Performance & Organization

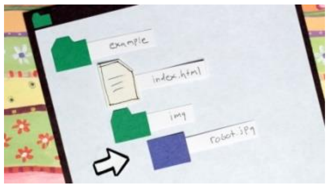
PART 1

Performance

There are many different features in a web page (web app) that can be the performance bottlenecks such as: JavaScript, or web font delivery, sluggish rendering, heavy images, heavy videos, service workers. It's hard to point out where we should start improving performance on a page and even how we can establish performance as a culture when developing new web apps.

Structure & Organization

"A code that works" is currently not enough!



Knowing the fundamentals of website performance and organization is currently a nice skill to have. The organization and architecture of the code can impact not only on the speed of development but also on the speed to render the pages. It's wise to take the time to design the right structure for a code base and identify how all the different components will work together as this will definitely speed up production and create a better experience to everybody involved (developers and users).

Website performance resembles the 80/20 rule (20% of optimizations will speed up basically 80% of the website).

It is important to identify a good strategy and structure for developing the code base and build a strong directory architecture, outlining design patterns, and finding ways to reuse common code.

The way you organize the style architecture tends to be very personal but there are some best practices that should be followed. For example: separate styles based on intent (common based styles, UI components, and business logic modules). This type of organization will make your work easier if you need to apply RWD techniques.

One way of structuring the style: normalize.css, layout.css, typography.css. The UI part could have: buttons.css, forms.css, nav.css, etc. and the business Logic Modules could have: header.css, footer.css. Another way could be that instead of having so many different CSS files, you could organize the structure within a couple or even a single CSS file.

The main objective, behind this organization issue, is to make you start thinking of websites as systems rather than simple individual pages. At some places, you will have a clear separation of the styles within the business logic module; for example, you may have a footer of a page that may have some buttons and the style for those buttons may be defined within the UI components while other styles for that same footer may be coming from the business logic module set. One of the advantages of this type of separation is that it encourages the ability for styles to be shared and reused.

Object-Oriented CSS

Object-Oriented CSS? Is CSS a programming language or anything similar to also have this object-oriented approach?

The classic way of writing CSS tends towards a sort of stylesheet craziness. It abuses of cascades, and super-mega-selectors which ends up with a harder maintenance plan.

Pioneered by Nicole Sullivan when she wrote, in her work, styles for bigger websites. OO CSS identifies two principles: Separate structure from skin; separate content from container. OO CSS aims to simplify selectors, protect the Cascade, with the main objective of faster, more efficient and maintainable stylesheets. The way this is done is by combining two principles: Separate Structure and Skin and Separate Containers and Content.

Every webpage has four things in common: **structure**, **skin**, **containers**, and **content**. Classic CSS basically makes no effort in distinguishing between these and this is the main cause of the many different problems developers face extending the time of maintenance.

Separate Structure from Skin – abstract the layout of an element away from the theme of a website – the structure of a module should be transparent to allow other styles to be inherited and displayed without conflict. The structure of the page revolves around element size and positioning and the structural properties are generally:

- display
- position
- margin
- padding
- height
- width
- overflow

The skin properties tend to be more visual properties of an already positioned elements and generally they are:

- font
- color
- gradient
- shadow

These two categories create the first level of abstractions in OO CSS. The CSS selectors and files separated by structure and skin can be included or applied separately, independently. This means that if a we need to change the color scheme, you would only change the skin properties or if you need to change the layout of a website, you would only change the structure properties. As you can see, the maintenance becomes more direct and then faster.

Separate containers and content – removing dependency of a parent element nesting children elements – example: a heading should look the same regardless of its parent container – this means that elements need to inherent default styles, then be extended with multiple classes as necessary. This principle mainly supports code reuse. For example: if you see something as***class="btn-medium btn-red"*** you understand right away that this will be like a medium red button, no matter to which container it's being applied - simply because both button styles (the structure class of **btn-medium** and the skin class **btn-red**) would be created independently of any type of container.

So, instead of writing:
```
#navbar {
  padding: 5px;
  margin: 5px;
  position: absolute;
  left: 0;
  width: 200px;
}
#navbar .list {
  margin: 3px;
}
```

```
#navbar .list .toplist {
  font-size: 1.3rem;
  color: blue;
}
#navbar .list .bodylist {
  font-size:  1rem;
  color: black;
  background-color: yellow;
}
```

You should have:

```
#navbar {
  padding: 5px;
  margin: 5px;
  position: absolute;
  left: 0;
  width: 200px;
}
.list {
  margin: 3px;
}
.toplist {
  font-size: 1.3rem;
  color: blue;
}
.bodylist {
  font-size:  1rem;
  color: black;
  background-color: yellow;
}
```

This second structure is to point out that no style should depend on its container which means that a blue header should be a blue header no matter where it is applied in the page.

Look at **ex1.html** – OO CSS defends building a component library, staying flexible, and using a grid. These simple rules can help you avoid the need to add additional styles every time you add a new page and/or a new feature to a website.

For deeper information on Object-Oriented CSS, take a look at the website http://oocss.org/

Scalable and Modular Architecture for CSS

Scalable and Modular Architecture for CSS
A flexible guide to developing sites small and large.

Also known as SMA CSS. It was developed by Jonathan Snook. He defends breaking up styles into five core categories: **Base**, **Layout**, **Module**, **State**,**Theme**.

Base – includes core element styles covering the general defaults. The CSS resets are a good example of base styles

Layout – identifies the sizing and grid styles of different elements to determine their layout. There are major and minor layout components in every design, for example a header block might be a major component while the logo and headline inside of this block might be minor components. The layout rules will apply to major components.

Module – more specific styles targeting individual parts of the page (ex.: navigation) or feature styles. It targets the minor components cited previously. They tend to be shown inside layout components and sometimes even within modules. We tend to avoid IDs and element selectors in this part.

State – used to augment or override other styles (ex.: to an active tab)

Theme – include styles based around the skin, or look & feel of the modules. It would define color and typography across a site and when you separate themes into their own set of styles, they become easier to be modified.

Jonathan Snook offers some naming convention when working with SMA CSS but he does not say that you need to follow his convention, but it is advisable to have one for consistency. His convention is:

- **Base** - nothing needed
- **Layout** - use ***l-*** or ***layout-*** as prefixes
- **State** - use ***is-*** as prefix
- **Module** - just use the module name (example: .callout) instead of trying to prefix each module but related modules receive a prefix in order to organize them.

Consider the following CSS:

#maindiv.article > #main > #content > #intro > p > strong { }

You have here a 6-generation depth. The greater the depth, the greater the dependency on the HTML structure. This is what we want to avoid because it means that page elements cannot be moved and that more duplication of code is extremely likely to happen.

Take a look at **ex2.html** – the alert class falls into the module category while the error-true falls into the state category – the styles from each of these categories are then inherited as needed.

For deeper information about SMA CSS, you can also look at SMA CSS home page

Performance Driven Selectors

Developers abuse of selectors! We should remember that how elements are selected within CSS affects performance.

