->
    <script src="fichier.js" type="text/JavaScript"></script>
</body>
</html>
```

Dans la page HTML peut éventuellement apparaître quelques assignations de variables qui, pour une plus grande simplicité de codage, ne sont pas dans un fichier JavaScript externe.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Configuration rapide de notre application JavaScript</title>
  </head>
  <body>
    <!--
        tout le contenu de la page
    -->
    <script src="fichier.js" type="text/JavaScript"></script>
    <script type="text/JavaScript">
        var configuration = {
            id_mbr: 502,
            posts_par_page: 50
        };
    </script>
  </body>
</html>
```

Le traitement d'une page web par un navigateur est séquentiel, de haut en bas. Placer les scripts au plus bas permet de charger les éléments importants de la page web (le contenu et la présentation) avant de s'occuper des paillettes que représente le script JavaScript.

Lors de tests avec des pages très épurés, il est possible que le DOM ne soit pas encore prêt et que `document.getElementById` ne retourne rien et donc que le script «ne marche pas». Dans ce cas, il faut repasser par l'évènement `window.onload` .

Bannir eval()

L'utilisation de **`eval()`** est à éviter absolument. Cette fonction *démarre le compilateur JavaScript*, ça mange de la mémoire et c'est lent à démarrer. Les performances sont donc mauvaises. De plus, le code contenu dans **`eval()`** ne peut pas être optimisé par le compilateur JS.

Quelques détails sur la sécurité : le script évalué (potentiellement malsain «*Never trust user input*») est exécuté avec les mêmes privilèges que le script où se trouve l'appel à la fonction **`eval()`**. Par exemple pour les extensions Firefox, qui ont accès à des méthodes spéciales du navigateur pour gérer les favoris et d'autres préférences, si on utilise **`eval()`** dans cette extension alors le code contenu dans **`eval()`** peut potentiellement utiliser les mêmes méthodes spéciales !

Pour ceux qui parsent encore leur JSON avec **`eval()`**, il est temps de passer à une vraie librairie : json2.js. Cette librairie utilise l'objet natif s'il est présent et évalue le script d'une façon sécurisée dans le cas contraire. Une présentation sommaire de l'API :

```
// on veut transformer cet objet en chaine JSON
var objet = {
    x: "une chaine",
    y: 5
};

var chaine = JSON.stringify(objet); //renvoie la chaine '{"x":"une chaine","y":5}'


// maintenant on veut un objet à partir d'une chaine JSON
var objet2 = JSON.parse(chaine); // renvoie un objet ayant la même structure que la variable objet ci-dessus

alert(objet2.x); // affiche : "une chaine"
alert(objet2.y); // affiche : 5
```

Attention à setTimeout() et setInterval()

Ces 2 fonctions sont dangereuses si une chaîne de caractères est passée en paramètre, car la fonction **eval()** est lancée implicitement à partir de la chaîne. Exemple : `setTimeout("alert('Boom!');", 2000);`

```
// pour exécuter une fonction sans paramètre
setInterval(maFonction, 1000);

// pour exécuter une fonction avec paramètre
setInterval(function () {
    maFonction(parametre1, parametre2);
}, 5000);

// une méthode plus élégante qui malheureusement ne marche pas sous IE9
setTimeout(maFonction, 5000, parametre1, parametre2 /*, etc. */);
```

La balise <a> et le préfixe JavaScript:

Dans <a href="JavaScript:fonction()", le préfixe **`JavaScript:`** ne doit pas apparaître. Dans l'attribut `href` d'un lien, doit figurer une URI valide qui pointe sur une ressource. Remplacer l'URI par une ancre vide comme `<a href="#" onclick="action();">action</a>`, est aussi mauvais.

Pour ajouter un évènement à un lien existant — par exemple une confirmation sur un lien de suppression, utiliser l'événement **`onclick :`**

```
<a href="url" onclick="return confirm('Supprimer cet objet ?');">supprimer l'objet</a>
```

Pour les actions ne possédant pas d'url (exemple : démarrage d'un compte à rebours, changement de couleur d'un élément, cacher un élément, ...) la balise consacrée est nommée `<button>` et possède un attribut `type` qui peut prendre les valeurs

- **`button`** si pas d'action par défaut, le bouton ne « fait rien »,
- **`submit`** c'est la valeur par défaut. Le bouton soumet le formulaire parent,
- **`reset`** remplace tous les champs par leurs valeurs par défaut.

L'utilisation est simple :

```
<button type="button" onclick="cacher();">Cachez moi !</button>
```

L'attribut `type="button"` est très important. Si on ne précise pas le type, par défaut `<button>` envoie le formulaire parent. (`<button>` peut être n'importe où ou presque dans le HTML d'une page !).

Proscrire document.write()

Le comportement de cette fonction est problématique. Il n'est pas constant et induit les débutants en erreur. Lors du chargement d'une page HTML cette fonction **ajoute** la chaîne passée en paramètre au contenu. Une fois la page chargée, cette fonction **remplace totalement** le HTML de la page par la chaîne en paramètre.

