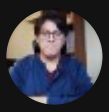

@davidenq Follow

Jun 21, 2016 · 18 min read

Guía de estilo, convenciones y buenas prácticas de desarrollo con Javascript.

Esta es una guía (no oficial, ni estandarizada) sobre reglas, convenciones y buenas prácticas usados en el lenguaje de programación Javascript. Es inspirado en varios documentos de autores reconocidos en el ámbito Javascript, y que han sido adoptados por muchas personas, inclusive por grandes entidades tales como PayPal, Stronloop, entre otras. Además, se tiene una fuerte referencia de los libros “Código Limpio” y “Javascript: The Good Parts”.

í ndice de contenidos

Introducción

El hecho de que conozcas muchos de los conceptos de algunos paradigmas de programación como la orientación a objetos, por ejemplo; y aunque escribas miles de líneas de código que cumplen con un cometido, no te hacen un buen programador si es que no aplicaste buenas prácticas de desarrollo, simplemente porque tu código no será un código limpio.

Pero, ¿Qué es código limpio?

Código limpio

Definiciones tomados del libro código limpio.

Bjarne Stroustrup, creador de C++.

> *código limpio hace bien una cosa.*

Dave Thomas, fundador de OTI.

> *El código limpio se puede leer y mejorar por parte de un programador que no sea su autor original. Tiene pruebas de unidad y de aceptación. Tiene nombres con sentido. Ofrece una y no varias formas de hacer algo.*

Michael Feathers, autor de Working Effectively with Legacy Code.

> *…El código limpio siempre parece que ha sido escrito por alguien a quien le importa. No hay nada evidente que hacer para mejorarlo*.

Son definiciones bastante claras acerca de lo que es un código limpio, y aunque podría dejarlo ahí y continuar con la guía, me gustaría dar mi punto de vista y agregar un poco más de información.

Considero que para escribir un buen código; que cumple con algunas características muy importantes; y que algunas de ellas ya han sido mencionadas en las definiciones anteriores, debes conocer no solo convenciones y estilos, sino también principios y patrones de diseño (de comportamiento, creacionales, estructurales, etc) que ya han sido establecido por los gurús de la programación, esto último será otro tema de conversación.

Para crear un código limpio se requiere de mucho esfuerzo, conocimiento, trabajo constante, y no siempre se consigue a la primera. Es por eso que cada línea de código escrita, se debe analizar, sintetizar y volver a reescribir si es necesario para que cumpla con las características mencionadas anteriormente.

> *Y es que Todo el código en cualquier proyecto debería verse como si una sola persona lo hubiera escrito, sin importar cuánta gente haya contribuido. [ref].*

¿Por qué se debe seguir una guía, convenciones, patrones, etc?

Porque para cuando hayas desarrollado una aplicación, o inclusive aún esté en fase de desarrollo, y has tenido una participación importante en el proyecto; aportando con cientos de miles de líneas de código y siguiendo las buenas prácticas de desarrollo

sabiendo que el código que escribiste, sigue las buenas prácticas de desarrollo.

En resumen:

> *"La productividad del equipo aumenta considerablemente y disminuye el sufrimiento dado que seguiste convenciones y buenas prácticas de desarrollo".* Y te vas con la conciencia tranquila.

No olvides que el desarrollo de software de cualquier tipo, es una disciplina que te demandará mucho, mucho tiempo, constancia, paciencia y perseverancia; dado que hay constantes cambios y nuevos problemas a los que hay que hacer frente.

Si estás dispuesto a ser un mejor programador y/o desarrollador, sigue una guía, por ejemplo la guía que se encuentra descrita a continuación, sin embargo si no estás conforme con esta guía, adjuntaré enlaces adicionales a otras guías no solo para Javascript sino también para otros lenguajes de programación.

Finalmente si decides tener esta guía como referencia, y si crees que alguna convención, estilo no termina de convencerte, pues es un reto que podemos asimilarlo y discutirlo y de ser necesario cambiarlo.

Guía de estilo, convenciones y buenas prácticas.

Reglas

Por el momento, aquí se menciona una única regla que se tendrá en cuenta.

La regla del Boy Scout

Hay una regla simple y directa para los Boy Scouts definida por Lord Robert Stephenson Smith Baden-Powell

> *Intenta dejar este mundo un poco mejor de como lo encontraste.*

Sin embargo, Uncle Bob, autor del libro código limpio, lo toma de otra manera.