Keep selectors short – by doing this, you are minimizing specificity which allows better inheritance and portability. When you use long/over qualified selectors, you are reducing performance because they force the browser to render each individual selector type from right to left and they also put a burden on all other selectors

Example:

```
header nav ul li a {…}      /* this is bad */

.primary-link {…}           /* better */

button strong span {…}      /* this is ok */
button strong span .callout {…}      /* but this is not good */

button span {...}       /* getting better */
button .callout {…}     /* much better */
```

The first example is the worst one because it does not even allow you to change the location of that anchor tag! If that anchor tag should move to footer, for example, the CSS properties specified would not work or you would need to change header to footer but worse would be if you would need to create an anchor tag in a footer or section element with exactly the same CSS properties, then you would need to repeat the whole set of properties to the new element or would make that line even longer by adding the comma (,) and then adding the "path" to the next anchor tag!

The main objective of shorter selectors is to **decrease specificity** which then creates a cleaner and more shareable code.

Favor classes but pay attention to some details: since selectors are rendered from right to left (when reading the CSS), it’s important to keep an eye on the **KEY SELECTOR** – the unit at the end, the one furthest to the right. The key selector identifies the first element a browser will find. So, when you have a poor key selector, the browser will go on a “wild hunt”. Another detail: avoid prefixing the class selectors with an element – it will prohibit the style (the class) to be applied to a different element

Examples:

Bad	**Good**
#container header nav {…}	.primary-nav {…}
article.feat-post {…}	.feat-post {…}

Whenever possible stay away from ID selectors – it is too specific (you cannot use the same ID more than once within the same page, right?). The exception is if your pages would be really short and you would have very specific areas (recognized by the id's) in those short pages and those areas would need always to look the same.

Specificity

This is an important, and sometimes forgotten, part of CSS. Imagine you have the following code:

```
<body>
  ...
  <ul id="listclass">
    <li>CNIT 132</li>
    <li>CNIT 133</li>
    <li>CNIT 132A</li>
  </ul>
  ...
</body>
```

In the CSS you have something as (of course, you would have other lines of code in the CSS file as well as in the HTML document - that's what the ... is representing):

```
...
#listclass li { font-weight: bold;  font-size: 1.2em;  color: blue; }
...
```

Somebody comes to you and ask you to have only the CNIT 132A with a red color. In general, the first thought would be to create a class such as:

```
.favoritem { color: red; }
```

And then apply that class to the li element in the HTML document that then would become:

```
<body>
  ...
  <ul id="listclass">
    <li>CNIT 132</li>
    <li>CNIT 133</li>
    <li class="favoritem">CNIT 132A</li>
  </ul>
  ...
</body>
```

If you test this code, you will notice that it will NOT work and the color will remain blue. This is due to what is called specificity and it's something that creates confusion among developers. In our case here, the class name has a lower specificity than the id name. And then, after you figure this out, you would write the CSS as:

```
...
#listclass li { font-weight: bold;  font-size: 1.2em;  color: blue; }
#listclass li.favoritem { color: red; }
...
```

How exactly specificity works?
There is a system to give values depending on what you have in the selector. It works like this:
STYLE ATTRIBUTE > ID > CLASS / PSEUDO-CLASS / ATTRIBUTE > ELEMENT

This shows that the **style** attribute is the highest in specificity and that **ID** is higher than **class** or **pseudo-class**, or **attribute** and the lowest one is the element.

In summary, this rule works like this:

- If the element has inline styling (uses the style attribute), that automatically wins - it will get 1,0,0,0 points
- For each ID value, you will have 0,1,0,0 points
- For each class, or pseudo-class, or attribute selector, you will have 0,0,1,0 points
- For each element, you will have 0,0,0,1 point

Notice that the commas (,) are just separating the groups of possibilities that we listed above, you could use the value without the commas (,).

Let's look at the example below:
ul#menu li.favorite a

What would be the "value" of that selector? It has 3 elements, 1 class, and 1 ID. So the value would be 0,1,1,3 (or 0113). If in your HTML you have <li style="color:red;">, the value for it would be 1,0,0,0 (or simply 1000). See? The 1000 value is, of course, bigger (higher) than 0113 and that li element would then get have the red color.

There are other important points to remember:

- The universal selector (*) has no specificity value, meaning, its value is 0,0,0,0 (zero)
- Pseudo-elements, for example ::first-line gets the same value of elements 0,0,0,1 while pseudo-classes, as seen before, gets the 0,0,1,0 which is the same value as classes
- The !important value appended to a CSS property is an automatic winner and the only way to win over this is coding another **!important** below the one you want to win over. Just remember that **it's not a very good idea to keep using !important just for the winning reason**!

Reusable Code

You should, as much as possible, reuse styles. For example: if two modules use the same background-color, rounded corners, font-size, there is no reason to explicitly state those styles twice, instead, they can be combined within a single class.

The reusing of code does not mean that there is a cost for semantics. One technique is to pair selectors together (separated by a comma); and another technique includes binding styles to one class then using multiple classes on the same element (frequently used in OO CSS and SMA CSS methodologies).

Example:

```
/* not good */
.news { background-color: yellow; border-radius:10px;}
.social {background-color: yellow; border-radius:10px;}

/* good */
.news, .social {background-color: yellow; border-radius:10px}

/* better */
.news-social {background-color: yellow; border-radius:10px;}
```

What matters is that **the code is being shared and reused and the overall size of the file is reduced**.

Another good example can be seen below:

```
/* not good */

h1 { margin: 30px 0; }

.mgbottom { margin-bottom: 30px; }

.mgtop { margin-top: 30px; }
```

Imagine that this CSS would be much longer with other elements or classes using those same px units! Imagine now that we would need to change all of those to be 40px or any other value. See? Now you would need to go all over your CSS file looking for each one to be changed. Instead, you could have something like this below using **variables**:

```
/* great */

$base-font-size: 16px;
$base-line-height: 1.5;
$spacing-unit: $base-font-size * $base-line-height;

h1 { margin: rem($spacing-unit); }

.mgbottom { margin-bottom: rem($spacing-unit); }

.mgtop { margin-top: rem($spacing-unit); }
```

CSS Variables:

CSS is not a programming language and as such has lacked support for native variables since its inception. Unless you would be using pre-processors such as Sass, you would write your CSS with no variables at all.

The web is moving really fast and it’s not a surprise that now CSS finally supports variables. Of course, pre-processors support a lot more features but the addition of CSS variables to the CSS package is, indeed, a good one.

We will learn more about CSS variables in another part of this course, so, hang in there!

Minify & Compress Files

Removing duplication and unnecessary code is the best way to cut down on file size but minifying files such as HTML, CSS, and JavaScript are other ways too.

