Bonnes pratiques javascript

Par nod_

www.siteduzero.com

Dernière mise à jour le 6/11/2012

Sommaire

Bonnes pratiques javascript

Par nod_

Mise à jour : 06/11/2012
Difficulté : Facile

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.

J'utilise probablement des concepts qui 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. Plus tard ça fera tilt et les réflexes seront déjà là .

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

Pour la notation :

- **rouge** : tout ce qu'il ne faut jamais faire ni utiliser,
- **orange** : faire attention lors de l'utilisation de cette fonction ou propriété,
- **vert** : ce dont il faut user et abuser.

Sommaire du tutoriel :

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 on préférera la placer à la fin de la page, juste avant **</body>** À noter que les deux méthodes sont valide du point de vue de la spécification HTML et XHTML.

L'attribut **language** est déprécié et ne doit pas figurer.

Code : HTML

```
<!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 apparaitre quelques assignations de variables qui, pour une plus grande simplicité de codage, ne sont pas dans un fichier javascript externe.

Code : HTML

```
<!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. De plus en plaçant le javascript tout en bas, il est possible d'utiliser le DOM directement et **avant** l'habituel événement `onload` de la page HTML. Quelques centaines de millisecondes qui font une différence.

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` . Cependant, sur un site ayant du contenu, il est rare de rencontrer ce problème.

Commentaires HTML et code javascript

Ici on va parler de cette pratique (et toutes les autres variantes sur le même principe) qui consiste à « cacher le code javascript aux "vieux" navigateurs » :

Code : HTML

```
<script type="text/javascript">
<!--


// -->
</script>
```

Il y a 10 ans c'était une technique valable, plus maintenant. Un navigateur qui affiche le code javascript au lieu de l'exécuter ou de l'ignorer est buggé. De nos jours il faut placer les scripts dans un seul ou plusieurs fichiers externes inclus à la page avec la balise **<script>** comme on l'a vu au-dessus, ce qui contourne le problème des commentaires et permet de profiter du cache du navigateur.

De plus, la très grande incompréhension du XHTML1.0, du XHTML1.1, des doctypes et du type mime consacré aux pages XHTML rend toute tentative de «camouflage» bancale puisqu'une mauvaise utilisation des commentaires peut facilement rendre le code et/ou la page invalide (au sens du validateur W3C).

Pour être clair, s'il faut mettre du code javascript directement dans le (x)HTML :

- **HTML 4+** : pas de commentaires,
- **XHTML 1.0 strict** : envoyé avec le type mime **application/xhtml+xml** il faut placer le code javascript dans une balise `<![CDATA[ ]]>` pour que le code javascript contenant des "<" soit ignoré par le parseur.
- **XHTML 1.0** : avec le type mime `text/html` c'est ridicule. Il vaut mieux utiliser **HTML 4**.

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. Pour la petite histoires l'objet natif `JSON` — utilisé pour le parser — de IE8 est basé sur ce script précisément. Le plus merveilleux c'est que l'API est consistante sur IE8, FF et WebKit ! (Opera traîne). 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 :

Code : JavaScript

```
// on veut transformer cet objet en chaine JSON
var objet = {
    variable1: "une chaine",
    variable2: 5
};

var chaine = JSON.stringify(objet); // renvoie la chaine
'{"variable1":"une chaine","variable2":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.variable1); // affiche : "une chaine"
alert(objet2.variable2); // affiche : 5
```

Attention à setTimeout() et setInterval()

Ce sont deux fonctions très dangereuses : en passant une chaine de caractères en paramètre, ces fonctions appellent **eval()**. Mal. Très mal. Surtout qu'il est possible de passer des paramètres à ces fonctions naturellement.

Code : JavaScript

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

// si il faut passer des paramètres à notre fonction du futur, il
faut l'englober
// dans une fonction anonyme qui contient l'appel à notre fonction
paramétré.
setInterval(function () {
    maFonction(parametre1, parametre2);
}, 5000);

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

Il ne faut **jamais** passer de chaine de caractère en paramètre des fonctions `setTimeout()` et `setInterval()`.

La balise <a> et le préfixe javascript:

Ce préfixe `javascript:` est encore une relique honteuse d'une autre époque. Il ne doit jamais apparaitre. Il est inutile de l'utiliser sachant que le code javascript n'a rien à faire dans un attribut `href` d'une balise `<a>` . Dans l'attribut `href` d'un lien doit figurer une URI valide qui pointe effectivement sur une ressource, dans ce cas `javascript:` est traité comme un protocole ! Protocole qui n'existe pas ; le lien est alors invalide **et** inutile. À l'heure du web accessible, ça fait tâche. Remplacer l'URI par une ancre vide, le fameux `<a href="#" onclick="action();">action</a>` , est aussi mauvais.