> *Siempre deja el lugar de acampamento más limpio que como lo encontraste*



Si abres un archivo para revisarlo y ves que no está cumpliendo algunos requisitos, no importa quién fue el autor original, toma un poco de tu tiempo y la molestia de mejorarlo; agregando o borrando comentarios, renombrando variables o funciones, formateando el código, etc.

> *Deja el código mejor de cómo lo encontraste.*

No estoy diciendo que se haga una refactorización completa del código sino, solo simples mejoras que pueden contribuir a que el código sea aún mejor para mantenerlo y que no se degrade sino más bien que mejore en muchos aspectos (lectura, testeable, etc).

Sigue la regla del Boy Scout, no seas individualista cuidando todos los detalles solo de tu desarrollo, ayuda a aquellos que están escribiendo mal y en concreto, a que mejoren.

Convenciones

Para lograr un código limpio, a continuación se describe convenciones que se utilizarán en Javascript, independientemente si es en Frontend o Backend.

Es importante mencionar que dichas convenciones son tomadas de varías referencias (al final se encuentran los enlaces web), he inclusive algunas convenciones han sido tomadas de Frameworks Javascript, que desde mi punto de vista y muy particular por cierto, han sido bien organizados. Por ejemplo, HapiJs (Si en verdad soy un fanboy de hapi). No puedo decir lo mismo acerca de otros frameworks ya bien conocidos y establecidos.

Espacios en blanco — Identación

Normalmente se puede trabajar con identación y espacios en blanco en el código y no es importante para el intérprete puesto que no los toma en cuenta, pero para el programador, esto provee una mejor legibilidad del código.

Entonces la recomendación es que:

- *Nunca se debe mezclar espacios y tabulaciones*.

Por lo general tu editor de texto o IDE, te permite configurar este parámetro y por defecto suele estar configurado con una identación de 4 espacios, pero en el caso de que no sea así, revisa la configuración de tu IDE o editor de texto y configura según la convención.

Para archivos .js que serán servidos públicamente (Frontend), recuerda que el interprete de Javascript no toma en cuenta espacios en blanco o identación, y no te preocupes por el tamaño del archivo inicial, si eres prudente deberás realizar procesos de minificación para crear archivos optimizados en su tamaño.

Longitud de una línea

El límite de las líneas de código deberán ser de 80 caracteres, esto ayudará a una mejor lectura del código. Es seguro que tu editor de texto o IDE soporta esta característica.

Comillas

Hay algunas recomendaciones acerca de esta notación. Aunque Javascript permite usar comillas simples o comillas dobles indistintamente, generalmente es recomendable, en ciertos casos, escribir con comillas simples y en otros con comillas dobles.

En ciertos casos se desea tener comillas dentro de un texto, y tendrán otras como delimitadores externos del texto.

Es valido

```
const foo ='"Bienvenidos"';
console.log(foo); // "Bienvenidos"

//
const foo1 ="'Bienvenidos'";
console.log(foo1); // 'Bienvenidos'

//Otro ejemplo
const foo2 = 'Bienvenidos,';
const foo3 = 'Hola';

console.log(foo2 + foo3 + ' Mundo.'); //Bievenidos, Hola Mundo.
```



simples.

Recomendado

```
const foo = 'bar';
console.log('bar');
function(foo, bar){
    const x = 'hi' + foo + ';' + 'bar';
}
```

Llaves

La llave de apertura deberá ir en la misma línea de la sentencia. Si colocas en la siguiente línea, en algunos casos muy particulares se puede producir algún error. Más adelante se explica dicho error en el uso de punto y coma en las sentencias.

No recomendado

```
// control flow stament
if ( true )
{
   //codes goes here
}

//Anonymous function declaration
function ( args )
{
    return true;
}

//Named function declaration
function foo()
{
   return true;
}

//Anonymous function expression
const bar = function ( args )
{
   return true;
}
```



```
const bar = ( args ) =>
{ true };
```

Recomendado

```
// control flow stament
if ( true ) {
 //codes goes here
}

//Anonymous function declaration
function ( args ) {
   return true;
}

//Named function declaration
function foo() {
   return true;
}

//Anonymous function expression
const bar = function ( args ) {
   return true;
}

//Arrow function
const bar = (arg) => {
   return true,
};

const bar = ( args ) => { true };

const bar = ( args ) => true;
```

Punto y coma

Javascript utiliza ASI (Automatic Semicolon Insertion) cuando se trata de insertar un "punto y coma" que ha sido omitido en cada instrucción , y que es separada en una línea diferente. Técnicamente es posible, sin embargo esta no es una buena práctica, porque en ciertas ocasiones se puede generar un error que será un quebradero de cabeza poder solucionarlo.