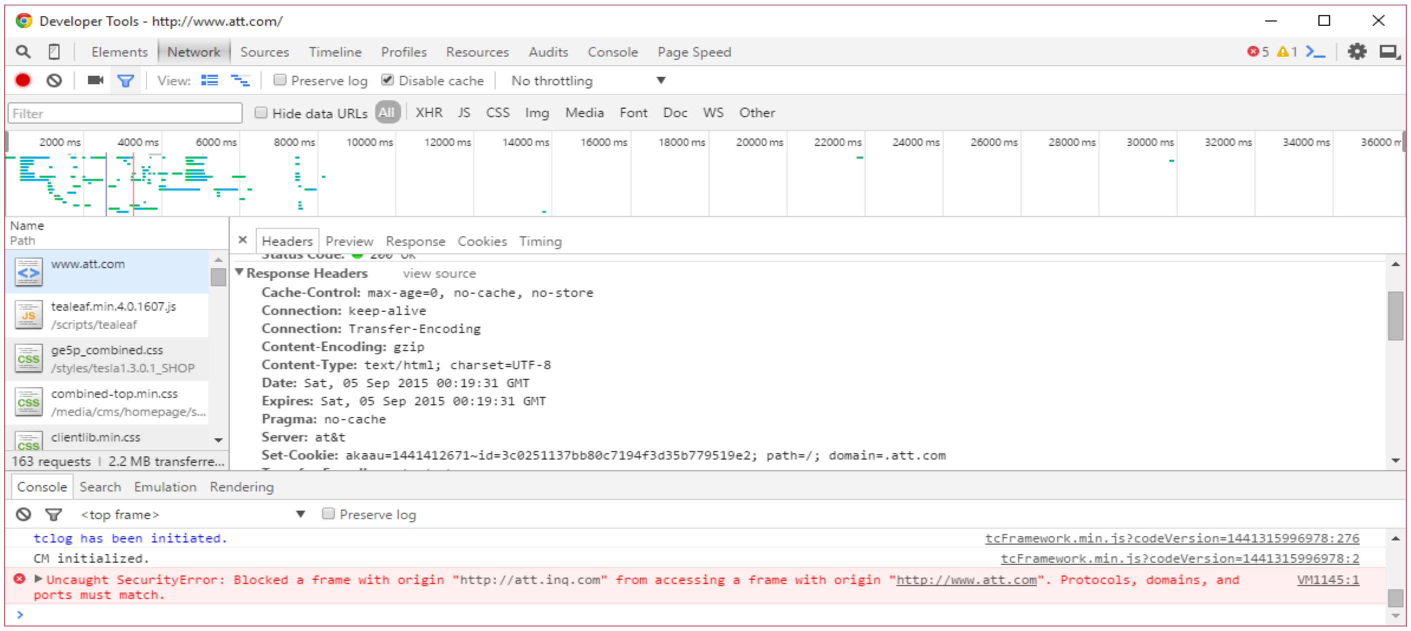
To compress files, there are some tools such as **gzip Compression** – it takes common files and identifies similar strings to compress down which will then up sending a smaller file from the server to the browser. Setting this up is not complicated and HTML5 Boilerplate has done a very good job with it. You will need to add, at the root directory of the web server, an **.htaccess** file labeling the specific files to be **gzipped** and the dot (.) at the beginning of the file name is correct as the **.htaccess** file is a hidden file.

More technical details on how the gzip file is done can be learned in this video - (about 59 min).

The .htaccess files **only work on Apache** servers and some modules need to be enabled.

There is a way that you can identify if gzip compression is enabled in a website. Using CDT (Chrome Developer Tools), inside **Network** tab, you can see an image like the one below:

To get to this screen, you need to open the page (the page opened was http://www.att.com) on the browser, then open CDT and click on the Network tab. Then refresh the page and then on the top list of ***Name / Path*** click once on the first item that is shown there and then click on the **Headers** and you will see the **Content-Encoding**.


Developer Tools - http://www.att.com/
Elements Network Sources Timeline Profiles Resources Audits Console Page Speed
View: Preserve log Disable cache No throttling
Filter Hide data URLs All XHR JS CSS Img Media Font Doc WS Other
2000 ms 4000 ms 6000 ms 8000 ms 10000 ms 12000 ms 14000 ms 16000 ms 18000 ms 20000 ms 22000 ms 24000 ms 26000 ms 28000 ms 30000 ms 32000 ms 34000 ms 36000 ms
Name Path
Headers Preview Response Cookies Timing
Console Search Emulation Rendering
<top frame> Preserve log

```
www.att.com
tealeaf.min.4.0.1607.js   /scripts/tealeaf
ge5p_combined.css   /styles/tesla1.3.0.1_SHOP
combined-top.min.css   /media/cms/homepage/s...
clientlib.min.css
163 requests | 2.2 MB transferre...

▼Response Headers    view source
   Cache-Control: max-age=0, no-cache, no-store
   Connection: keep-alive
   Connection: Transfer-Encoding
   Content-Encoding: gzip
   Content-Type: text/html; charset=UTF-8
   Date: Sat, 05 Sep 2015 00:19:31 GMT
   Expires: Sat, 05 Sep 2015 00:19:31 GMT
   Pragma: no-cache
   Server: at&t
   Set-Cookie: akaau=1441412671~id=3c0251137bb80c7194f3d35b779519e2; path=/; domain=.att.com
```

```
tclog has been initiated.                     tcFramework.min.js?codeVersion=1441315996978:276
CM initialized.                               tcFramework.min.js?codeVersion=1441315996978:2
▶Uncaught SecurityError: Blocked a frame with origin "http://att.inq.com" from accessing a frame with origin "http://www.att.com". Protocols, domains, and   VM1145:1
ports must match.
>
```

Minifying HTML, CSS, JavaScript Code

One simple way to save some bytes and make the download of your files faster is by minifying the files you are using: *.html, *.css, and *.js files.

You need to remember that one of the ways that Google will rank your pages is by the speed of the page load. Of course, your users will evaluate your website by this measurement too, so it's important to save bytes wherever you can to make your pages load faster.

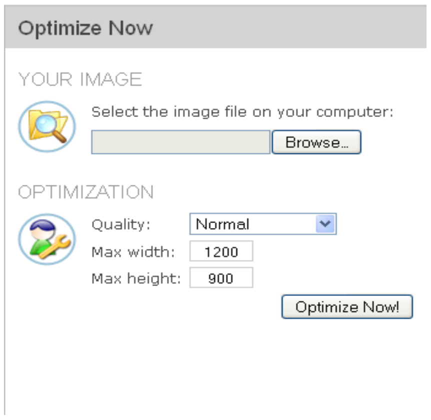
The HTML, CSS, and JavaScript code files are created so human beings can read and maintain those files and then they contain extra spaces, comments, formatting that the servers, neither the browsers, need to process and render the content.

Minify means remove any unnecessary characters that are not used, nor needed, for the code to be executed. Of course, this will speed up the loading of the pages which will then make users and also search engines "happy"! The minifying process will generally remove: white spaces, new line characters, comments, and block delimiters.

Minifying code is not the same as compressing the data. Compressing requires an un-compression step before the code can be executed. This extra step adds to the time before the code can run. On the other hand, minified code can be executed as-is.

