

**Developer’s Handbook**

**HTML5 Best Practices – Vol. I, 2013**

**ANUP BISHNOI**

**VIKAS BURMAN**



**COPYRIGHT NOTICE**

Copyright © 2013-2014 Nagarro Software Pvt. Ltd.

**All rights reserved.**

This document is confidential and proprietary to Nagarro Software Pvt. Ltd. and no part of this document should be reproduced, published, transmitted or distributed in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, or stored in any information storage or retrieval system of any nature nor should this be disclosed to third parties without the prior express written authorization of Nagarro Software Pvt. Ltd.

Contents

[1](#_Toc378698855)

[HTML5 Best Practices – Vol. I, 2013 1](#_Toc378698856)

[Introduction 4](#_Toc378698857)

[Importance of Best Practices 5](#_Toc378698858)

[JavaScript Best Coding Practices 6](#_Toc378698859)

[I. Naming Conventions 6](#_Toc378698860)

[II. Variable Declaration and Scope 9](#_Toc378698872)

[III. Namespaces 10](#_Toc378698877)

[IV. Constructors 11](#_Toc378698884)

[V. Equality Checking and Conditionals 13](#_Toc378698892)

[VI. Functions and Closures 14](#_Toc378698896)

[VII. Code Indentation and Whitespace 16](#_Toc378698902)

[VIII. Operators 17](#_Toc378698909)

[IX. DOM Efficiency 18](#_Toc378698912)

[X. Miscellaneous 19](#_Toc378698917)

[XI. JSHint 20](#_Toc378698923)

[CSS Best Coding Practices 24](#_Toc378698924)

[I. Naming Conventions 24](#_Toc378698925)

[II. Code Indentation and Formatting 25](#_Toc378698930)

[III. Selector Efficiency and Performance 29](#_Toc378698939)

[IV. CSS Units 31](#_Toc378698946)

[V. CSS3 32](#_Toc378698950)

[VI. Mobile and Responsive 33](#_Toc378698955)

[VII. Miscellaneous 35](#_Toc378698959)

[HTML5 Best Coding Practices 37](#_Toc378698964)

[I. Markup 37](#_Toc378698965)

[II. Browser Support 39](#_Toc378698979)

[III. HTML5 Features 39](#_Toc378698982)

[IV. Mobile 40](#_Toc378698987)

[V. User Awareness and Control 43](#_Toc378699001)

[VI. Usability 43](#_Toc378699003)

[VII. Rendering Performance 44](#_Toc378699007)

[VIII. Security 45](#_Toc378699016)

# Introduction

HTML5 is taking over the world. Blame it on the ubiquity of the internet, the omnipresence of JavaScript, the flourishing breed of web designers, or simply the new world order, but HTML5 is going from strength to strength.

You can now make web apps, cross-platform mobile apps, native Windows 8 apps, desktop apps, and much more using plain old HTML and JavaScript. But it’s not plain old HTML anymore, these new and upgraded platforms support HTML5, a set of new features and capabilities encompassing massive improvements in the sacred trinity of web development: HTML, JS and CSS.

In this document, we present a checklist of best practices and guidelines that have evolved as the HTML5 ecosystem continues to grow at a tremendous pace. These guidelines have been segregated into 3 sections:

* JavaScript
* CSS
* HTML5

All sections are equally relevant in making a new-age HTML5 application, independent of the underlying platform, unless otherwise mentioned.

While we discuss best practices, certain key takeaways are highlighted using noticeable icons:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **C:\Documents and Settings\ashish957\Local Settings\Temporary Internet Files\Content.IE5\K43CZBSU\MC900335720[1].wmf** | **Note** | **C:\Documents and Settings\ashish957\Local Settings\Temporary Internet Files\Content.IE5\VOV99POW\MC900098047[1].wmf** | **Important** | **C:\Documents and Settings\ashish957\Local Settings\Temporary Internet Files\Content.IE5\K43CZBSU\MC900442128[1].png** | **Warning** |

# Importance of Best Practices

Best coding practices for software development can be broken into many levels based on the coding language, the platform, the target environment and so forth. Using best practices for a given situation greatly reduces the probability of introducing errors into your applications, regardless of which software development model is being used to create that application.

In a team environment or group collaboration, best coding practices ensure the use of standards and uniform coding, reducing oversight errors and the time spent in code review.

Whether you’re writing HTML5 tags, JavaScript code, or CSS styles it is of utmost importance that you are writing ‘Good Code’ following the best practices that have evolved in the language.

Working Code, as opposed to Good Code, is easy to write. Anyone can copy code off the Internet (Stack Overflow, for example) and never even touch the depths of good design. You can easily make something work following internet tutorials and in fact, that is where most web developers stop learning.

Writing Good Code is an art. It is Good Code that clients hire Nagarro for, and it is Good Code that Nagarro hires developers for. It is this demographic, people that aspire to write Good Code and keep improving, that this document aims to address.

In our definition, the following are the characteristics of good code in order of priority:

* Reliable
* Maintainable
* Efficient

|  |  |
| --- | --- |
| C:\Documents and Settings\ashish957\Local Settings\Temporary Internet Files\Content.IE5\K43CZBSU\MC900335720[1].wmf | Reliable, maintainable and readable code provides more long-term benefit than smart, cryptic and performance-oriented code. |

It should be understood that these best coding practices are not just a way to enforce naming conventions in your code, they give you a way to analyze your source code in a multi-dimensional way. By understanding and following these practices, you can recognize patterns that have evolved over the years and benefit from the knowledge obtained and validated by industry experts.

# JavaScript Best Coding Practices

Everybody thinks they know JavaScript. They do, to a certain extent, but JavaScript is a rabbit hole that runs deep.

One of the primary reasons for the immense popularity of this language is how easy it is to start with. Anyone can start by following a basic tutorial and start writing his/her own JavaScript code in a matter of hours. But more often than not, that code will be brittle, and it is strongly recommended to adhere strictly to the following best practices because they will make your code robust and safeguard you against common errors.

## Naming Conventions

This section describes the naming guidelines that provide guidance on selecting appropriate identifiers for various kinds of variables in JavaScript, such as constructors, constants, object properties and methods, etc. Choosing identifiers that conform to these guidelines improves the readability and reusability of your code.

### Use Pascal casing for Constructor names.

|  |  |
| --- | --- |
| C:\Documents and Settings\ashish957\Local Settings\Temporary Internet Files\Content.IE5\K43CZBSU\MC900335720[1].wmf | **Pascal Casing** refers to the style of writing a concatenated string of words where first character of all words are Upper Case and all other characters are Lower Case.  For example, **B**lack**C**olor. |

function MyConstructor {

// Constructor Definition

}

### Use Camel casing for variable names and object properties.

|  |  |
| --- | --- |
| C:\Documents and Settings\ashish957\Local Settings\Temporary Internet Files\Content.IE5\K43CZBSU\MC900335720[1].wmf | **Camel Casing** refers to the style of writing a concatenated string of words where first character of the first word is Lower Case and all other first characters of subsequent words are Upper Case.  For example, **b**lack**C**olor. |

var **t**otal**C**ount = 0,

**m**y**S**tring = “my string”,

**t**oday = new Date(),

**o**bj = { **p**rop: “value” };

### Use Camel casing for function names and function arguments.

function **m**y**F**unction(**f**irst**A**rg, **s**econd**A**rgument) {

// Function Definition

}

### Use Camel casing for instance methods.

var obj = {};

obj.methodName = function (arg) {

// Method definition

}

### Do not use reserved words as variable names or property names.

|  |  |
| --- | --- |
| C:\Documents and Settings\ashish957\Local Settings\Temporary Internet Files\Content.IE5\K43CZBSU\MC900442128[1].png | Using class, default, private as keys here (respectively) won’t work in some environments. |

Bad example:

var obj = {

class: “MyConstructor”,

default: { prop: “value” },

private: false

};

Good example:

var obj = {

type: “MyConstructor”,

defaults: { prop: “value” },

hidden: false

};

Use only alpha-numeric characters as object property names, so that they can be accessed with Dot Notation.