Le chargement d'une page HTML est séquentiel, c'est un exercice périlleux pour le navigateur que de rajouter du contenu à une page en train de charger. Si c'est périlleux, les bugs ne sont pas loin. Pour éviter les comportements « étranges » il ne faut pas utiliser **`document.write()`** !

L'alternative est d'utiliser le DOM. Si on veut ajouter du contenu à une page HTML placer un élément vide à l'endroit voulu et le remplir une fois la page chargée :

```
<!DOCTYPE html>
<html>
  <head>
    <title>Une page web</title>
  </head>
  <body>
    <h1>Titre de la page</h1>
    <p>Du contenu?</p>
    <p id="dynamique"></p>
    <p>Encore plus de contenu</p>
    <script type="text/JavaScript">
       document.getElementById("dynamique").innerHTML = "chaine de caractères" ;
    </script>
  </body>
</html>
```

Le HTML reste propre et le contenu est ajouté. Le JavaScript devrait se trouver dans une page séparée, il est ici dans le HTML pour simplifier.

Utiliser innerHTML ?

Il convient de faire attention à l'utilisation **`innerHTML`**. Ce n'est pas une propriété standard du DOM mais est très utile pour les scripts nécessitant la performance (et les scripts simplistes).

Pour rajouter des centaines d'éléments à une page, remplacer toute une zone de notre page avec du nouveau contenu, ou quand on veut supprimer tous les enfants d'un élément, il est plus rapide — à écrire et à exécuter — de le faire avec **`innerHTML`** qu'avec le DOM. Mais en utilisant (mal) **`innerHTML`**, on peut « casser » le HTML et les performances auxquelles on s'attendait.

Règles à suivre pour avoir le minimum de problème et le maximum d'efficacité :

- Toujours vérifier avec un outil adapté (en utilisant firebug — `console.log();` — ou un simple `alert();`) que le HTML que l'on cherche à rajouter est **bien formé** : toutes les balises ouvertes sont fermées, pas de caractère illégal dans les attributs, etc.
- Appeler **`innerHTML`** **une seule et unique fois** par élément. Si on l'utilise dans une boucle et que l'on rajoute un bout de chaîne à la fois, les performances sont aussi mauvaises (voir pire !) qu'avec le DOM.

Exemple : on cherche à rajouter 100 éléments à notre liste HTML ayant un id `todo` :

```
var   // pour limiter la portée des variables
 liste = document.getElementById("todo"), //pour éviter les document.getElementById
                                          // consécutifs qui ralentissent le code

 taches = "",   // chaine a rajouter à notre liste

 i; // déclarer la variable d'incrémentation ici ou dans la boucle
      for (i = 0; i < 100; i++) {
            taches += '<li id="tache_'+ (i+1) +'">tâche n°:'+ (i+1) +'</li>';
      }
    // une fois la chaîne créée, l'ajouter à la page (opération la + coûteuse du script):
      liste.innerHTML = taches;
```

Une erreur courante est de remplacer `taches` dans la boucle par `liste.innerHTML`. Pour la rapidité du script, c'est fatal. Utiliser le DOM pour toute manipulation plus évoluée des nœuds du document HTML

innerHTML versus DOM manipulation

https://dev.to/grandemayta/JavaScript-dom-manipulation-to-improve-performance-459a
https://andrew.hedges.name/experiments/innerhtml/original.html

with()… out!

`with()` peut paraître une bonne idée sur le moment. Grave Erreur. On parle d'une construction qui permet de « raccourcir » l'écriture lorsque l'on travaille avec des objets ayant beaucoup de « profondeur ». Tout de suite un exemple qui montre ce dont on parle et les imprécisions que cette construction entraîne :

```
function monAction () {
    var element = document.getElementById("idElement"),
        color = "yellow";

    // on ne veut pas avoir à réécrire "element.style." pour chaque propriété
    // on utilise alors with() pour "raccourcir".
    with (element.style) {
        color = "red";
        zIndex = "20";
        fontFamily = "Verdana";
    }
}
```

Maintenant, question : qu'est-ce que `color` modifie ? la variable `color` de la fonction ou `element.style.color` ? Impossible à savoir. Il est tout aussi impossible de savoir à quoi se réfère `this` à l'intérieur du **`with()`**. Pointe-t-il sur `element.style` ou fait-il référence à l'objet parent ? Mystère.

La bonne façon de faire est de créer une nouvelle variable pour lever les ambiguïtés.

```
function monAction () {
    var element = document.getElementById("idElement"),
        style = element.style, // on crée une variable pour lever l'ambiguïté
        color = "yellow";

    style.color = "red";
    style.zIndex = "20";
    style.fontFamily = "Verdana";
}
```

Là, tout est clair. On sait exactement ce que l'on modifie.

Règles de syntaxe

Quelques points noirs du langage :

- les accolades dans les conditions sont optionnelles,
- le compilateur « devine » où ajouter des points virgules en cas d'oublis,
- une variable est globale par défaut.