There are many different ways to minify code and one of those is found at Minify Code website

Image Compression



Select the image file on your computer:

Quality: Normal
Max width: 1200
Max height: 900

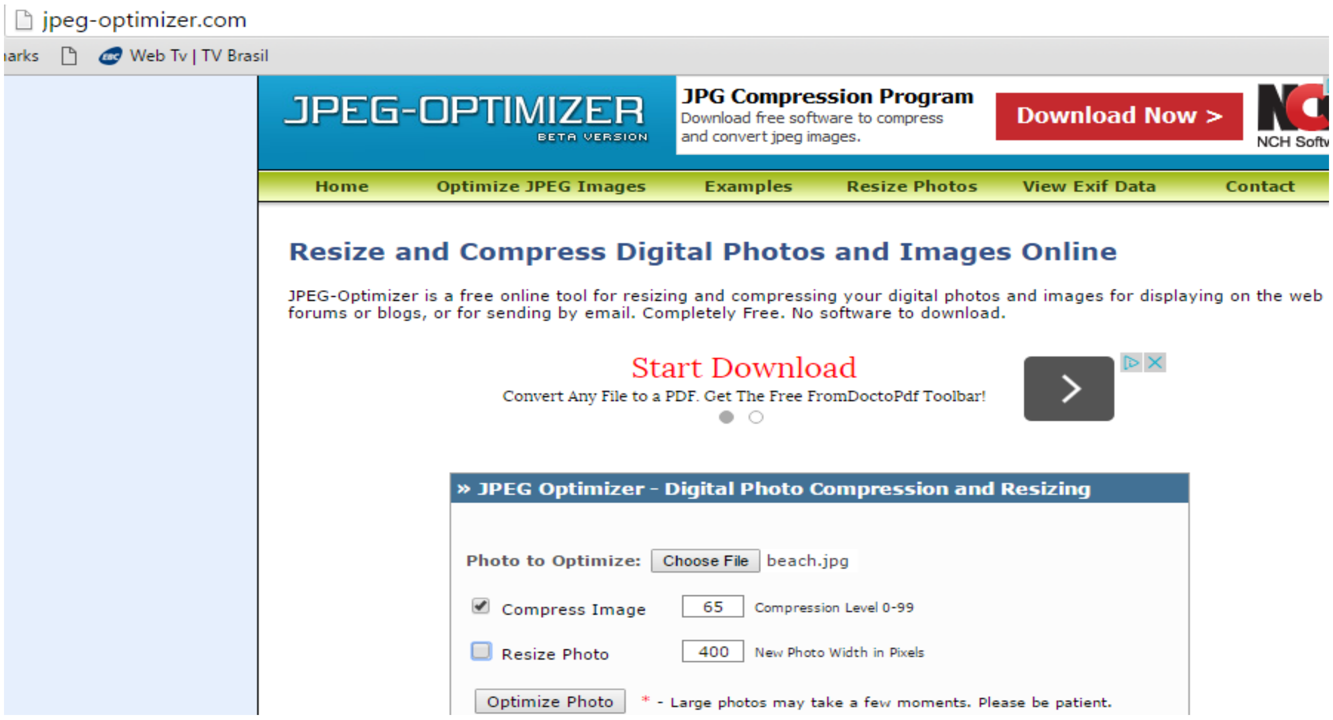
Images should be compressed too or at least optimized. There are many tools out there that you can use for that purpose: Adobe Photoshop,JPEGmicro, ImageOptim (for Mac), PNGGauntlet (for Windows), and online you can use jpeg-optimizer or Squoosh

Some fear that compressing images involves reducing quality and jpg images are, for sure, types of images that lose some quality when reduced but nothing that your users would notice. Some of the software mentioned above apply compression with some type of lossless techniques.

Setting the image's height and width attributes **DOES HELP** render the page quicker as the browser sets aside the appropriate space for the image and moves on to render other elements while the image downloads. These attributes **SHOULD NOT** be used to shrink images **-** if you need a smaller image, it's better to edit the image to the size you need and you may even, with that, "save some bytes".

If you look at the folder of files you have, you will notice that there is one jpg file named **beach.jpg** that we will be using for this next exercise.

The **beach.jpg** will be optimized using the online jpeg-optimizer. Notice in the image below that, after I used the ***Choose File*** button to choose **beach.jpg**, I uncheck the box that says "Resize...." as I wanted to keep the same width and height of the original image. I also left in 65 the compression level which means 65%. Then, I clicked the **Optimize Photo** button to optimize the **beach.jpg**.


jpeg-optimizer.com

JPEG-OPTIMIZER BETA VERSION

JPG Compression Program
Download free software to compress and convert jpeg images.

Download Now >

Home | Optimize JPEG Images | Examples | Resize Photos | View Exif Data | Contact

Resize and Compress Digital Photos and Images Online

JPEG-Optimizer is a free online tool for resizing and compressing your digital photos and images for displaying on the web forums or blogs, or for sending by email. Completely Free. No software to download.

Start Download
Convert Any File to a PDF. Get The Free FromDoctoPdf Toolbar!

» JPEG Optimizer - Digital Photo Compression and Resizing

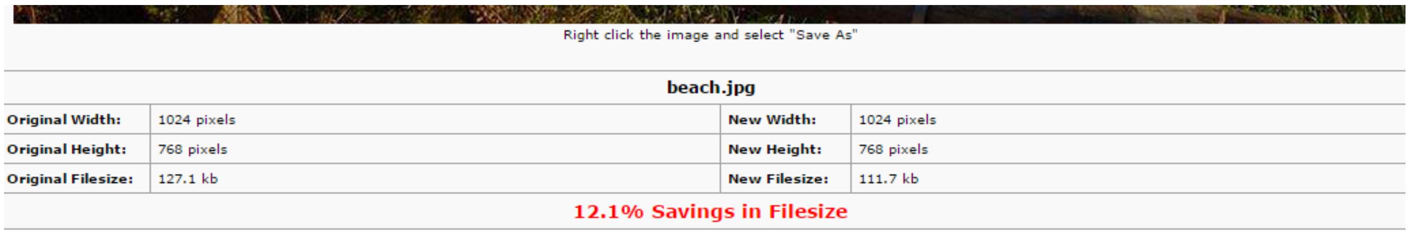
Photo to Optimize: Choose File beach.jpg

[✓] Compress Image 65 Compression Level 0-99

[ ] Resize Photo 400 New Photo Width in Pixels

Optimize Photo * - Large photos may take a few moments. Please be patient.

The result of this process is shown below as another screen shows up with the image optimized and a table showing the result and how much you save in bytes. To download the image, you would simply right-click on the result image and save it to your computer as an image.


Right click the image and select "Save As"

beach.jpg

Original Width:	1024 pixels	New Width:	1024 pixels
Original Height:	768 pixels	New Height:	768 pixels
Original Filesize:	127.1 kb	New Filesize:	111.7 kb

12.1% Savings in Filesize

New image format - WebP?!?

We have just talked about losing quality when optimizing jpg (jpeg) images!

Well, what about the "new kid in the block" **WebP**?