Using Dot Notation to access property values in objects is faster than using Bracket Notation.

For example:

var obj = {

prop: “value”,

“another property”: “another value” // Although valid, but avoid

};

obj.prop // Dot Notation

obj[“another property”] // Bracket Notation

### Use ALL\_CAPS\_WITH\_UNDERSCORE to signify constants and/or global variables.

For example:

var APP\_DEFAULT\_URL = “http://localhost”;

|  |  |
| --- | --- |
| C:\Documents and Settings\ashish957\Local Settings\Temporary Internet Files\Content.IE5\VOV99POW\MC900098047[1].wmf | Always try to minimize the number of global variables in use. |

### Use Camel casing for Namespaces, just like variable names.

For example:

var myApp = {};

myApp.myValue = 2;

myApp.subNamespace = { anotherValue: 5 };

### Do not use Hungarian notation to name variables.

In Hungarian notation the name of a variable indicates its data type by using a respective prefix, like, ‘n’ for number, ‘s’ for string etc.

Bad Examples:

var nAge;

var s\_firstName;

Good Examples:

var age,

firstName;

### Use meaningful and self-descriptive words to name variables and other identifiers.

The name of the variable should not be cryptic or abbreviated. Developers tend to avoid thinking to give a proper name to a variable while defining it. A meaningful name increases the readability and understandability of the code many fold, especially for new developers.

Bad examples:

var fName;

var addr;

var pcode;

Good examples:

var firstName,

address,

pinCode;

### Always choose easily readable identifier names.

Always choose easily readable identifier names.

For example:

var horizontalAlignment;

is more readable in English than

var alignmentHorizontal;

### Name your anonymous functions as well.

Naming an anonymous function helps in debugging by providing sensible names to function calls in the Call Stack.

For example:

var log = function log(msg) {

console.log(msg);

};

## Variable Declaration and Scope

Variable Scope is a common source of confusion to new JavaScript developers. This section aims to guide the developer about variable declaration placement based on the scoping rules in the language.

### Group multiple ‘var’ declarations together for coherence.

var level = 0,

start = “a”,

end = “z”;

var i, j, k;

### Do not put any ‘var’ declarations in conditional or looping blocks.

Bad example:

if (age > 21) {

var isAllowed = true;

} else {

var isAllowed = false;

}

Good example:

var isAllowed;

if (age > 21) {

isAllowed = true;

} else {

isAllowed = false;

}

### Put all ‘var’ declarations at the top of a function definition.

JavaScript only has function scope, which means that only functions can create new variable scopes.

Hence, having all variables used in a function declared at the top of the function makes the scope of that variable more obvious at a glance.

### Do not use a variable before it is defined in code.

JavaScript hoists all variable declarations (not assignments) to the top of function scope anyway. When left to itself, this does not correspond well with how a programmer may understand his code’s flow.  
Also, Function Hoisting moves the declaration as well as the function body of a function declaration statement to the top of the current scope, which makes things even harder to understand.

Hence, always declare/assign a variable before its first use in any scope.

## Namespaces

This section contains the guidelines and standards for namespaces and namespace hierarchies in JavaScript code.

### Avoid having more than one global namespace.

Use only one global variable (for example, app) to minimize global namespace pollution. Use a module system like AMD ([RequireJS](http://requirejs.org/)) or CommonJS to eliminate global namespace pollution altogether.

### Avoid very deep namespace hierarchies.

Such hierarchies make code unreadable and unnecessarily verbose.

For example:

var show = myAwesomeApp.uiHandlers.popupWindow.messageDialog.showAsync;

### Keep a reference to a long namespace name in a local variable and use henceforth in that function scope.

For example:

function doSomething() {

var utils = myAwesomeApp.common.utils,

showHide = utils.ui.showHide;

showHide(headerElement);

var fetchOptions = utils.data.fetch.defaultOptions;

fetchOptions.type = “GET”;

fetchOptions.url = “http://localhost”;

}

### Write modular code and match file names to namespace hierarchies.

Code should be separated out in files as per the required namespaces.

For example:

myApp.utils is defined in /utils.js

myApp.lang.en is defined in /lang/en.js

### Use ‘index.js’ (by convention) to instantiate a namespace object when the namespace properties are defined in their respective files.

For example, if myApp.lang.en is defined in /lang/en.js, then

myApp.lang is initialized as an empty object in /lang/index.js

### Use dash-separated lower case file names for ‘.js’ files on file system.

Bad examples:

**Main.js**

**MainView.js**

**myController.js**

Good examples:

**main.js**

**main-view.js**

**my-controller.js**

## Constructors

JavaScript does not have formal constructors. Any function can be called with the new operator and expected to work as a constructor. Hence, there is all the more need for conventions to distinguish between regular functions and functions meant to be constructors.

### Do not use ‘new’ to create Arrays or Objects. Use literals.

Bad example:

var obj = new Object();

var arr = new Array();

Good example:

var obj = {};

var arr = [];

### Always do minimal work in a constructor.

Constructors should not do much work other than to capture the constructor parameters. The cost of any other processing should be delayed until required.

### Try to avoid constructors altogether, opting for returning plain objects instead.

Avoidable Example:

function MyClass() {

this.prop = “value”;

}

var obj = new MyClass();

Preferred way:

var obj = (function () {

return {

prop: “value”

};

}());

### Never return anything from a constructor.

In JavaScript, the value returned from a constructor using the return statement becomes the output of the new operator, instead of the object that was constructed in the constructor definition.

function MyClass() {

this.prop = “value”;

return {

prop: “another value”

};

}

var obj = new MyClass();

obj.prop // “another value”

### Minimize the number of passed arguments to the constructor.

Instead, provide an options hash that can act as a set of named arguments to easily disambiguate between the various parameters.

Bad example:

function (name, age, isPublic, isShareable, success, error, logger) {

// Function definition

}

Good example:

function (options) {

var name = options.name,

age = options.age,

isPublic = options.isPublic,

isShareable = options.isShareable,

success = options.success,

error = options.error,

logger = options.logger;

// Function definition

}

### Never augment native prototypes.

It is entirely possible to augment or change native prototypes such as Array or Object constructors.  
  
You must **never** augment or change these native prototypes as that would make the environment completely unreliable for your own code as well as third-party libraries.

### Always throw exceptions from constructors if appropriate.

Constructors should throw and handle exceptions if the passed arguments are invalid or unexpected. More generally, a constructor should not fail unexpectedly or hide any exceptions that it cannot handle.

