HOW TO LOSE WEIGHT *in the browser*

And what if we got together a bunch of experts who work on large sites to create a definitive front-end performance guide?

And not just one of those boring guides made for robots, what if we did something fun? What about getting together **Briza Bueno** *(Americanas.com)*, **Davidson Fellipe** *(Globo.com)*, **Giovanni Keppelen** *(ex-Peixe Urbano)*, **Jaydson Gomes** *(Terra)*, **Marcel Duran** *(Twitter)*, **Mike Taylor** *(Opera)*, **Renato Mangini** *(Google)*, and **Sérgio Lopes** *(Caelum)* to make the best reference possible?

That's exactly what we've done! And we'll guide you in this battle to create even faster sites.

— **Zeno Rocha**, project lead.

?? does performance really matter?

Of course it matters and you know it. So why do we keep making slow sites that lead to a bad user experience? This is a community-driven practical guide that will show you how to make websites faster. Let's not waste time showing **millionaire performance cases**, let's get straight to the point!

23 avoid inline/embedded code

There are three basic ways for you to include CSS or JavaScript on your page:

1) Inline: the CSS is defined inside a `style` attribute and the JavaScript inside an `onclick` attribute for example, in any HTML tag;

2) Embedded: the CSS is defined inside a `<style>` tag and the JavaScript inside a `<script>` tag;

3) External: the CSS is loaded from a `<link>` and the JavaScript from the `src` attribute of the `<script>` tag.

The first two options, despite reducing the number of HTTP requests, actually increase the size of your HTML document. But this can be useful when you have small assets and the cost of making a request is greater. In this case, run tests to evaluate if there's any actual gains in speed. Also be sure to evaluate the purpose of your page and its audience: if you expect people to only visit it once, for example for some temporary campaign where you never expect return visitors, inlining/embedded code will help in reducing the number of HTTP requests.

> Avoid manually authoring CSS/JS in the middle of your HTML (automating this process with tools is preferred).

The third option not only improves the organization of your code, but also makes it possible for the browser to cache it. This option should be preferred for the majority of cases, especially when working with large files and lots of pages.

> ***Useful tools*** / ***References***

22 styles up top, scripts down bottom

When we put stylesheets in the `<head>` we allow the page to render progressively, which gives the impression to our users that the page is loading quickly.

```
<head>
  <meta charset="UTF-8">
  <title>Browser Diet</title>

  <!-- CSS -->
  <link rel="stylesheet" href="style.css" media="all">
</head>
```

But if we put them at the end of the page, the page will be rendered without styles until the CSS is downloaded and applied.

On the other hand, when dealing with JavaScript, it's important to place scripts at the bottom of the page as they block rendering while they're being loaded and executed.

```
<body>
  <p>Lorem ipsum dolor sit amet.</p>

  <!-- JS -->
  <script src="script.js"></script>
</body>
```

> ***References***

21 try out async

To explain how this attribute is useful for better performance, it's better to understand what happens when we don't use it.

```
<script src="example.js"></script>
```

In this form, the page has to wait for the script to be fully downloaded, parsed and executed before being able to parse and render any following HTML. This can significantly add to the load time of your page. Sometimes this behavior might be desired, but generally not.

```
<script async src="example.js"></script>
```

The script is downloaded asynchronously while the rest of the page continues to get parsed. The script is guaranteed to be executed as soon as the download is complete. Keep in mind that multiple async scripts will be executed in no specific order.

> ***References***

CSS

20 *minify your stylesheets*

To maintain readable code, it's a good idea to write comments and use indentation:

```
.center {
  width: 960px;
  margin: 0 auto;
}


/* --- Structure --- */


.intro {
  margin: 100px;
  position: relative;
}
```

But to the browser, none of this actually matters. For this reason, always remember to minify your CSS through automated tools.

```
.center{width:960px;margin:0 auto}.intro{margin:100px;position:relative}
```

This will shave bytes from the filesize, which results in faster downloads, parsing and execution.

For those that use pre-processors like **Sass**, **Less**, and **Stylus**, it's possible to configure your compiled CSS output to be minified.

> ***Useful tools*** / ***References***

19 combining multiple css files

Another best practice for organization and maintenance of styles is to separate them into modular components.

```
<link rel="stylesheet" href="structure.css" media="all">
<link rel="stylesheet" href="banner.css" media="all">
<link rel="stylesheet" href="layout.css" media="all">
<link rel="stylesheet" href="component.css" media="all">
<link rel="stylesheet" href="plugin.css" media="all">
```

However, an HTTP request is required for each one of these files (and we know that browsers can only download a limited number resources in parallel).

```
<link rel="stylesheet" href="main.css" media="all">
```

So combine your CSS. Having a smaller number of files will result in a smaller number of requests and a faster loading page.

Want to have the best of both worlds? Automate this process through a build tool.