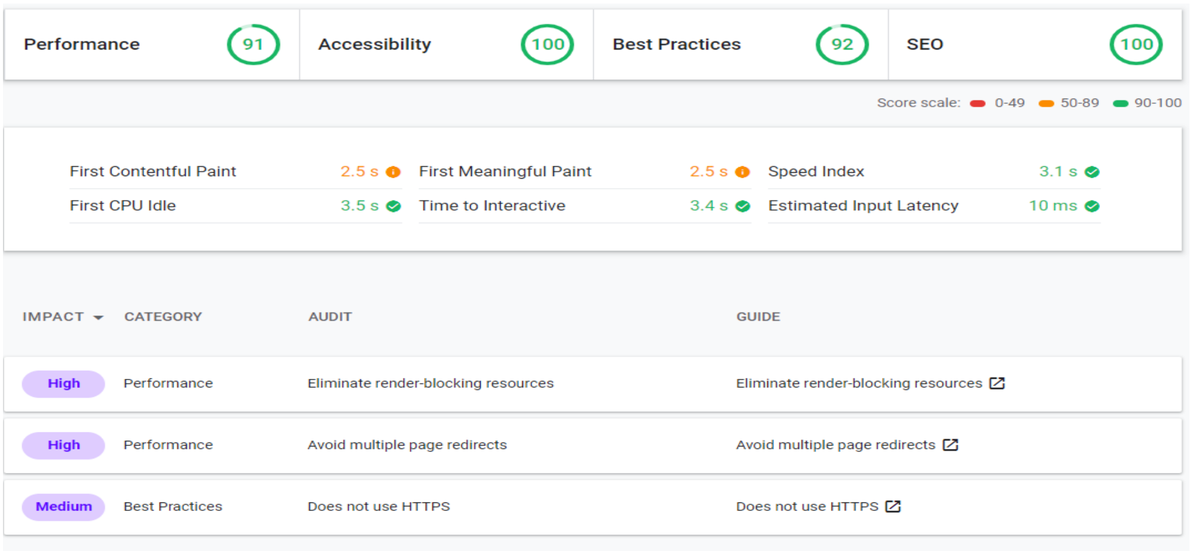
WebP is a new image format that brings lossless compression for images. They are generally 26% smaller in size when compared to png format and around 30% smaller in size when compared to jpeg images. Another detail is that WebP supports transparency. They are being used by web developers to create smaller but richer images that will be downloaded faster.

You can know more about WebP at the Google Developers website. If you need or want to experiment converting images to WebP format you can use some free online converters such as Image Online Converter

Note: What about JPE format? This is simply another notation for JPG (jpeg) format!

Notice that you can also use https://developers.google.com/speed/pagespeed/insights/ in order to analyze a web page and have some tips, from Google, of what can be done to increase performance of that page.

You can also use the website Web Dev (Web Dev – was in Beta mode in Dec 2018) to test your site and some suggestions will come up for you – this site basically will review the performance of your site and you will get some detailed guidance on how to improve it – here below you see the image of how the information might be shown to you:



Performance	91	Accessibility	100	Best Practices	92	SEO	100

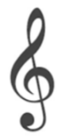
Score scale: 🟥 0-49 🟧 50-89 🟩 90-100

First Contentful Paint	2.5 s ●	First Meaningful Paint	2.5 s ●	Speed Index	3.1 s ✓
First CPU Idle	3.5 s ✓	Time to Interactive	3.4 s ✓	Estimated Input Latency	10 ms ✓

IMPACT	CATEGORY	AUDIT	GUIDE
High	Performance	Eliminate render-blocking resources	Eliminate render-blocking resources
High	Performance	Avoid multiple page redirects	Avoid multiple page redirects
Medium	Best Practices	Does not use HTTPS	Does not use HTTPS

Why using images when there are free Unicodes available?

Sometimes you might be using images that would not be necessary if you know that there is a Unicode that can be used instead. If you check Unicode Table and use the drop-down list (where you see Control Character written), you will notice that there are many different subgroups that you can choose from – for example, instead of downloading a gif or png file for an arrow, you can take a look at the Arrows group and you will see lots of different arrows to be used in your code. So, imagine, you are building a music website and you need to insert this symbol on the right, what would be the Unicode to be used? Once you find that symbol in the Unicode website, if you click once, you will see that the HTML code shows up to you and, for this case here, it would be &#119070;



NOTE: You will need to have the <meta charset="utf-8"> in your HTML code!!! See the importance of this meta tag???

Thinking about Performance from the perspective of saving energy!

Currently the average website on desktop is about 4 times as large as it was back in 2010 (according to HTTP Archive). In mobile devices the data transfer is more expensive in terms of energy that is used. We need, as web developers, to be more responsible as more we add to our pages and keep asking ourselves: is this really necessary?

Consider looking at Website Carbon to check if your site is saving or using too much energy. You can also check your site against Green Web Foundation to see if your web hosting is considered "green".

Make sure that you wrote the most efficient CSS possible to save in bytes and that you are only using JavaScript where extremely necessary. Check out the fonts you are using – are third-party fonts really necessary in your site? What about those third-party frameworks? Remember that every piece adds to a site and we should keep it pleasant but with the minimum load possible as content continues to be "the king"!

A note to be added: energy costs of data transfer depend on the type of network that is used. The range seems to be from around 0.08kWh per GB (Gigabyte) for broadband connections up to 37kWh per GB for 2G networks. A better estimate can be 0.50 kWh per GB (used to be 2.9 kWh per GB) for 3G networks according to study from Joshua Aslan, Kieren Mayers, Jonathan G. Koomey, and Chris France and that’s why you should think always mobile first when designing and developing a site!

Thinking about Performance from the perspective of CSS being used

There are ways that you can implement better performance (especially regarding download time) to your website when you are writing your CSS. This article about optimizing CSS for faster pages, gives an interesting approach and some “food” to think about

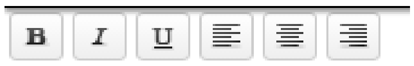
PART 2

HTTP Requests

Reduce HTTP Requests – Each time a request is made to the server, the page load time increases. There are some small details that can make the difference in that matter and one easy way is by **combining like files** – The easiest way to reduce the number of HTTP requests is to combine similar files, for example, you can combine all the CSS files into one and all the JS files into one.

As a "rule of thumb" the CSS should be loaded at the beginning of the document while JS at the end (just before the closing body tag). JS can generally only render one file at a time which prohibits anything else from loading and that's why it should come at the end of the page. Sometimes, tough, you might need to have the JS file/code in the head section of the HTML document, especially if it’s being used to help render the page (example: when using jQuery Mobile, or Bootstrap, or even when the JS being used will help determine the content or layout of the page).

Another possibility to reduce HTTP requests in regard to images is by using **Image Sprites** – using one background image across multiple elements (cuts down HTTP requests). To create a sprite, take a handful of background images (the most commonly used ones) and arrange them into one single image. Then, using CSS add the sprite as a background image to an element and use the background-position to display the correct background image. An example of using sprite can be done on a menu of a web page: take a look at **ex3.html** and now imagine that the**sprite.png** is a single image with all the little images one beside the other and each image is 16px wide and 16px high. The images would be presented like you see here:



The use of **Image Data URI** can also help reduce HTTP request. The image data URI works great for small images. There are some problems to consider: they can be difficult to change and maintain and they do not work in older browsers (IE 7 and below). There are few tools that can help generating data URIs such as [EZGif converter]and [pattern generators] - just make sure that the actual data URI will be less weight than the actual image.