## Equality Checking and Conditionals

Correct Equality Checking in JavaScript is not intuitive, and the intuitive way to check for equality is not correct. This section lists the quirks of the language with respect to equality checking and the best practices to follow when working with conditional blocks.

### Never use == or !=, always use === and !==.

In JavaScript, **===** and **!==** check for type as well as value, whereas **==** and **!=** check for equality *after* typecasting the operands to the same type.

In short, **==** checks for loose equality, whereas **===** checks for strict equality.  
Hence, always use **===** and **!==**.

For example:

[] == “” // true

[1] != “1” // false

null == undefined // true

[] === “” // false

[1] !== “1” // true

null === undefined // false

### Do not assume given value to be strictly equal to undefined (or null, false, etc.)

Simply use the value as a boolean check itself.

Bad examples:

if (value === undefined) {

// null, false, “”, 0, will not be able to enter this block

}

if (typeof value === “undefined”) {

// Same as before, but overkill

}

if (value === false) {

// null, undefined, “”, 0, etc will not enter this block

}

Good examples:

if (!value) {

// null, undefined, false, “”, 0, all enter the block successfully

}

### Always use curly brackets to demarcate a conditional’s block of code.

JavaScript allows you to skip the curly braces in special cases, but the same makes adding more code to the scenario quite error prone. Hence, always use curly braces.

Bad example:

if (condition) doSomething();

while (condition) doSomething();

Good Example:

if (condition) {

doSomething();

}

while (condition) {

doSomething();

}

## Functions and Closures

This section contains the guidelines for working with JavaScript functions and closures most effectively.

### Use Strict Mode.

Always use JavaScript’s Strict Mode in your own top-level closures. Strict Mode eliminates errors

(function () {

“use strict”;

// CODE

}());

|  |  |
| --- | --- |
| C:\Documents and Settings\ashish957\Local Settings\Temporary Internet Files\Content.IE5\K43CZBSU\MC900335720[1].wmf | Note that “use strict” is just a harmless string in legacy environments which do not support Strict Mode. |

|  |  |
| --- | --- |
| C:\Documents and Settings\ashish957\Local Settings\Temporary Internet Files\Content.IE5\VOV99POW\MC900098047[1].wmf | Be aware that using the Use Strict pragma in the global scope would activate Strict Mode for the global context. This is not recommended because a lot of third-party libraries included in the global scope may not satisfy Strict Mode. |

### Avoid underscore-prefixed names meant to signify private variables. Instead, go for truly private variables using closures.

JavaScript has no in-built concept of private variables. But closures can be used to effectively ‘close over’ some variables and render them private.

Bad example:

function MyClass() {

// Property considered to be private by convention

this.\_privateVal = 10;

this.publicVal = this.\_privateVal \* 2;

}

var obj = new MyClass();

Good example:

var obj = (function () {

// Truly private variable, never exposed

var privateVal = 10;

return {

publicVal: privateVal \* 2

};

}());

### Use Immediately-Invoked Function Expressions to create a new scope wherever needed.

This technique is also known as the Module Pattern when used to create a new object with some private and other publicly exposed methods or data.

For example:

var lady = (function () {

// private variable “age” closed over

// in the new created scope

var age = 20;

return {

// Exposed public method

getAge: function () { return age; }

};

}()); // Function expression invoked immediately after creation

### Never declare a function in a non-function block (if, while, etc.)

Instead, assign the function to a variable and use the variable.

Creating function scopes inside a loop is sure to create bugs if the function closes over any variable from the outer scope. This is because of our expectation to have block scope, coming from other languages, while JavaScript only has function scope.

|  |  |
| --- | --- |
| C:\Documents and Settings\ashish957\Local Settings\Temporary Internet Files\Content.IE5\VOV99POW\MC900098047[1].wmf | Any var declarations inside a conditional or looping block are actually visible even outside the block. |

### Avoid using ‘that’ or ‘self’ as a replacement for ‘this’ in a closure. Prefer a descriptive name for the object.

Bad example:

function MyClass() {

var that = this;

that.method = function () {

return that.prop;

}

}

Good example:

function MyClass() {

var myObj = this;

myObj.method = function () {

return myObj.prop;

}

}

## Code Indentation and Whitespace

This section contains guidelines on how to indent your code in order to maximize readability.

### Use soft tabs set to 4 spaces.

|  |  |
| --- | --- |
| C:\Documents and Settings\ashish957\Local Settings\Temporary Internet Files\Content.IE5\K43CZBSU\MC900335720[1].wmf | Soft Tabs refers to the setting where pressing Tab on the keyboard inserts predefined spaces in the code, instead of a Tab character. |

|  |  |
| --- | --- |
| C:\Documents and Settings\ashish957\Local Settings\Temporary Internet Files\Content.IE5\K43CZBSU\MC900442128[1].png | Never mix Tabs and Spaces in indentation. It will become a major pain-point in a team of multiple members. |

Using Soft Tabs inserts spaces instead of Tab characters and ensures that the code looks the same in any editor environment.

### Put the opening curly brace on the same line as function name.

Bad example:

function myFunc()

{

// Function body

}

Good example:

function myFunc() {

// Function body

}

### Use 1 space before a leading curly brace.

Bad example:

function myFunc(){

// Function body

}

Good example:

function myFunc() {

// Function body

}

### Use trailing commas.

Bad example (Leading commas):

var names = {

bob: “Robert”

, steve: “Steven”

, will: “William”

};

Good example (Trailing Commas):

var names = {

bob: “Robert”,

steve: “Steven”,

will: “William”

};

### Avoid lines longer than 80 characters in length.

Line length should be severely limited beyond 80 characters since code readability suffers greatly beyond that.

### Indent long chaining calls.

For example:

$(“#ideas”)

.find(“.good.idea”)

.addClass(“selected”)

.end()

.find(“.bad.idea”)

.addClass(“rejected”);

## Operators

This section lists best practices to be followed while working with various operators in JavaScript.

### When concatenating strings, always end preceding line with + operator.

Bad example:

var str = “Quick brown fox jumps over the lazy”

+ “dog, quick brown fox jumps over the”

+ “lazy dog.”;

Good example:

var str = “Quick brown fox jumps over the lazy” +

“dog, quick brown fox jumps over the” +

“lazy dog.”;

### Use ‘default’ operator to assign default values to variables and function parameters.

For example:

function calculateAge(num, unit, multiplyingFactor) {

num = num || 30;

unit = unit || “years”;

var factor = multiplyingFactor || 2;

return String((num \* factor) + “ “ + unit);

}

## DOM Efficiency

Browser DOM API is inherently slow and unintuitive, yet DOM is where most JavaScript code spends its life time. In today’s world of JavaScript apps on mobile devices, it is imperative that your DOM queries and manipulation be efficient and much under the 60fps limit for a smooth experience.

The following guidelines provide ways to make your DOM code faster.

### Instead of applying CSS properties directly on a DOM element, apply a CSS class containing all the style properties.

Apart from separating behavior from presentation, this approach minimizes browser reflows.

DOM API is a synchronous API. Hence, each styling operation which causes a reflow stops code execution, completes reflow and then continues with JavaScript code execution. This makes the operation extremely slow in the case of multiple reflows in succession.

Applying a CSS class makes the DOM reflow only once, despite multiple CSS style properties being added simultaneously.