> ***Useful tools*** / ***References***

18 prefer <link> over @import

There's two ways to include an external stylesheet in your page: either via the `<link>` tag:

```
<link rel="stylesheet" href="style.css">
```

Or through the `@import` directive (inside an external stylesheet or in an inline `<style>` tag):

```
@import url('style.css');
```

When you use the second option through an external stylesheet, the browser is incapable of downloading the asset in parallel, which can block the download of other assets.

> ***References***

JAVASCRIPT

17 load 3rd party content asynchronously

Who has never loaded a third-party content to embed a Youtube video or like/tweet button?

The big problem is that these codes aren't always delivered efficiently, either by user's connection, or the connection to the server where they are hosted. Or this service might be temporarily down or even be blocked by the user's or his company's firewall.

To avoid it becoming a critical issue when loading a site or worse, lock the entire page load, always load these codes asynchronously (or use ***Friendly iFrames***).

```
var script = document.createElement('script'),

    scripts = document.getElementsByTagName('script')[0];

script.async = true;

script.src = url;

scripts.parentNode.insertBefore(script, scripts);
```

Alternatively, if you want to load multiple 3rd party widgets, you can asynchronously load them with using **this script**.

> ***Video*** / ***References***

16 cache array lengths

The loop is undoubtedly one of the most important parts related to JavaScript performance. Try to optimize the logic inside a loop so that each iteration is done efficiently.

One way to do this is to store the size of the array that will be covered, so it doesn't need to be recalculated every time the loop is iterated.

```
var arr = new Array(1000),
    len, i;

for (i = 0; i < arr.length; i++) {
  // Bad - size needs to be recalculated 1000 times
}

for (i = 0, len = arr.length; i < len; i++) {
  // Good - size is calculated only 1 time and then stored
}
```

> Results on JSPerf

> Note: *Although modern browsers engines automatically optimize this process, remains a good practice to suit the legacy browsers that still linger.*

In iterations over collections in HTML as a list of Nodes (*NodeList*) generated for example by `document.getElementsByTagName('a')` this is particularly critical. These collections are considered "live", i.e. they are automatically updated when there are changes in the element to which they belong.

```
var links = document.getElementsByTagName('a'),
    len, i;

for (i = 0; i < links.length; i++) {
  // Bad - each iteration the list of links will be recalculated to see if there was a ch
ange
}

for (i = 0, len = links.length; i < len; i++) {
  // Good - the list size is first obtained and stored, then compared each iteration
}

// Terrible: infinite loop example
for (i = 0; i < links.length; i++) {
  document.body.appendChild(document.createElement('a'));
  // each iteration the list of links increases, never satisfying the termination conditi
on of the loop
  // this would not happen if the size of the list was stored and used as a condition
}
```

> ***References***

15 *avoid document.write*

The use of `document.write` causes a dependency to the page on its return to be fully loaded.

This (bad) practice has been abolished for years by developers, but there are still cases where its use is still required, as in synchronous fallback for some JavaScript file.

HTML5 Boilerplate, for example, uses this technique to load jQuery locally if Google's *CDN* doesn't respond.

```
<script src="//ajax.googleapis.com/ajax/libs/jquery/1.9.0/jquery.min.js"></script>
<script>window.jQuery || document.write('<script src="js/vendor/jquery-1.9.0.min.js"><\/s
cript>')</script>
```

> Attention: *`document.write` performed during or after `window.onload` event replaces the entire content of the current page.*

```
<span>foo</span>
<script>
  window.onload = function() {
    document.write('<span>bar</span>');
  };
</script>
```

The result of the final page will be only *bar* and not *foobar* as expected. The same occurs when it runs after `window.onload` event.

```
<span>foo</span>
<script>
  setTimeout(function() {
    document.write('<span>bar</span>');
  }, 1000);
  window.onload = function() {
    // ...
  };
</script>
```

The result will be the same as the previous example if the function scheduled by `setTimeout` is executed after `window.onload` event.

> ***References***

minimize repaints and reflows

14

Repaints and reflows are caused when there's any process of re-rendering the DOM when certain property or element is changed.

Repaints are triggered when the appearance of an element is changed without changing its layout. Nicole Sullivan describes this as a change of styles like changing a `background-color`.

Reflows are the most costly, since they are caused by changing the page layout, such as change the width of an element.

There is no doubt that excessive reflows and repaints should be avoided, so instead of doing this:

```
var div = document.getElementById("to-measure"),
    lis = document.getElementsByTagName('li'),
    i, len;

for (i = 0, len = lis.length; i < len; i++) {
  lis[i].style.width = div.offsetWidth + 'px';
}
```

Do this:

```
var div = document.getElementById("to-measure"),
    lis = document.getElementsByTagName('li'),
    widthToSet = div.offsetWidth,
    i, len;

for (i = 0, len = lis.length; i < len; i++) {
  lis[i].style.width = widthToSet + 'px';
}
```

When you set `style.width`, the browser needs to recalculate layout. Usually, changing styles of many elements only results in one reflow, as the browser doesn't need to think about it until it needs to update the screen. However, in the first example, we ask for `offsetWidth`, which is the layout-width of the element, so the browser needs to recalculate layout.