La solution est simple. Il suffit d'utiliser les standards… Pour rajouter un évènement à un lien existant — par exemple une confirmation sur un lien de suppression ou la prise en charge par « AJAX » du chargement d'un lien — alors, il faut utiliser l'événement `onclick`. Pour confirmer une suppression, on peut par exemple utiliser :

Code : HTML

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

Maintenant pour les actions ne possédant pas d'url — le démarrage d'un compte à rebours, le changement de couleur d'un élément, cacher un élément (les <secret> et autre) — la balise consacrée est nommée `<button>` et possède un attribut `type` qui peut prendre les valeurs

- **`button`** il n'y a 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 :

Code : HTML

```
<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. (Cependant `<button>` n'a pas besoin de se trouver dans un formulaire, il 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 chaine passée en paramètre au contenu. Une fois la page chargée, cette fonction **remplace totalement** le HTML de la page par la chaine en paramètre.

On a vu plus haut que 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 — encore une fois — d'utiliser les standards et plus précisément le DOM. Si on veut rajouter du contenu à une page HTML le plus simple est de placer un élément vide à l'endroit voulu et de le remplir une fois la page chargée. Si l'on applique les recommandations sur la place de `<script>` , c'est simple et direct :

Code : HTML

```
<!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 = "ma chaine
de caractères à rajouter avec javascript";
    </script>
</body>
</html>
```

Le html reste propre et le contenu est rajouté comme il faut, le javascript devrait se trouver dans une page séparée, il est ici dans le HTML pour simplifier.

Utiliser innerHTML ?

On vient juste d'utiliser `innerHTML` or il convient de faire attention à son utilisation. Ce n'est pas une propriété standard du DOM mais elle est très utile pour les scripts nécessitant la performance (et les scripts simplistes). En bonus elle est disponible dans IE et tout navigateur digne de ce nom.

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 il faut rester prudent sur son utilisation. En utilisant (mal) `innerHTML` on peut « casser » le (x)HTML et les performances auxquelles on s'attendait.

Voilà les quelques 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'en utilisant le DOM.

Pour l'exemple on cherche à rajouter 100 éléments à notre liste HTML ayant un id `todo`.

Code : JavaScript

```
// on oublie pas le mot-clef pour limiter la portée des variables
var
```

```
    // une bonne habitude à prendre, garder une référence à l'objet
dans une variable
    // pour éviter les document.getElementById consécutifs qui
ralentissent le code
    liste = document.getElementById("todo"),

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

    // on déclare la variable d'incrémentation au début de
préférence.
    // décalarée ici ou dans la boucle, elle a la même portée.
    i;


for (i = 0; i < 100; i++) {

    // il est préférable d'utiliser des quotes simples quand on
travaille avec du HTML
    // pas besoin d'échapper les innombrables " dans les attributs
HTML
    taches += '<li id="tache_'+ (i+1) +'">tâche n°:'+ (i+1)
+'</li>';
}


// une fois la chaine crée on la rajoute à la page. C'est
l'opération la plus coûteuse du script.
liste.innerHTML = taches;
```

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

with()… out!

Après quelques recherches sur le javascript il se peut que l'on finisse par tomber sur **`with()`** qui 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 :

Code : JavaScript

```
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.

Code : JavaScript

```
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. Les radins des accolades iront voir plus bas les détails sur la notation « littérale » (qui est une bonne pratique, elle) pour « gagner » des caractères et « raccourcir » leurs scripts.

Règles de syntaxe

Maintenant des détails importants sur la syntaxe du javascript.

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.

Ça n'a pas l'air méchant de loin, mais ça l'est. C'est pourquoi il faut toujours déclarer ses variables avec **var** (voir plus bas) et ajouter un point virgule à la fin de chaque expression :

Code : JavaScript

```
// 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.

Code : JavaScript

```
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
}
```

Il est crucial pour une application javascript qui commence à grossir d'avoir une syntaxe consistante. Les feintes d'écritures pour gagner 3 pauvres caractères et un saut de ligne ne valent pas la peine. D'ailleurs, il existe des outils dédiés au traitement du code final pour en réduire au maximum la taille. Pour pouvoir les utiliser, il faut suivre scrupuleusement ces quelques règles (en particulier sur les points-virgules).

La notation « littérale »

C'est une façon élégante de créer des objets très courants (Array, Objet et RegExp). Avis aux radins des accolades du paragraphe précédent, si il y a des caractères à gagner dans un script c'est ici que ça se passe. Un effet bénéfique de cette notation est qu'elle limite l'utilisation du mot-clef **new**. Un petit oubli de ce mot-clef et les 7 plaies d'Égypte vont s'abattre sur votre script, causant des problèmes plus mystiques les uns que les autres (qui viennent juste du fait que l'objet original a été sauvagement modifié, au lieu d'une «copie» de celui-ci).

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

Code : JavaScript

```
// 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
```

C'est pas plus sexy comme ça ?

Toujours utiliser var

On a vu plus haut que les variables en javascript sont globales par défaut. Ce qui peut très facilement entrainer 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.

Code : JavaScript

```
// 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; // il est préférable de 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.

Code : JavaScript

```
// 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 bien entendu :).

Code : JavaScript

```
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"
}
```

On n'a pas pollué l'espace global, tout le monde est content et personne ne se marche sur les pieds.

(petit) Récapitulatif

Ne pas utiliser :

- `eval()`
- le préfixe `javascript:`
- `with()`
- `document.write();`
- Des commentaires pour « cacher » le code javascript

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

Code : HTML

```
<!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>
```

En suivant cet ensemble de conseils, votre code sera bien plus robuste et facile à maintenir. Il faut bien être conscient que tout cela ne diminue en rien les possibilités du javascript. Bien au contraire.

Armé de ces conseils il est alors possible d'utiliser au mieux toute la puissance du langage et honnêtement c'est à ce moment-là que javascript devient franchement fun.

Avant de partir, quelques liens :

- quelques pages d'un très bon livre (c'est rare pour javascript !) en français : Javascript : La référence (+ un rhino),
- des conférences géniales (en anglais) sur le bon javascript c'est sur YUI theatre. C'est comme le cochon : tout est bon,
- Une référence du javascript (c'est son job: «ingénieur javascript») : Douglas crowford,
- quirksmode.org et surtout les tables de compatibilité ! Toujours avoir sous la main.

Aimez le javascript, il vous aimera (et moi aussi) en retour.