### Use Document Fragments to render outside of the DOM and then attach to DOM.

Multiple DOM operations in fast succession make the browser sluggish. Hence, it’s preferable to detach the target element from the DOM as a Document Fragment, change its properties, attributes, children, etc, and reattach to DOM once all operations are done.

Adding multiple elements to the DOM should also the follow the same approach. A new Document Fragment should be created with all the elements, and this container element should now be attached to the DOM in a single operation, causing only one document reflow.

### Define local variables to store DOM element references when using the same element multiple times in the same function call.

Bad example:

var divs = document

.getElementById("container")

.getElementsByTagName("div");

var spans = document

.getElementById("container")

.getElementsByTagName("span");

Good example:

var doc = document,

container = doc.getElementById("container");

var divs = container.getElementsByTagName("div"),

spans = container.getElementsByTagName("span");

### Avoid DOM reflows.

There are some DOM properties which makes the DOM reflows on change like font, visibility changes which can cause changes at every level of the tree – all the way up to the root, and all the way down into the children of the unmodified node. This leads to more time being spent performing reflow.

## Miscellaneous

This section contains assorted best practices that must be followed in JavaScript code at large.

### Use Semicolons after all expression statements (not if / while / function blocks).

Do not depend on Automatic Semicolon Insertion. Use JSLint / JSHint to make sure you’re not missing semicolons at key places.

### Never use ‘eval’.

They say, ‘Eval is evil.’

Only library writers may even need to use ‘eval’. It is a can of worms and must always be avoided by the average developer.

### Always use ‘hasOwnProperty’ as the first check in ‘for… in’ loops.

Using ‘hasOwnProperty’ ensures that you don’t iterate over inherited properties in the object.

For example:

for (var key in obj) {

if (obj.hasOwnProperty(key)) {

// Looping code

}

}

### Always leave a new line before a single-line comment. Comments must add value.

The comment must not simply state the line of code in English, but add insight and value to the line of code.

The comment should make it clear to a new developer why the line of code is in the state it is, and the thoughts and decisions that have gone into it earlier.

For example:

for (var key in obj) {

// Using hasOwnProperty prevents iteration of inherited properties

if (obj.hasOwnProperty(key)) {

// Nullify all values in object

obj[key] = null;

}

}

### Multi-line comments should be reserved for JSDoc-style formal documentation.

For example:

/\*\*

\* MyClass description in this project

\* @param {string} name The only argument to the constructor

\* @constructor

\*/

project.MyClass = function MyClass(name) {

// Constructor definition

}

## JSHint

JSHint is a code linting tool that catches JavaScript errors statically, before the code has run in any environment. It is highly recommended to squash bugs early on and for general sanity when coding in JavaScript.

The following are the recommended JSHint options that can be used in any project:

{

// Settings

"passfail" : false, // Stop on first error.

"maxerr" : 100, // Maximum errors before stopping.

// Predefined globals whom JSHint will ignore.

"browser" : true, // Standard browser globals

"node" : true,

"rhino" : false,

"couch" : false,

"wsh" : true, // Windows Scripting Host.

"jquery" : true,

"prototypejs" : false,

"mootools" : false,

"dojo" : false,

"predef" : [ // Extra globals.

"define",

"require"

],

// Development.

"debug" : true, // Allow debugger statements

"devel" : true, // Allow development statements like console.log

// EcmaScript 5.

"es5" : true, // Allow EcmaScript 5 syntax.

"strict" : false, // Require `use strict` pragma in every file.

"globalstrict" : false, // Allow global "use strict".

// The Good Parts.

"asi" : false, // Tolerate missing semicolons.

"laxbreak" : false, // Tolerate unsafe line breaks.

"bitwise" : true, // Prohibit bitwise operators (&, |, ^, etc.).

"boss" : false, // Tolerate assignments inside if, for & while.

"curly" : true, // Require {} for every new block or scope.

"camelcase" : true, // Require camelCasing

"eqeqeq" : true, // Require triple equals i.e. `===`.

"eqnull" : true, // Tolerate use of `== null`.

"evil" : false, // Tolerate use of `eval`.

"expr" : false, // Tolerate `ExpressionStatement` as Programs.

"forin" : true, // Tolerate `for in` without `hasOwnPrototype`.

"immed" : true, // Require IIFEs to be wrapped in parens

"indent" : 4, // Require indentation of 4

"maxparams" : 5, // Limit function parameters to 5

"quotmark" : "double", // Require Double quotes for strings, like JSON

"latedef" : false, // Prohibit variable use before definition.

"loopfunc" : false, // Allow functions to be defined within loops.

"noarg" : true, // Prohibit arguments.caller, arguments.callee.

"regexp" : true, // Prohibit `.` and `[^...]` in regexp.

"regexdash" : false, // Tolerate unescaped last dash i.e. `[-...]`.

"scripturl" : true, // Tolerate script-targeted URLs.

"shadow" : false, // Allows re-defining variables later in code.

"supernew" : false, // Tolerate `new function () {};`, `new Object;`.

"undef" : true, // Require variables to be declared before use.

"validthis" : true, // Allow using `this` loosely in strict mode

// Persone styling prefrences.

"newcap" : true, // Require capitalization of all constructors.

"noempty" : false, // Prohibit use of empty blocks.

"nonew" : false, // Prohibit use of constructors for side-effects.

"nomen" : false, // Prohibit use of initial underbars in names.

"onevar" : false, // Allow only one `var` statement per function.

"plusplus" : false, // Prohibit use of `++` & `--`.

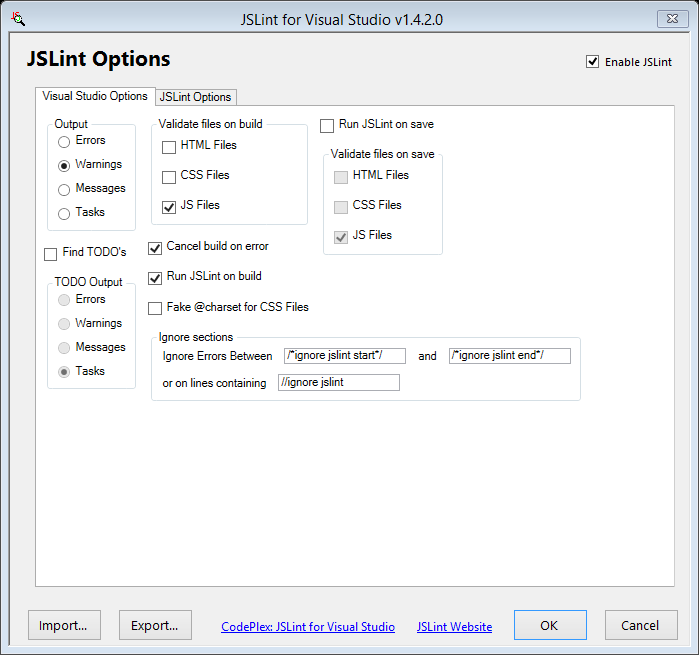
"sub" : true, // Tolerate all forms of property accessors.