  

```
console.log(getName());

function getName(){
   return
   {
     name: '@davidenq'
   }
}
```

Ejecutando el código se obtendrá el resultado será:

```
undefined
```

Sin embargo, se esperaba como resultado:

```
{ name:'@davidenq' }
```

Analizando el código

En el instante de ejecutar el código, el intérprete de Javascript detecta donde es necesario colocar un ; y simplemente lo inserta. En este caso el intérprete consideró que después dereturn termina la instrucción porque a continuación hay un salto de línea por lo que inserta un ;. Luego lee las siguientes líneas y coloca otro ;al final de la llave. Al ejecutar el código ya sabemos cuál es resultado: undefined

```
//Esto es lo que ASI interpreta
function getName(){
    return; //mistake
    {
        name: "Bootcampio"
    };
}
```



coma cuando sea necesario.

¿Dónde está permitido colocar punto y coma en las instrucciones?

Cuando se trata de algunos flujos de control

```
//control flow stament
do {
   //codes goes here
} while ( condition ); //necesary semicolon
```

Cuando se trata de expresiones de funciones o funciones flecha

```
//Anonymous function expression
const bar = function ( args ) {
    //codes goes here
};

//Named function expression
const bar = function foo( args ) {
    //codes goes here
};

//Autoload function
(function( args ){
    //codes goes here
})( args );

//Arrow function
const bar = () => {
    return foo;
};

const bar = () => { foo };

const bar = () => foo;
```

Luego de notación de objetos

Luego de un arreglo

```
const fruits = ['Apple', 'Orange', 'Pear'];
```

Cuando retornas una expresión o resultado

```
…
return { … };
…

…
return ( … );
…

…
return expression;
…
```

¿Dónde se puede omitir el punto y coma en las instrucciones?

Cuando se trata de algunos flujos de control

```
if( condition ) {
    //codes goes here
} else if (otherCondition ){
    //other codes goes here
} else {
    //other codes goes here
}

for( operation ) {
    //codes goes here
}

while( condition ) {
    //codes goes here
}
```



```
    //codes goes here
}

//Named function declaration
function foo( args ){
    //codes goes here
}
```

Ámbito de las variables (Scope) y Hoisting

Cuando se habla del ámbito de las variables (scope), hay que tener en cuenta si es un ámbito global o local. Normalmente hablamos de un ámbito local cuando dichas variables están contenidas dentro de una función. Entonces dicha función sería el contenedor de la variable, siendo este su scope. Esta variable no tiene ningún valor fuera de su contenedor.

Veamos con ejemplos más claros acerca de estos conceptos.

```
var day = 'Saturday';

function showDay() {

    console.log(day); //
    var day = 'Monday';
    console.log(day); //

}

showDay();
console.log(day);
```

Al ejecutar el código, automáticamente el intérprete de Javascript moverá la variable x a la parte superior del ámbito contenedor (Esto es lo que se conoce cómo Hoisting).

```
var day = 'Saturday';//Declaración uno

function showDay(){
    var day;//Fue movida por el interprete Javascript. Esto no se ve
            //reflejado en el editor de texto
```



```
console.log(day)
```

El resultado obtenido en consola será:

```
>_ undefined
>_ Monday
>_ Saturday
```

Aunque se haya declarado la variable day antes de la función y seteado con un valor (ver Declaración uno). El primer console.log muestra el resultado undefined y no Saturday como se esperaba. Esto se debe a que la declaración uno es realizada en el ámbito global, y la declaración dos en el ámbito de su contenedor, que es la función misma. Por lo tanto, la declaración dos es movida por Javascript al inicio de su ámbito contenedor, haciendo que está variable se setee a undefined, posteriormente es seteada a Monday y finalmente termina su ámbito contenedor y day, vuelve a tener el valor Saturday seteado inicialmente.