An example using data URI can be seen here below:

```
<img height="100" width="660" alt="Rigged
Pattern"src="data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAAAoAAAAICAYAAADA+m62AAAAPUlEQV
QYV2NkQAO6m73+X/bdxogujiIAU4RNMVwhuiQ6H6wQl3XI4oy4FMHcCJPHcDS6J2A2EqUQpJhohQAyIyYy0nBAGgA
AAABJRU5ErkJggg==">
```

One other tip regarding HTTP requests is to **cache common files** - specific files may be cached when a page loads for the first time and then in future visits, the browser can render directly from the cache. To set the expiration headers for caching files, you will need the **.htaccess** file – in the HTML5 Boilerplate, there is a section marked for Expires Headers.

Default = one month for images, videos, web fonts, common media types; one year for CSS, JS – in the case of the CSS, it might be better to change this to be 1 week for example

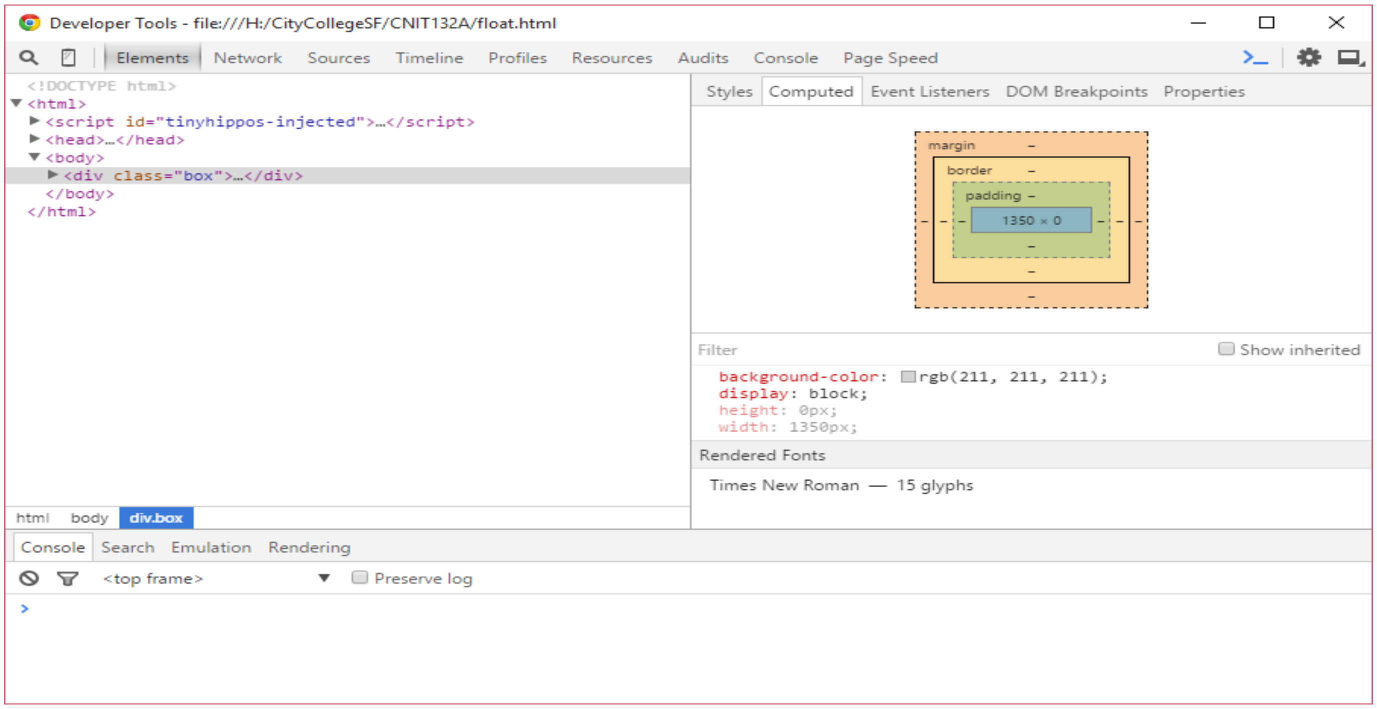
Positioning - Part 1

When developers apply positioning in any web page, the CSS property **float** is the natural one to use. When you float an element, it becomes dependent on the other elements around it.

Remember that the DOM represents all of the different elements and their relationship to each other.

Floats are great but they also bring problems – the most known one is when a parent element contains numerous floated elements – the floated elements do not impact the outer edges of the parent container and, because of that, the parent element loses context of exactly what it contains and collapses (giving the parent element a height of 0 and ignoring other properties). Sometimes this will go unnoticed especially when the parent does not have any styles tied to it and the nested elements align correctly.

Let us see how it happens. Open **float.html** and then inspect the div element (parent container). Now, add a CSS property to this parent element of height and you will notice that the gray color will show up at the top of the 3 boxes. This means that, in reality, the parent with the 3 floated containers has a **height = 0**. You can even choose the tab Computed on the right side and you will see the height = 0 defined there – see image below:



```
<!DOCTYPE html>
▼<html>
  ►<script id="tinyhippos-injected">…</script>
  ►<head>…</head>
  ▼<body>
    ►<div class="box">…</div>
   </body>
</html>
```

```
background-color: rgb(211, 211, 211);
display: block;
height: 0px;
width: 1350px;
```