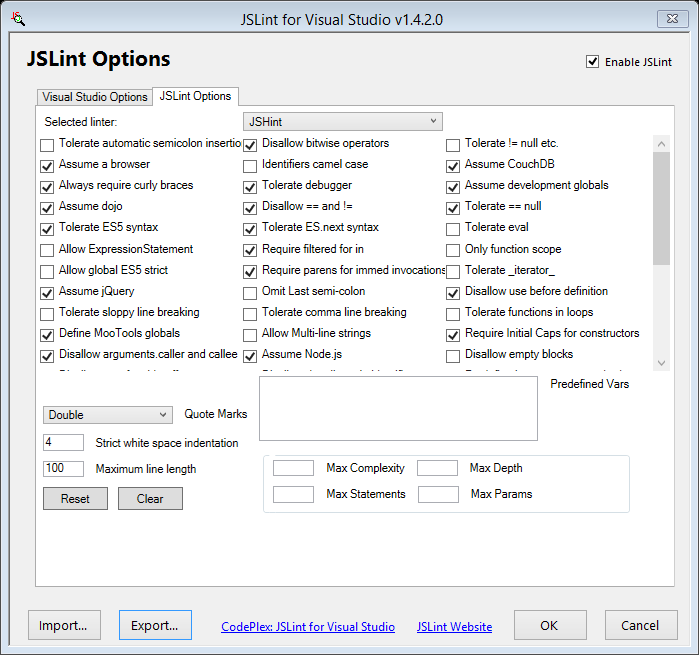
"trailing" : true, // Prohibit trailing whitespaces.

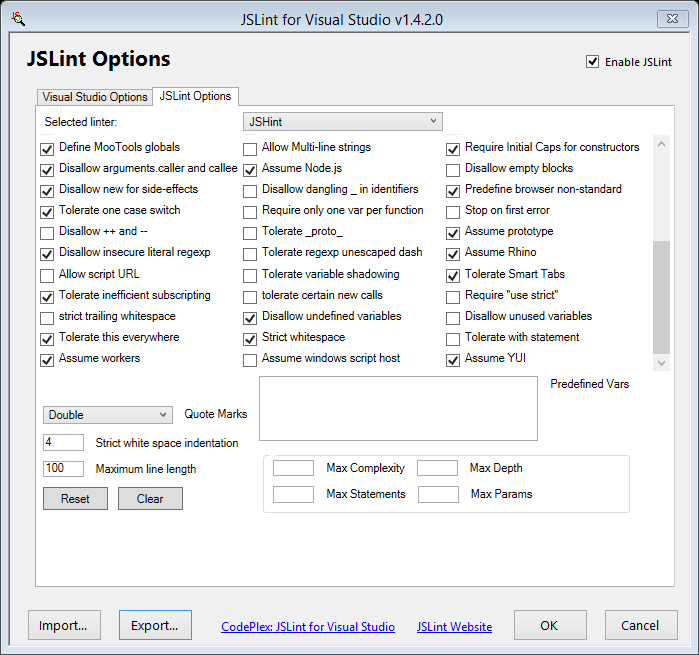
"white" : true // Check strict whitespace and indentation rules.

}

The following are the recommended settings for the **JSLint Visual Studio extension**:







# CSS Best Coding Practices

CSS3 is all the rage. And with great power comes great responsibility. CSS development presents some of the same pitfalls as JavaScript development. Developers tend to take the CSS component of the project comparatively lightly, and pay for it later in the day fixing UI bugs.

With all the power of CSS, it is possible to achieve things that could only be done with JavaScript earlier. It is imperative, hence, to write readable, maintainable, cross-browser and less hacky CSS.

## Naming Conventions

This section describes the conventions for naming of CSS selectors and CSS files.

### Use dash-seperated lower case names for CSS Classes, IDs and attributes in selectors.

For example:

.class-name,

#id-name,

[attribute-name=”value”] {

color: red;

}

### Use dash-separated lower case file names for ‘.css’ files on file system.

Bad examples:

**ResponsiveStyles.css**

**responsiveStyles.css**

**responsive\_styles.css**

Good example:

**responsive-styles.js**

|  |  |
| --- | --- |
| C:\Documents and Settings\ashish957\Local Settings\Temporary Internet Files\Content.IE5\K43CZBSU\MC900335720[1].wmf | Dash-separated lowercase text is also known as **title-case** text. |

### Use semantic class names, keeping the style implementation details strictly in the class definition.

Bad example:

.red {

color: red;

}

Good Example:

.banner {

color: red;

}

|  |  |
| --- | --- |
| C:\Documents and Settings\ashish957\Local Settings\Temporary Internet Files\Content.IE5\VOV99POW\MC900098047[1].wmf | You can take semantic to simply mean “meaningful”. The class names must ideally be a business term, while the style properties (implementation details) exist only in the definition of the class. |

### Never name the CSS class simply as a shorthand for the style property contained inside.

Bad example:

.red {

color: red;

}

.taC {

text-align: center;

}

.taC {

text-align: center;

}

Good example:

.banner {

color: red;

text-align: center;

}

|  |  |
| --- | --- |
| C:\Documents and Settings\ashish957\Local Settings\Temporary Internet Files\Content.IE5\VOV99POW\MC900098047[1].wmf | Class names should be semantic, not a replica of the style properties contained inside. |

## Code Indentation and Formatting

This section specifies the conventions regarding formatting and related aspects of CSS code.

It is strongly recommended to adhere to these guidelines strictly in order to make your code more readable and maintainable by other developers.

### Use one selector per line when a rule-set has a group of selectors separated by commas.

Bad example:

.class-name, #id-name, [attribute-name=”value”] {

color: red;

}

Good example:

.class-name,

#id-name,

[attribute-name=”value”] {

color: red;

}

### Put the opening curly brace of definition on the same line as selector.

And use 1 space before the opening curly brace.

Bad example:

.class-name

{

color: red;

}

Good example:

.class-name {

color: red;

}

### Put only one CSS property on each line in the ruleset declaration block.

Bad example:

.class-name {

color: red; border: **0**; background: green;

}

Good example:

.class-name {

color: red;

border: **0**;

background: green;

}

|  |  |
| --- | --- |
| C:\Documents and Settings\ashish957\Local Settings\Temporary Internet Files\Content.IE5\VOV99POW\MC900098047[1].wmf | You may group related style properties in a ruleset by using a blank line to separate them. |

### Use soft tabs set to 4 spaces.

|  |  |
| --- | --- |
| C:\Documents and Settings\ashish957\Local Settings\Temporary Internet Files\Content.IE5\K43CZBSU\MC900335720[1].wmf | Soft Tabs refers to the setting where pressing Tab on the keyboard inserts predefined spaces in the code, instead of a Tab character. |

|  |  |
| --- | --- |
| C:\Documents and Settings\ashish957\Local Settings\Temporary Internet Files\Content.IE5\K43CZBSU\MC900442128[1].png | Never mix Tabs and Spaces in indentation. It will become a major pain-point in a team of multiple members. |

Using Soft Tabs inserts spaces instead of Tab characters and ensures that the code looks the same in any editor environment.