Obviamente, este tipo de declaraciones no suele llevarse a cabo cuando se está escribiendo código. Es decir, declarar dos veces la misma variable en dos ámbitos diferentes; normalmente no suele ocurrir, pero podría darse el caso. No obstante, a modo de ejemplo, vemos que Javascript lo permite sin que salte algún error por declarar repetidas veces una misma variable. Esto cambia en ECMASCRIPT6 con las palabras reservadas let y const.

Ahora veamos un caso más práctico. Supongamos que vamos a recorrer un array:

```
for(var i = 0; i < 3; i++){
    var msg = 'Hi';
    console.log(msg + i);
}

function foo(){
    var bar = 'bar';
}
```



El resultado obtenido en consola será:

```
>_ Hi0
>_ Hi1
>_ Hi2
>_ 3
>_ Hi
>_ bar is not defined
```

Se supone que console.log(i) y console.log(msg) no deberían mostrar resultado alguno, es más debería haber arrogado algún error console.log(i)yconsole.log(msg). Por otro lado, console.log(bar) si arroja un error. Esto en otros lenguajes de programación fuertemente tipados, no estaría permitido; arrojando siempre una excepción.

En el caso de for, debería haber arrojado como resultado undefined puesto que suponemos el ámbito de la variable declarada i es la sentencia for(), y no transciende más allá de su contenedor. Sin embargo, el resultado fue 3. Lo mismo sucede con la variablemsg.

Esto se debe a lo mencionado anteriormente; Javascript mueve todas las variables a la parte superior del ámbito.
Pero aquí está la diferencia; las funciones for, if, while, switch; ellas mismas no delimitan un ámbito, por lo tanto, las variables que se declaren dentro, formarán parte de un ámbito mayor; global o a nivel de función.

Entonces la solución a esto sería la siguiente:

```
for(let i = 0; i < 3; i++){
    const msg = 'Hi';
    console.log(msg + i);
}

function foo(){
    const bar = 'bar';
}
```



Declaración de variables

Teniendo en cuenta todo lo mencionado anteriormente en el ámbito de las variables y hoisting, ahora es recomendable usar const y let en lugar de var.

Podrías optar por declarar las variables en la parte superior desu ámbito aunque Javascript no requiera esto. Tratando de que su ámbito sea local y no global. Lo que se obtiene con esto es:

- Código más limpio
- Proporciona un único punto de búsqueda de las variables que son utilizadas en un ámbito.
- Ayuda a conocer que variables están involucradas en el ámbito contenedor.

Formato de declaración

Esto nuevamente, es un punto discutible y cuestión de gustos. Pero sirve como una buena referencia dado que muchos optan por la opción recomendada, sin embargo, otros prefieren las opciones no recomendadas. En fin, si estás siguiendo esta guía, mi sugerencia y no una obligación, es que adoptes la opción recomendada. Sino es así, no hay problema alguno, en este caso es solo cuestión de gustos.

No recomendado

```
// separar por comas
const day = 'Monday',
      count = 10;

//Alinear
const day   = 'Monday',
      count = 10;

//En la misma línea
const day = 'Monday', count = 10;
```

Recomendado

```
const keys = ['foo', 'bar'];
const values = [1, 2];
```

Declaración de variables fuera de las sentencias

No recomendado

```
/*
No hagas que el ciclo for lea repetidas veces la longitud del array.
Solo y solo si estás utilizando colas y/o pilas en el mismo array
sería recomendable dado que el array va disminuyendo y/o aumentando
su tamaño. Caso contrario no lo hagas.
*/

for ( var i = 0; i < fruits.length; i++ ) {
    var type = //something do;
    console.log(type);
}
```

Recomendado

```
const sizeArray = fruits.length;
let type;

for ( i = 0; i < sizeArray ; i++ ) {
    type = //something do;
}
```

Declaración de variables dentro de un contenedor(Reduciendo variables globales)

No recomendado

```
const basket = ['orange', 'apple', 'apple', ….];
const fruit = 'apple';
const numberApples = 0;

console.log(countApples(basket));

function countApples ( fruits ) {
```



```
        }
    }
    return numberApples;
}
```

Recomendado

```
const basket = [‘orange’, ‘apple’, ‘apple’, ….];

console.log(countApples(basket));

function countApples ( fruits ) {

    const fruit = ‘apple’;
    let numberApples = 0;
    const sizeArray = fruits.length;

    for ( const i = 0; i < sizeArray ; i++ ) {
         if( fruit === fruits[i] ){
             numberApples++;
         }
    }
    return numberApples;
}
```

Comparación de variables

Si de comparar true o false se trata, procura seguir la siguiente convención.