Times New Roman — 15 glyphs

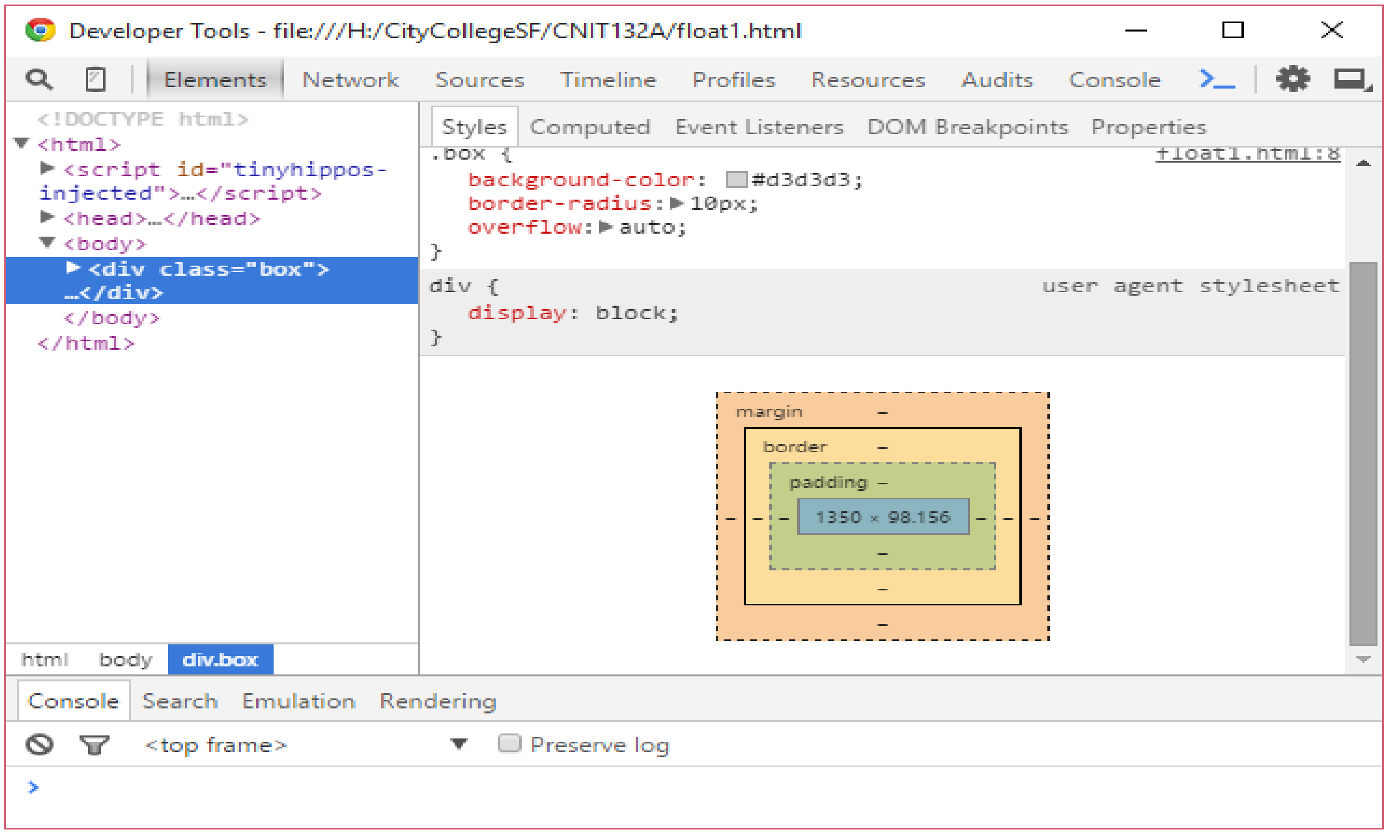
One way you could force these floats to be really contained would be by placing an empty element right before the parent element's closing tag with the style of ***clear: both;***

It's a valid strategy, many times used, but not exactly semantic because depending on the number of floats that will need to be cleared, the number of empty elements can begin to stack up really fast not providing any real content to the page.

Positioning - Part 2

One technique that you can use for the problem of the parent div with height = 0 is the **overflow** and **clearfix**.

Overflow Technique: You can use the CSS overflow property for the parent element with ***overflow:auto;*** which will give an actual height to that parent. In old IE versions, for this to work, you would also need a height or width on the parent element (generally a width of 100%). But in Apple computers that might be using IE, the **auto** value to the overflow property will add scrollbars to the parent element – then, it is better to use the **hidden** value. Now, when you open **float1.html**, you will notice that the parent element has a height defined (see inside the box on the right side) – see image below:


Developer Tools - file:///H:/CityCollegeSF/CNIT132A/float1.html

```
<!DOCTYPE html>
▼<html>
  ▶<script id="tinyhippos-injected">…</script>
  ▶<head>…</head>
  ▼<body>
    ▶<div class="box">
    …</div>
    </body>
</html>
```

```
.box {                                    float1.html:8
   background-color: #d3d3d3;
   border-radius:▶10px;
   overflow:▶auto;
}
div {                           user agent stylesheet
   display: block;
}
```

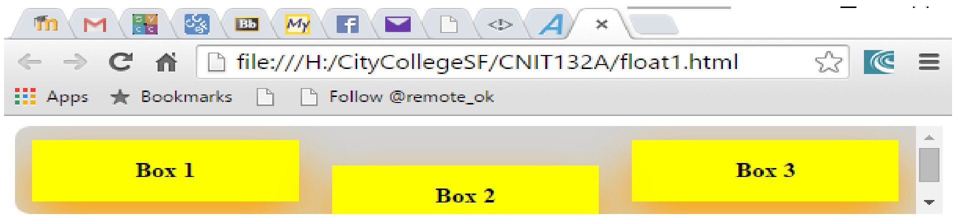
Well, although you might think that everything has been resolved, let's add the following CSS property to the **class inner** in **float1.html**:

```
box-shadow: 0 15px 20px orange;
```

Also, add the following selector and CSS property to **float1.html**:

```
.inner:nth-child(2) { position:relative; top:20px;}
```

Opening in Chrome browser, you should have something as the image below:



There are drawbacks in this technique – for example if you want to add styles to the children that will span outside the parent such as **box shadow**. Different browsers treat the overflow property differently – notice the scrollbar added to Chrome in the image above otherwise the Box 2 would have been cropped.

Positioning - Part 3

Clearfix Technique: It’s a little bit more complex than the previous technique we saw in the Part 2 but with better support from browsers. It’s based on the **:before** and **:after pseudo-elements**so we can create hidden elements above and below the parent. The **:before** prevents the top margin of the children from collapsing by creating an anonymous table-cell element using the**display:table** – helping with consistency even in older versions of IE. The **:after** prevents the bottom margin from collapsing and clear the nested floats. By adding the **zoom** property (a non-standard method, implemented by Microsoft, to zoom elements) to the parent, it will trigger the **hasLayout** mechanism especially for IE 6 and 7.

Open **float2.html** and notice the changes made when comparing with the **float.html** and **float1.html**.

Which technique should I use then? It will depend on the content and your personal preference but make sure to document well especially if working with a group.

One common practice is to set up a class and then use that class to any parent element that will need to contain floats. Something as:

```
.floats:before, :floats:after { content:””; display:table;}
.floats:after { clear:both;}
.floats {zoom: 1;}
```

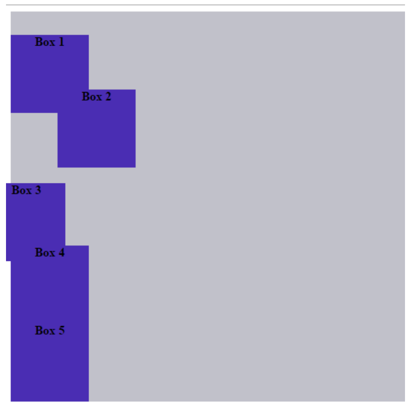
The Position CSS Property

This property accepts 5 different values and, by default, elements receive the value of **static** which means they will not accept any specific box offset properties. So, in a natural flow, box elements will be presented one below the other.

The **relative** value accepts box offset properties such as: ***top***, ***right***, ***bottom***, ***left***. It shifts the element from its default position in any direction. The box offset properties only work on elements with position value of **relative**, **absolute** or **fixed**.

The **absolute** or **fixed** positioning specify the distance between the element and the edges of its parent element.

Although you can set box offset properties for position relative, the elements still remain in the normal flow of the page – this means that when an element is positioned relatively, the surrounding elements will observe the default position of the elements relatively positioned. Open **relative.html** and you should see something as showing in this picture here.



If you would code top and bottom at the same time for one element that is relatively positioned, <u>the top will take priority</u>. In the case of having left and right coded for a single element positioned relatively, <u>the one that will be prioritized depends on the direction of the language the page</u> (English page, the left will take priority, the Arabic page, the right will take priority).