### Separate unrelated ruleset declaration blocks by one or more blank lines.

Bad example:

.class-name {

color: red;

}

#id-name {

border: **0**;

}

Good example:

.class-name {

color: red;

}

#id-name {

border: **0**;

}

### Group related rulesets together, without separating them with blank lines or intervening unrelated rulesets.

Bad example:

.header {

color: red;

}

#id-name {

border: **0**;

}

.header .logo {

border: **4px**;

}

Good example:

.header {

color: red;

}

.header .logo {

border: **4px**;

}

#id-name {

border: **0**;

}

### Use double quotes around attribute values in selectors.

Bad example:

[attribute-name=value] {

color: red;

}

Good example:

[attribute-name=”value”] {

color: red;

}

### Comma-separated shorthand property values must be broken into separate lines around the comma.

Bad example:

.selector {

background-image:

linear-gradient(**#fff**, **#ccc**), linear-gradient(**#f3c**, **#4ec**);

box-shadow:

**1px** **1px** **1px** **#000**, **2px** **2px** **1px** **1px** **#ccc** inset;

}

Good example:

.selector {

background-image:

linear-gradient(**#fff**, **#ccc**),

linear-gradient(**#f3c**, **#4ec**);

box-shadow:

**1px** **1px** **1px** **#000**,

**2px** **2px** **1px** **1px** **#ccc** inset;

}

## Selector Efficiency and Performance

It is important to know the tricks that make selector parsing more efficient and speedy for the browser environment. In any decently sized application, especially on mobile, selector parsing can become a performance bottleneck if not taken care of.

These guidelines help you increase your app’s performance in the context of CSS.

### Try to minimize the use of descendant selectors, instead going for the child selector.

Bad example:

.item-list .item .title {

color: red;

}

Good example:

.item-list > .item > .title {

color: red;

}

|  |  |
| --- | --- |
| C:\Documents and Settings\ashish957\Local Settings\Temporary Internet Files\Content.IE5\K43CZBSU\MC900442128[1].png | Simply replacing the descendant selector with the child selector will only work if the latter selector is a direct child of the former selector. |

### Avoid universal selectors.

CSS selector parsing happens from the right hand side of a selector expression, continuously filtering elements based on selectors on the left. For this reason, having a universal selector (**\***) as the right-most component in a selector means that all the elements in the document will be considered first and subsequently filtered to a much smaller set. This turns out to be highly inefficient for the browser.

Bad example:

.nav \* {

margin: **0**;

}

Good example:

.nav table,

.nav div,

.nav ul {

margin: **0**;

}

### Do not over-qualify id-based selectors with anything else as there can be only one element with a particular id on the page.

Bad examples:

ul#item-list {

margin: **0**;

}

#item-list.collection {

margin: **0**;

}

Good example:

#item-list {

margin: **0**;

}

### Do not over-qualify class-based selectors with tag names.

CSS class is a semantic name given to a particular styling. Specifying an element tag binds the semantic concept represented by the class name to a particular element tag name and makes it a dependency. Instead, only semantic class names should be used to specify styles, as much as possible.

Bad example:

ul.item-list {

margin: **0**;

}

Good example:

.item-list {

margin: **0**;

}

### End every property declaration with a semicolon, including the last one in a ruleset.

In CSS, semicolon at the end of the last style declaration in a ruleset is optional. But missing it increases the chances of a parsing error if the declaration is moved from its place.

Bad example:

.header {

color: red;

border: **0**

}

Good example:

.header {

color: red;

border: **0**;

}

### If present, then @charset statement should be the very first line of a CSS file.

For example:

@charset "UTF-8";

.header {

color: red;

}

## CSS Units

### Prefer to use px over em as a unit for CSS values.

All major browsers (including IE7 and IE8) support text resizing of pixel units and zooming by the user appropriately. Hence, the purpose of **em**s has already been handled well for most use cases.

|  |  |
| --- | --- |
| C:\Documents and Settings\ashish957\Local Settings\Temporary Internet Files\Content.IE5\K43CZBSU\MC900442128[1].png | IE6 does not support text resizing of pixel units on zoom. But we hope you are not making web applications for IE6 anyway. |

### Prefer to use unit-less values in line-height, as opposed to percentage or em.

Unit-less values in **line-height** provide a multiplier on the **font-size**, instead of inheriting a percentage value of their parent element.

It is preferable because it is easier to visualize text in this manner and the value is more intuitively apparent at first glance, instead of trying to figure out the parent’s properties.

### Do not specify a unit for property values equal to 0.

Bad example:

border: 0px;

Good example:

border: 0;

## CSS3

CSS3 comes with its own set of empowering possibilities. To ensure maximum browser compatibility and performance, the following is a set of guidelines that make your CSS3 code more robust.

### The un-prefixed version of CSS3 property should be the last among the prefixed ones.

Bad example:

.container {

-moz-transition: width **1s**;

-webkit-transition: width **1s**;

transition: width **1s**;

-ms-transition: width **1s**;

-o-transition: width **1s**;

}

Good example:

.container {

-webkit-transition: width **1s**;

-moz-transition: width **1s**;

-ms-transition: width **1s**;

-o-transition: width **1s**;

transition: width **1s**;

}

### Use prefixes only for properties where they are still needed.

Bad example:

.rounded {

-webkit-border-radius: **10px**;

-moz-border-radius: **10px**;

-ms-border-radius: **10px**;

-o-border-radius: **10px**;

border-radius: **10px**;

}

Good example that works for all modern devices:

.rounded {

border-radius: **10px**;

}

|  |  |
| --- | --- |
| C:\Documents and Settings\ashish957\Local Settings\Temporary Internet Files\Content.IE5\K43CZBSU\MC900335720[1].wmf | Use online CSS Prefixers (for example, <http://prefixr.com>) to check if you’re using too many prefixes or too little. |

|  |  |
| --- | --- |
| C:\Documents and Settings\ashish957\Local Settings\Temporary Internet Files\Content.IE5\K43CZBSU\MC900335720[1].wmf | Use browser capability support (using <http://caniuse.com>) to check whether the style property needs a prefix at all. |

### Enable Hardware Acceleration in transitions to enhance visual experience.

Hardware acceleration makes the browser send the relevant elements’ view computations and rendering straight to the GPU, hence making the paint cycling and rendering much faster.

An element can be enabled for Hardware Acceleration simply by setting this property on the element:

transform: translate3d(0, 0, 0);

With the vendor prefixes, the properties to be added look like this:

-webkit-transform: translate3d(0, 0, 0);

-moz-transform: translate3d(0, 0, 0);

-ms-transform: translate3d(0, 0, 0);

-o-transform: translate3d(0, 0, 0);

transform: translate3d(0, 0, 0);

|  |  |
| --- | --- |
| C:\Documents and Settings\ashish957\Local Settings\Temporary Internet Files\Content.IE5\VOV99POW\MC900098047[1].wmf | Enabling Hardware Acceleration on mobile devices affects battery life, and must be used with discretion. |

### Use CSS Transitions instead of JavaScript.

CSS Transitions are a powerful way to abstractly describe the animation path and let the animation computation and rendering be optimized by the browser.

JavaScript animations are slow and clunky because the browser cannot predict the path of the animation. Since the animation path is computed at runtime by user-written code, the browser has no way to optimize it or offload the computation to the GPU.

Enabling hardware acceleration on JavaScript based animations has no effect on the smoothness of experience.

## Mobile and Responsive

The following is a collection of guidelines to follow when developing for mobile devices and varied viewport sizes. Taking care of these concerns should put you on a good footing with respect to the robustness of the responsive CSS code in your HTML5 application.

### Use fluid layouts for page structure and avoid setting column-widths in pixels.

Bad example:

.page-container {

width: **960px**;

}

Good example:

.page-container {

width: **100%**;

}

### Use CSS3 Media Queries to customize page content layout based on target device.

CSS3 Media Queries provide a way to activate specific CSS code based on the screen width of the target device.