If you need to read layout data from the page, do it all together before setting anything that changes layout, as in the second example.

> ***Demo*** / ***References***

13 avoid unnecessary dom manipulations

Everytime you touch the DOM without needing to do so, a Panda dies.

Seriously, browsing by DOM elements is costly. Although JavaScript engines are becoming increasingly powerful and fast, always prefer to optimize queries of the DOM tree.

One of the simplest optimizations is the caching of frequently accessed DOM elements. For example, instead of querying the DOM every iteration of a loop, query it once and save the result in a variable, using that every iteration of the loop instead.

```
// really bad!
for (var i = 0; i < 100; i++) {
  document.getElementById("myList").innerHTML += "<span>" + i + "</span>";
}
```

```
// much better :)
var myList = "";

for (var i = 0; i < 100; i++) {
  myList += "<span>" + i + "</span>";
}

document.getElementById("myList").innerHTML = myList;
```

```
// much *much* better :)
var myListHTML = document.getElementById("myList").innerHTML;


for (var i = 0; i < 100; i++) {
  myListHTML += "<span>" + i + "</span>";
}
```

> ***Results on JSPerf***

12 *minify your script*

Just like CSS, to maintain readable code, it's a good idea to write comments and use indentation:

```
BrowserDiet.app = function() {


  var foo = true;


  return {
    bar: function() {
      // do something
    }
  };


};
```

But to the browser, none of this actually matters. For this reason, always remember to minify your JavaScript through automated tools.

```
BrowserDiet.app=function(){var a=!0;return{bar:function(){}}}
```

This will shave bytes from the filesize, which results in faster downloads, parsing and execution.

> *Useful tools* / *References*

11 combine multiple js files into one

Another best practice for organization and maintenance of scripts is to separate them into modular components.

```
<script src="navbar.js"></script>
<script src="component.js"></script>
<script src="page.js"></script>
<script src="framework.js"></script>
<script src="plugin.js"></script>
```

However, an HTTP request is required for each one of these files (and we know that browsers can only download a limited number resources in parallel).

```
<script src="main.js"></script>
```

So combine your JS. Having a smaller number of files will result in a smaller number of requests and a faster loading page.

Want to have the best of both worlds? Automate this process through a build tool.

> *Useful tools* / *References*

JQUERY

10 always use the latest version of jquery

The core jQuery team is always looking to bring improvements to the library, through better code readability, new functionality and optimization of existing algorithms.

For this reason, always use the latest version of jQuery, which is always available here, if you want to copy it to a local file:

```
http://code.jquery.com/jquery-latest.js
```

But *never* reference that URL in a `<script>` tag, it may create problems in the future as newer versions are automatically served to your site before you've had a chance to test them. Instead, link to the latest version of jQuery that you need specifically.

```
<script src="//ajax.googleapis.com/ajax/libs/jquery/1.9.1/jquery.min.js"></script>
```

Just like the wise **Barney Stinson** says, *"New is always better"* :P

> ***References***

9 selectors

Selectors is one of the most important issues in the use of jQuery. There are many different ways to select an element from the DOM, but that doesn't mean they have the same performance, you can select an element using classes, IDs or methods like `find()`, `children()`.

Among all of them, select an ID is the fastest one, because its based on a native DOM operation:

```
$("#foo");
```

> Results on JSPerf

8 use for instead of each

The use of native JavaScript functions nearly always results in faster execution than the ones in jQuery. For this reason, instead of using the `jQuery.each` method, use JavaScript's own `for` loop.

But pay attention, even though `for in` is native, in many cases it performs worse than `jQuery.each`.

And the tried and tested `for` loop gives us another opportunity to speed things up by caching the length of collections we're iterating over.

```
for ( var i = 0, len = a.length; i < len; i++ ) {
    e = a[i];
}
```

The use of reverse `while` and reverse `for` loops is a hot topic in the community and are frequently cited as the fastest form of iteration. However it's typically avoided for being less legible.

```
// reverse while
while ( i-- ) {
  // ...
}


// reverse for
for ( var i = array.length; i--; ) {
  // ...
}
```

> ***Results on JSPerf*** / ***References***

7 *don't always use jquery...*

Sometimes vanilla JavaScript can be easier to use and have better performance than jQuery.