No recomendado

```
if ( x === true ) {
    //codes goes here
}

//or

if ( x === false ) {
    //codes goes here
}
```



```
if ( x ) {
    //codes goes here
}

//or

if ( !x ) {
    //codes goes here
}
```

Otras comparaciones

```
const x = 10;
```

Operador	Descripción	Comparación	Resultado
==	igual valor	x == 10	true
==	igual valor	x == 15	false
==	igual valor	x == "10"	true
===	igual valor e igual tipo	x === 10	true
===	igual valor e igual tipo	x === 15	false
===	igual valor e igual tipo	x === "10"	false

Procura utilizar una comparación estricta de valor y tipo con el operador según corresponda=== or !==

Evaluación condicional

No recomendado

```
// Evaluando si el array tiene elementos
if ( array.length > 0) {
```



```
}

// Evaluando que un string no es vacío
if( string !== "" ) {
    //codes goes here
}
```

Recomendado

Evaluar por el valor true o false de la expresión.

```
// Evaluando si el array tiene elementos
if ( array.length ) {
    //codes goes here
}

// Evaluando si el array no tiene elementos (está vacío)
if ( !array.length ) {
    //codes goes here
}

// Evaluando que un string no es vacío
if ( string ) {
    //codes goes here
}

// Evaluando que un string es vacío
if ( !string ) {
    //codes goes here
}
```

Notaciones cortas

No declares las variables primitivas como objetos, puesto que ralentizan la ejecución del código y produce efectos no deseados.

Ejemplos tomados de www.w3schools.com

Por ejemplo:

```
const x = "John";
const y = new String("John");
```



No recomendado

```
const name = new String("John");
```

Recomendado

```
const name = "John";
```

No recomendado

```
const lunch = new Array();
lunch[0]='Dosa';
lunch[1]='Roti';
lunch[2]='Rice';
lunch[3]='what the heck is this?';
```

Recomendado

```
const lunch = [
    'Dosa',
    'Roti',
    'Rice',
    'what the heck is this?'
];
```

No recomendado

```
const o = new Object();
o.name = 'Jeffrey';
o.lastName = 'Way';
o.someFunction = function() {
```



Recomendado

```
const o = {
    name: 'Jeffrey',
    lastName = 'Way',
    someFunction : function() {
       console.log(this.name);
    }
};
```

Formato de nombres de variables y funciones

- Utiliza camelCase para nombrar funciones, declaración de variables, instancias, etc.

Por ejemplo:

Extraido de la guía de estilo de Google Closure Library.

```
- functionNamesLikeThis;
- variableNamesLikeThis;
- methodNamesLikeThis;
```

- Utiliza PascalCase para nombrar constructores, prototypes, clases, etc

Por ejemplo:

Extraido de la guía de estilo de Google Closure Library.

```
- ConstructorNamesLikeThis;
- EnumNamesLikeThis;
```

En variables

- Para declarar valores constantes utiliza Mayusculas y separado por un guión bajo cada palabra.



Recomendado

```
const SYMBOLIC_CONSTANTS_LIKE_THIS;
```

- Para declarar valores variables utiliza camelCas

No recomendado

```
const admin_user;
const days_since_creation;
```

Recomendado

```
const adminUser;
const daysSinceCreation;
```

En funciones

Sean estas anónimas o con nombre. ¿Has dicho anónimas? no que son funciones sin nombre?. Si es verdad, sin embargo, no hay problema en que pueda asignarle un nombre a una función anónima, el interprete seguirá considerando a tal función como una función anónima.

Por ejemplo:

```
//Anonymous function expression
const foo = function () {
    return "anonymous function";
}
console.log(foo()); // 'result anonymous function'

//Named function expression
```



```
//Arrow function

const foo = () => {
    return "arrow function";
};

console.log(bar()); //'result bar is not defined'
console.log(foo()); //'result named function'
console.log(foo); //'result [Function:bar]'
```

Nombres y comentarios en ingles

En general, los lenguajes de programación basan su vocabulario en el idioma inglés, puesto que es el lenguaje más utilizado en el mundo. Esta también es una recomendación discutible. Generalmente la recomendación es que se escriba tanto comentarios, como nombres de variables, de funciones y demás, en ingles. Sin embargo si te sientes mejor escribiendo en el idioma que más lo utilizas, hazlo, pero ten en cuenta que si compartes un módulo o librería probablemente no tenga un alcance global.