For example:

@media (max-width: **767px**) {

*/\* Smaller devices \*/*

.sidebar { display: none; }

}

@media (min-width: **768px**) and (orientation: landscape) {

*/\* Specific screen width and orientation targeted \*/*

.sidebar { display: block; }

}

|  |  |
| --- | --- |
| C:\Documents and Settings\ashish957\Local Settings\Temporary Internet Files\Content.IE5\VOV99POW\MC900098047[1].wmf | Screen width alone is not enough to judge the device at hand. Pixel Density should also be taken into account to differentiate between Retina and Non-Retina devices. |

### Use Media Queries to apply retina-specific background-image properties.

Retina devices are high resolution devices where Pixel Density is increased by a factor of 2 to accommodate the larger number of pixels into the same device width.

Media Queries can help us distinguish between a retina and a non-retina device.

For example:

@media only screen and (device-width: **320px**) {

.iphone-only {

background-image: url("images/logo-1x.png");

}

}

@media only screen and (device-width: **320px**) and (-webkit-min-device-pixel-ratio: **2**) {

.retina-iphone-only {

background-image: url("images/logo-2x.png");

}

}

Here, a higher resolution picture is loaded by the browser in the case of retina device, as compared to a regular non-retina device.

## Miscellaneous

The following is a collection of important best practice guidelines that cannot be segregated into one of the categories above. It is very important to follow these guidelines for healthy CSS code.

### Prefer to apply text-transform property to convert strings to all Lowercase or Uppercase, instead of storing/typing as such.

Being all Uppercase or all Lowercase is a matter of styling and should be done with CSS.

Writing text in markup as such, or storing the strings in Database as all lowercase or uppercase makes them less portable.

### Never use inline styles, neither directly with the HTML elements nor in the style tag.

Inline styles become impossible to manage as soon as the application grows in size beyond a certain point. They are hard to write, harder to maintain, and can behave in unexpected ways due to unobvious style inheritance.

Bad examples:

<h1 style="color: red; border: 0">Hello</h1>

<style>

.class-name { color: red; }

</style>

Good example:

<link rel="stylesheet" href="styles.css" />

### Never use @import in the CSS file.

Always use <link> HTML tag to include a CSS file with a web page.

It is possible to include a CSS file by **@import**ing from another CSS file, but this method has issues in Internet Explorer. The whole functionality is best avoided.

Instead of **@import**ing, it is recommended to concatenate the different CSS files in a build step and provide a single CSS file to the browser environment.

Bad example:

@import url("styles.css");

Good example:

<link rel="stylesheet" href="styles.css" />

### Avoid !important modifier completely.

Well planned CSS never needs the **!important** modifier.

It is better to utilize CSS concepts like Cascading and Selector Specificity to achieve the desired effect.

Every time that you feel like using **!important**, try increasing the specificity of the selector rather.

Bad example:

.item {

color: blue;

}

.selected {

color: red !important;

}

Good example:

.item {

color: blue;

}

.item.selected {

color: red;

}

# HTML5 Best Coding Practices

HTML5 brings with it a world of possibilities. The HTML5 landscape is changing so fast that it becomes difficult to even know all the best practices out there, let alone follow them. What follows is a set of guidelines that should give you confidence in your HTML5 code, and help you increase the shelf-life of your code in this ever-changing new world.

## Markup

This section lists HTML5 best practices to be followed with regards to HTML5.

### Mention DOCTYPE at start of the page.

It is an instruction to the web browser about what version of HTML the page is written in, and HTML validators use DOCTYPE for loading the correct specs for validating the page.

For example:

<!DOCTYPE html>

### Use “print” media type in <link> tags to specify print-specific styles.

Use of “print” media type helps in selecting the print-specific style so that some colors on the page or the page layout can be changed to get better print of the page.

For example:

<link rel="stylesheet" href="print.css" media="print">

Here, print.css will be applied only when the document is being printed.

### Leave out the closing tag for elements in HTML5 that do not require a closing tag.

The following tags do not require a closing tag in HTML5:

* <br>
* <input>
* <img>
* <hr>
* <link>

So by not using the closing tags on the HTML page bandwidth can be saved which is important while working with mobile devices.

Bad example:

<input type="text" />

<br />

Good example:

<input type="text" >

<br>

### Use CSS links in <head> tag.

### Use of all *CSS* links in the <head> tag avoids Flash of Unstyled Content (FOUC) problem. FOUC is a momentary flash of unstyled page content which occurs when *CSS* is applied after an element is already rendered.

### Place javascript in <body> tag.

### There are multiple benefits of placing JavaScript at the bottom of <body> tag:

### Page rendering begins sooner.

### Defers connection limits.

### Meta tags.

Meta tags make the web page more meaningful for search engines hence providing SEO advantage with the search engines. Meta tags specify page description, keywords or other attributes of the web page.

### HTML5 block level elements.

HTML5 provides some new block level elements like <header>, <footer>, <article> and others. Use of these elements improve readability and easier code maintenance especially for selectors in CSS.

### Validate HTML.

HTML code validation reduces cross-browser incompatibilities, allows faster parsing of HTML code by browser, and improves UI consistency across browsers. Link below can be used for validation HTML pages:

(<http://validator.w3.org/>)

### Use quotes.

HTML5 provides flexibility of using quotes for attributes but it may create problem in those cases when there are spaces in attribute value. For example:

When there is an element with two classes, then two classes must be separated with spaces. In this case quotes are mandatory.

Bad example:

<p class=myClass otherClass id=someId> Start the reactor.

Good example:

<p class=”myClass otherClass” id=”someId”> Start the reactor.

### Inline validations

With the help of *pattern* attribute we can write some quickie regular expression to verify a particular textbox.

<form action="" method="post">

    <label for="username">Create a Username: </label>

    <input type="text"

       name="username"

       id="username"

       placeholder="4 <> 10"

       pattern="[A-Za-z]{4,10}"

       autofocus

       required>

    <button type="submit">Go </button>

</form>

### Use some templating engine.

### Keep element’s id unique.

Having multiple elements on the page with conflicting IDs creates several problems such as browser inconsistencies, validation skip, and inability to select some elements, among others.

## Browser Support

### Always use common fonts.

Never rely on uncommon system fonts because any uncommon font might not be supported by some browser so to make all browser support try to use common fonts.

### Provide fallbacks for Multimedia.

All browser have their own limitations on what kind of media it can handle and how. So always try to provide fallbacks for multimedia elements.

## Mobile

### Use Cookies Sparingly

Information stored in cookies, however, is sent to the server for every request and so using them for excessive amounts of data can negatively impact performance, particularly on a mobile network.

Also, in the mobile context, cookie support cannot be relied upon since it may be disabled either in the device configuration or by the mobile network. For this reason, applications should endeavor to remain functional even if cookies are unavailable.

### Use appropriate client-side storage technologies for local data.

Use of client-side storage provides following benefits:

* Application data stored locally can be displayed immediately when the application is started (without the need for a server roundtrip) allowing start-up latency to be reduced.
* By making updates locally at first and replicating changes back to the server in the background when connectivity is available.

### Replicate local data on server if necessary.

If locally stored data is not replicated on any server then in the case of a lost or damaged device all data is lost. So to prevent this scenario and to provide a consistent view across multiple devices local data should be replicated on server.