Consider the following HTML:

```
<div id="text">Let's change this content</div>
```

Instead of doing this:

```
$('#text').html('The content has changed').css({
  backgroundColor: 'red',
  color: 'yellow'
});
```

Use plain JavaScript:

```
var text = document.getElementById('text');
text.innerHTML = 'The content has changed';
text.style.backgroundColor = 'red';
text.style.color = 'yellow';
```

It's simple and **much** faster.

> ***Results on JSPerf*** / ***References***

IMAGES

6 *use css sprites*

This technique is all about grouping various images into a single file.

And then positioning them with CSS.

```
.icon-foo {
  background-image: url('mySprite.png');
  background-position: -10px -10px;
}

.icon-bar {
  background-image: url('mySprite.png');
  background-position: -5px -5px;
}
```

As a result, you've dramatically reduced the number of HTTP requests and avoided any potential delay of other resources on your page.

When using your *sprite*, avoid leaving too much white space between images. This won't affect the size of the file, but will have an effect on memory consumption.

And despite nearly everyone knowing about sprites, this technique isn't widely used—perhaps due to developers not using automated tools to generate sprites. We've highlighted a few that can help you out with this.

> ***Useful tools*** / ***References***

5 data uri

This technique is an alternative to using CSS sprites.

A **Data-URI** is a way to inline the content of the URI you would normally point to. In this example we are using it to inline the content of the CSS background images in order to reduce the number of HTTP requests required to load a page.

Before:

```
.icon-foo {
  background-image: url('foo.png');
}
```

After:

```
.icon-foo {
  background-image: url('data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAAAEAAAABAQMAAAAl21
bKAAAAA1BMVEUAAACnej3aAAAAAXRSTlMAQObYZgAAAApJREFUCNdjYAAAAAIAAeIhvDMAAAAASUVORK5CYII%3D'
);
}
```

All browsers from IE8 and above support this base64-encoded technique.

Both this method and the CSS spriting method require build time tools to be maintainable. This method has the advantage of not requiring manual spriting placement as it keeps your images in individual files during development.

However has the disadvantage of growing the size of your HTML/CSS considerably if you have large images. It is not recommended to use this method if you aren't gzipping your HTML/CSS during HTTP requests as the size overhead might negate the speed gains you get from minimizing the number of HTTP requests.

> ***Useful tools*** / ***References***

4 don't rescale images in markup

Always define the `width` and `height` attributes of an image. This will help avoid unnecessary repaints and reflows during rendering.

```
<img width="100" height="100" src="logo.jpg" alt="Logo">
```

Knowing this, John Q. Developer who has a *700x700px* image decides to use this technique to display the image as *50x50px*.

What Mr. Developer doesn't realize is that dozens of unnecessary *kilobytes* will be sent over the wire.

Always keep in mind: just because you can define with width and height of an image in HTML doesn't mean you should do this to rescale down large images.

> ***References***

3 optimize your images

Image files contain a lot of information that is useless on the Web. For example, a JPEG photo can have *Exif* metadata from the camera (date, camera model, location, etc.). A PNG contains information about colors, metadata, and sometimes even a miniature embedded thumbnail. None of this is used by the browser and contributes to filesize bloat.

There are tools that exist for image optimization that will remove all this unnecessary data and give you a slimmer file without degrading quality. We say they perform *lossless* compression.

Another way to optimize images is to compress them at the cost of visual quality. We call these *lossy* optimizations. When you export a JPEG, for example, you can choose the quality level (a number between 0 and 100). Thinking about performance, always choose the lowest number possible where the visual quality is still acceptable. Another common lossy technique is to reduce the color palette in a PNG or to convert PNG-24 files into PNG-8.

To improve user perceived performance, you should also transform all your JPEG files in progressive ones. Progressive JPEGs have great browser support, are very simple to create and have no significant performance penalty. The upside is that the image will appear sooner on the page (**see demo**).

> ***Useful tools*** / ***References***

BONUS

2 diagnosis tools: your best friend

If you'd like to venture into this world of web performance, it's crucial to install the **YSlow** extension in your browser—they'll be your best friends from now on.

Or, if you prefer online tools, visit the **WebPageTest**, **HTTP Archive**, or **PageSpeed** sites.

In general each will analyse your site's performance and create a report that gives your site a grade coupled with invaluable advice to help you resolve the potential problems.

1 that's it for today!

We hope that after reading this guide you can get your site in shape. :)

And remember, like all things in life, **there's no such thing as a silver bullet**. Performance tuning of your application is a worthwhile venture, but should not be the sole basis of all your development decisions—at times you'll need to weigh out the costs and benefits.

Want to learn more? Check out the **References** that we used to write this guide.

Have suggestions? Send a tweet to **@BrowserDiet** or a **pull request** on Github.

Don't forget to share with your friends, let's make a faster web for everyone. \o/

Designed by **Briza Bueno** • Illustrated by **Scott Johnson**

Made by **Zeno Rocha** • Brought to you by **Liferay**