Toujours déclarer ses variables avec **var** (voir plus bas) et ajouter un **`";"`** à la fin de chaque expression :

```
// on assigne une variable
var variable = "une chaine";

// plus subtil, on assigne une fonction anonyme à une variable
var maFonction = function () {
    // le code de la fonction
    return "valeur à retourner";
};

// on utilise plus souvent ce genre de chose
Date.prototype.maFonctionPerso = function (format) {
    // on formate la date comme on veut
};
```

```
// on crée un objet
var monObjet = {/* les propriétés et méthodes */};


// un appel de fonction
maFonction();


// ATTENTION : pour une déclaration de fonction pas besoin de point virgule !
function uneFonctionInnocente (param) {
    // code de la fonction
}
```

Toujours mettre les accolades aux conditions et aux boucles, même (et surtout) pour une seule instruction. Un oubli est si vite arrivé. C'est un détail qui simplifie grandement la maintenance et réduit le temps passé à débugger le code — on a juste une instruction à rajouter sans avoir à s'occuper de savoir si les accolades sont présentes ou non.

```
if (valeur === "un truc") {
    // exécuter du code
}
else if (valeur === "un autre truc") {
    // exécuter du code
}
else {
    // executer du code
}


for (var i = 0; i < 50; i++) {
    // instructions
}
```

La notation « littérale »

C'est une façon élégante de créer des objets très courants (Array, Objet et RegExp).

Voici donc la forme préférée pour la création de ces objets :

```
// Les tableaux

// constructeur: Array
var tab = [];// un nouveau tableau vide
var tab = ['zéro','un','deux','trois'];// avec du monde dedans


// Les objets

// constructeur: Object
var obj = {}; // un objet vide
var obj = { // avec du monde dedans
    propriete: "valeur",
    methode:function () {}
};


// Les expressions régulières

// constructeur: RegExp
var reg = /\d+/ig; // une regex
```

Toujours utiliser var

Les variables en JavaScript sont globales par défaut. Ce qui peut très facilement entraîner des conflits. Quand on déclare une variable avec le mot-clef **`var`** devant, on rend cette variable locale à la fonction parente — pour nos amis venus d'autres horizons (C/C++, etc.) les boucles ne peuvent pas avoir de variables locales en JavaScript, c'est un privilège des fonctions.

```
// dans ces deux fonctions la variable "i" a exactement la même portée

function uneBoucle () {
    // ici on déclare la variable dans la boucle, c'est pas le mieux
    for (var i = 0; i < 100; i++) {
        // du code
    }
}


function uneAutreBoucle () {

    var i; // déclarer _toutes_ les variables en même temps au début de la fonction

    for (i = 0; i < 100; i++) {
        // du code
    }
}
```

Pour connaître la portée d'une variable, il suffit de trouver la première fonction parente : c'est à l'intérieur de cette fonction que la variable est définie. S'il n'y a pas de fonction parente alors la variable est globale au script.

```
// pas de "var" : donc c'est une variable globale
casper = "variable globale";


// il n'y a pas de fonction parente explicite alors c'est aussi une variable globale
var variableGlobale = true;


function topSecrete () {

    var casper = "fantôme";

    alert(casper);
}


topSecrete();  // affiche : fantôme
alert(casper); // affiche : variable globale
```

On peut déclarer plusieurs variables locales à la suite avec un seul mot-clef **`var`** en séparant les assignations par des virgules — sans oublier de terminer par un point virgule final.

```
function maFonction () {

    var variable1 = "",
        variable2 = [],
        variable3 = 0,
        variable4, variable5;

    function locale() {
        // dans cette fonction toutes les variables de la fonction parente
        // sont utilisables. De plus cette fonction est locale à «maFonction»
    }

    // ici variable4 et variable5 sont des variables locales de type "undefined"
}
```

(petit) Récapitulatif

Ne pas utiliser :

- `eval()`
- le préfixe `JavaScript:`
- `with()`
- `document.write();`

Faire attention avec :

- `setTimeout()` et `setInterval()`
- `innerHTML`

Utiliser :

- le mot-clef **`var`** pour déclarer ses variables
- des accolades et des points-virgules partout où il en faut
- la syntaxe littérale pour créer des objets
- placer son code JavaScript après tout le contenu HTML de la page

```
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="content-type" content="text/html;charset=utf-8">
    <title>Une page toute gentille</title>
  </head>
  <body>
    <a href="/url/service/classique" onclick="vaChercherAjax();">Un vrai lien</a>
    <button type="button" onclick="actionImmediate();">Action !</button>
    <script src="mesScriptsExternes.js" type="text/JavaScript"></script>
    <script type="text/JavaScript">
        var configuration = {
                utilisateur: "nod_",
                id: "fixe",
                age: ""
            };

        // ou si on a défini un objet global ailleurs dans le code
        MonObjetGlobal.configuration = {
            utilisateur: "nod_",
            id: "fixe",
            age: "",
            sexe: "oh oui"
        };
    </script>
  </body>
</html>
```

NB : En suivant cet ensemble de conseils, votre code sera bien plus robuste et facile à maintenir.

Outil en ligne pour 3 fenêtres (HTML CSS JS)

https://codepen.io/pen/?&editable=true