### Enable Automatic Sign-in

On a mobile device data input is difficult as compared to a desktop, so for avoiding this kind of situation, automatic sign-in should be implemented on next usage session. A sign-out options also should be present in case a user wants to log off the session.

### Use Transfer Compression

As data usage on a mobile network is limited, so it is preferable to compress the data before transferring it over a network. However the cost (in time and battery usage) of decompressing the data should be balanced against the gains in transport efficiency.

### Minimize Application and Data Size

Smaller applications will download and execute more quickly and more reliably than larger ones on constrained devices.

Bundle HTML, JavaScript and CSS files to reduce application size.

### Optimize Network Requests

Establishing the necessary connections in order to complete an HTTP request can take significantly longer on a mobile network than on a fixed network.

Consider the following possibilities when designing an application:

**Batching requests:**

Since a single request for more data is likely to provide a better user experience than several smaller requests, wherever possible, batch up multiple requests at the application level.

**Device Context:**

If supported by the device, use awareness of current connectivity (e.g. WiFi) to select an appropriate level of interaction.

### Aggregate Static Images into a Single Composite Resource (Sprites)

Doing lots of requests to get small contents (like images, icons, etc.) means multiple round-trips to the server, which is not efficient. Application should try to minimize service requests. Therefore instead of multiple small files try to download single large file.

### Include Background Images Inline in CSS Style Sheets

Server round trips can be saved if background images and gradients can be included in *CSS* as base64 encoded strings.

### Optimize For Application Start-up Time

Consider the following techniques to help minimize application start time:

* **Use Offline Technology:** Offline Web technologies (for example, AppCache) allow the resources of a Web application (its HTML, JavaScript, and CSS files) to be specified and stored locally so that the application can start without requiring a round-trip to the server.
* **Use Local Storage:**Where appropriate, store a snapshot of the last application state so it can be displayed immediately on start-up without requiring a server roundtrip.
* **Minimize Number of Local Storage Queries:** The number of local storage queries required to generate the initial view is a significant contribution to start-up latency. Try to minimize the number of local storage queries required before the first view can be displayed.

### Use Fragment IDs to Drive Application View

Web applications can switch views without a full page reload by showing and hiding sections of content. However, this means that the browser <back> button doesn't work by default, and it is not possible to link directly to specific views within an application. Usability is enhanced by enabling both of these features:

* Enabling deep links.
* Enabling the browser history provides a natural method to navigate application views that is natively supported by the browser.

### Use Meta Viewport Element To Identify Desired Screen Size

Certain classes of browser attempt to display desktop pages on a small screen by automatically zooming the display. This can be problematic for applications that have already been optimized for a small screen. The viewport *Meta* tag tells the device at what scale to render the page.

### Make Grids fluid

Always check for fluidness in between media query breakpoints because you are never sure about the device width on which the page is going to be rendered.

## User Awareness and Control

### Ensure the User is informed about Use of Personal and Device Information

Ensure that the user is informed if the application needs to access personal or device information. The user should be informed of the types of information that will be used by the application and whether / how that data will be exchanged with the server.

These notices should be provided when the user first accesses the Web application, or on first access to user information. It should provide the user with enough information to judge whether or not they want to allow the application access to their data.

## Usability

### Always show messages alert in Lehman language

Never use technical vocabulary to show messages to user in either form (alert or some message box).

### Always provide obvious features

Never leave out obvious features which must be included on a page. E.g. undo feature on some editor, pinchin/pinchout for images on touch devices.

### Provide meaningful and unique URL for every page.

All pages should have their unique and meaningful URLs so that user can easily bookmark any page for future references.

## Rendering Performance

### Use requestAnimationFrame instead of setInterval for animations.

*requestAnimationFrame* is API exposed by browser for efficient animations, it caters for different aspects related to animation like frame per seconds, refresh rate for screen, visibility of page which is having animation i.e. it you are running the animation loop in a tab that’s not visible, the browser won’t keep it running.

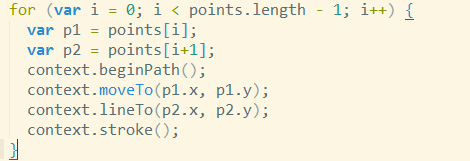
### Pre-render to off screen canvas

If you’re re-drawing similar primitives to the screen across multiple frames, as is often the case when writing a game, you can make large performance gains by pre-rendering large parts of the scene. Pre-rendering means using a separate off-screen canvas (or canvases) on which to render temporary images, and then rendering the off-screen canvases back onto the visible one.

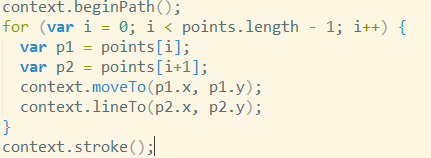
### Batch canvas calls together.

When drawing multiple lines, it's more efficient to create one path with all the lines in it and draw it with a single draw call.

Bad example:



Good example:



### Avoid unnecessary canvas state changes

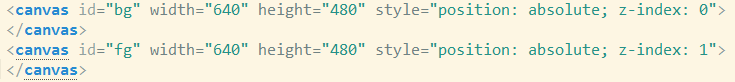
The HTML5 canvas element is implemented on top of a state machine that tracks things like fill and stroke styles, as well as previous points that make up the current path. Manipulating the state machine can also incur a performance overhead.

### Render screen differences only, not the whole new state.

As one would expect, rendering less on the screen is cheaper than rendering more. If you have only incremental differences between redraws, you can get a significant performance boost by just drawing the difference. In other words, rather than clearing the whole screen before drawing.

### Use multiple layered canvases for complex scene.

Drawing large images is expensive and should be avoided if possible. In addition to using another canvas for rendering off screen, we can also use canvases layered on top of one another, by using transparency in the foreground canvas. You might set this up as follows, with two absolutely positioned canvases one on top of the other.



The advantage over having just one canvas here is that when we draw or clear the foreground canvas, we don’t ever modify the background.

### Avoid shadowBlur.

Like many other graphics environments, HTML5 canvas allows developers to blur primitives, but this operation can be very expensive

### Avoid floating point coordinates.

HTML5 canvas supports sub-pixel rendering, and there’s no way to turn it off. If you draw with coordinates that are not integers, it automatically uses anti-aliasing to try to smooth out the lines.

## Security

### Use sessionStorage instead of localStorage whenever possible.

Use the object *sessionStorage* instead of *localStorage* if persistent storage is not needed. *sessionStorage* object is available only to that window/tab until the window is closed.

### Do not store sensitive information in localStorage or client-side database.

*localStorage* and client-side databases are directly exposed to user so never store sensitive information in these locations.

### Never show exceptions details to user.

A hacker can use exception log to get information which should not be exposed. So never show exceptions details to user always show some intended message in case of some exceptions.

### Always use HTTPS for network communication.

### Perform client-side validation.

Client side validation can save one round trip in many cases as if some input fails in client side validation then no server call required for further processing.



**COPYRIGHT NOTICE**

Copyright © 2013-2014 Nagarro Software Pvt. Ltd.

**All rights reserved.**

This document is confidential and proprietary to Nagarro Software Pvt. Ltd. and no part of this document should be reproduced, published, transmitted or distributed in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, or stored in any information storage or retrieval system of any nature nor should this be disclosed to third parties without the prior express written authorization of Nagarro Software Pvt. Ltd.