Nombres con sentido

La intención de asignar un nombre con sentido a carpetas, archivos, variables, funciones, etc. Es indicar cuál es su cometido, de tal manera que quede claro el propósito por el cuál fue creado.
Y es que esta tarea se realiza constantemente, y toma su tiempo porque muchas de las veces cuesta mucho saber que nombre se le puede asignar para indicar de forma directa su intención y cometido. Pero también tiene sus beneficios asignar nombres coherentes, puesto que más adelante será menor el esfuerzo cuando se vuelva a leer el código pues facilita su compresión.

Por ejemplo:

```
const d; //Elapsed time in days
//or
const day;//Elapsed time
```

Una mejor opción sería: (*extraido del libro Código limpio*)

```
const daysSinceCreation;
int daysSinceModification;
```

Recuerda,

> *¡no declares variables pensando que se da por sentado que es claro su cometido e importancia!.*

Comentarios

Procura asignar comentarios solo cuando sea necesario y que ayuden a entender mejor el código.

Ejemplos extraído del libro código limpio

no recomendado

```
//day of the month
const dayOfMonth;

/*
 * Return
 * @return: day of the month
 *
 * /
 function getDayOfMonth(){
 …
 }
```

Es claro que muchas de las declaraciones son obvias. Sin embargo, es lo que se suele encontrar en el código
No olvides que declarar nombres con sentido ayudan a la legibilidad del código y por ende a evitar comentarios innecesarios.



```
function getDayOfMonth(){
    ...
}
```

En el ejemplo anterior son obvias las intenciones solo con leer el nombre y no falta asignar algún comentario para dar a entender el cometido. No hay que redundar.

Recuerda:

> Procura asignar un nombre con sentido que refleje el cometido, así se evitará realizar comentarios que a veces son redundantes e innecesarios.

Asignaciones y menciones

No agregues menciones en el código que has modificado (eliminado o agregado). Deja que el sistema de control de versiones mantenga esta información fuera del código. (claro si es que estás acostumbrado a usar un sistema de control de versiones).

Si tu desarrollas una librería para el sistema y que pudiera ser liberado y utilizado por otros sistemas, agrega tu información. No hay un esquema básico, sin embargo a continuación se muestra un ejemplo.

```
/**
* Descripción general de lo que hace la librería
*
* @author:
* @email:
* @param: {Tipo} Descripción
* @return:{Tipo} Descripción
* @module: nombre del módulo o libreria
* @licencia:
*
*/
```

Recomendaciones.

- Si está utilizando algún IDE cómo WebStorm, Visual Studio, NetBeans, Eclipse



sobre tu código. En mi experiencia con HapiJs

- Si está utilizando un editor de texto como SublimeText, Bracket o Atom, revise si es que existe un plugin disponible para su instalación, caso contrario podría utilizar JSLint online.

Conclusiones finales

Recuerda seguir las buenas prácticas de desarrollo para evitar horas innecesarias en reescribir un código mal escrito, llevando con ello la reducción de la productividad de trabajo.

Varios ejemplos y definiciones son tomados de algunas referencias bibliográficas. En cada uno de ellos se hace referencia a dicho enlace. Otros posiblemente no contengan el enlace, porque se me paso por alto o porque simplemente son ejemplos propios, sin embargo, está documentación estará en constante revisión y se agregarán las referencias en caso de ser necesario.

Esta publicación ha sido bastante extensa y creo que debería haber sido fragmentado en varias publicaciones. En fin, aún falta más informacion referente a principios de diseño como DRY o S.O.L.I.D. Espero algún día publicarlo, está en borrador, sin embargo toma tiempo la revisión y corrección de la información. Esta mismo ha estado en borrador desde mediados del 2015 y es a la fecha que me he animado a publicarlo.

Referencias bibliográficas

Enlaces web

- https://github.com/rwaldron/idiomatic.js/tree/master/translations/es_ES
- https://mhdev.readthedocs.org/es/latest/js-style.html
- http://www.w3schools.com/js/js_scope.asp
- http://code.tutsplus.com/tutorials/24-javascript-best-practices-for-beginners–net-5399
- https://www.thinkful.com/learn/javascript-best-practices-1



- Robert C. Martin, Código Limpio — Manual de estilo para el desarrollador ágil de software. Prentice Hall, Copyright @2009