Position absolute: elements are removed from the natural flow and are positioned directly in relation to their parent that is relatively or absolutely positioned. In case there is no such parent, then the element is positioned in relation to the body element.

If you do not specify any box offset property for an element that you use **position:absolute**, this element will be positioned in the top left of its closest relatively positioned parent. If you set only one box offset property – for example, top, the element be positioned vertically but will leave the horizontal positioning to the default value of flush left.

Take a look at **absolute.html** file. Look at the code and open it in Chrome browser. Remove the ***position:relative;*** from the class **box** and see what happens with the box (they are now position absolutely in relation to the body).

When an element has a fixed height and width and position with absolute value, the top property will take priority over bottom in case both are coded. If both left and right are coded for a fixed width, the priority will be given to the direction of which the language of the page is written.

If an element does not have height or width but has the value of absolute for position and uses a combination of top and bottom, it will display an element with height spanning the entire specified size. The same happens when using both left and right – the element will be displayed with a full width based on both the left and right box offset properties. If you use all 4 box offset properties, the element will display with a full specified height and width.

Position Fixed: Works like position absolute but the position being relative to the browser viewport and not scrolling with the page – this means that elements with that type of position will always be present no matter where a user stands on a page.

If you use multiple box offset properties with fixed position you will produce the same behaviors of absolutely positioned elements.

Open **fixed.html** and notice how the boxes are positioned.

Sometimes the fixed positioning is used to build fixed header or footer "tied" to a certain side of a page. As the user scrolls the page, the fixed element stays always within the viewport.

Open **fixedfooter.html** and observe how both left and right box offset are declared. The footer will then span the whole width of the bottom of the page without disrupting the box model so you can apply margins, borders, and padding as much as you want.

The **z-index** property is used to help stack elements on top of each other and will determine the "order" those elements will be shown. The element with the highest z-index value will appear on the top regardless of its placement in the DOM. You can only apply the z-index to elements to which you apply position **relative**, **absolute**, or **fixed**.

There is an interesting exercise somewhat related to positioning of elements but using the % (percentage) unit. Check this great explanation by Wattenberger with examples you can work with!

Complex Selectors

What are the selectors we have? They are: **type** (the element the way it's declared in the HTML), **class**, **ID**

Child Selectors – you can select elements that fall within one another and then they end up having a parent-child relationship.

Descendant Selector – it's the most common child selector – it will match every element that follows an identified parent (ancestor). The descendant does not need to come directly after the ancestor (it can be a "grand kid"). You create a descendant selector by spacing apart the elements. For example: ***article h2 { color: red; }*** – it will only selects h2 elements that are inside an article element

Open **selectorsh2.html** and notice that both h2 will be shown with red color although the second h2 is not a direct descendant of the article element. But the first h2 is not shown in red as it's not a descendant (direct or indirect) of an article element.

Direct Child Selector – Now, if you substitute the space by >, it will be different – **only the DIRECT h2 descendant will be red**.

Sibling Selectors – these can be made by General Sibling Selector or Adjacent Sibling Selector

General Sibling Selector – it allows elements to be selected based on their sibling elements (that share the same common parent) – using the **~** (tilde) between two elements within a selector

If you substitute the **>** by **~** in the **selectorsh2.html**, you will have a general sibling selector that will look for a h2 element that follow and share the same parent of the article element. So, for a h2 to be selected, it must come after the article element. Notice that only the h2 after the article element and the last h2 will be shown in the color red as they both come after an article.

Adjacent Sibling Selector – when you need to select a sibling that comes directly after the other. Just substitute the **~** by a **+** sign in the **selectorsh2.html** and see what happens – only the h2 right after the article element is selected and shown in red color.

Take a look at **siblingsh2.html** and you will see a lot of those sibling selectors being used.

Attribute Selectors – you include the attribute of the element between square brackets **[ ]**. Take a look at **selattrib.html**

You can also change that attribute selectors into **attribute contains selectors** – you would just need to have the value of the attribute with a wild character (*) – for example: ***a[href*="login"] { …. }*** would match anchor tags such as: <a href="/login.aspx">…</a> or <a href="thislogin.html">…</a>

Take a look at **selcontains.html**

You can change, in **selcontains.html**, the selector to be: ***a[href^="login"] { …. }*** ➔ you will be just changing the asterisks (*) to be **^** and see the difference – in this case, only the first line will be changed according to the CSS properties specified meaning that the attribute href value should BEGIN with "login".

Now, change **selcontains.html** to have the following: ***a[href$="html"] { ……. }*** ➔ this change will now target anchor elements in which the value of the attribute href ENDS by "html"

Pseudo Classes & Pseudo Elements

Pseudo Classes

There are some well-known pseudo classes such as **:hover**, **:visited**, **:link**, **:focus**, **:active**.

User Interface State Pseudo classes – there are some pseudo classes generated around the UI state of the elements such as **:enabled**, **:disabled**, **:checked**, **:indeterminate**. Many browsers generally fade out disabled inputs in a way to inform users that the input is not available to interact with but you can style those by using the **:disabled** pseudo class. The **:enabled** is the default state of any input. The **:checked** and **:indeterminate** work generally around checkbox and radio button. When a checkbox or radio button has neither been selected nor unselected, you get the **indeterminate** state.

Structural & Position Pseudo Classes – some new were brought by CSS3.
:first-child – selects an element if it is the first child within its parent – the **:last-child** will do the same for the last child within its parent. The **:only-child** will select an element if it's the only element within a parent. There might be developers that would target this type of element by writing **:first-child:last-child** but when you use **:only-child**, you hold a lower specificity.

The **:first-of-type** selects the first element of its type within a parent and the **:last-of-type** will select the last element of its type within a parent. The **:only-of-type** will select an element if it's the only of its type within a parent.

Take a look at **pseudoclasses.html**. Now, modify the **first-of-type** to **first-child** and see what happens. Why the first paragraph within the article does not get the background-yellow? (Because it's not the first child, the first child is h2).

Change **last-of-type** to **last-child** – answer the same question but now for the cyan background.

Now change the **only-of-type** to **only-child** and see what happens – why??? Is h3 the ONLY child inside the article??? I don't think so….. now, put a div element around the h3 and leave the **only-child** there…. Now, it will get the color red because it's the ONLY child of the div!!!

Remember that you also have **:nth-child(n)**, **:nth-last-child(n)**, **:nth-of-type(n)**, **:nth-last-of-type(n)**

You just need to pay attention that when you change **:nth-child(n)** to be **:nth-last-child(n)**, you are switching the direction of counting starting from the end of the document tree. So, for example, when you have ***li:nth-last-child(3n+2)*** it will identify every third list item starting from the second to last item in a list moving towards the beginning of the list.

Take a look at **nthchild.html** – open in the browser and check the code. Now, modify the **nth-child** to be **nth-last-child** and see what happens when you do this modification - it starts to select the elements from second to last until the top – so, it's like the last element becomes the top one (element #1 and so on…).

Target pseudo class – used to style elements when the ID attribute value matches that of the URI fragment identifier – the fragment identifier is recognized by the hash character (#). Open**target.html** and click on the link at the top of the page and see how the last section is formatted – check the code.

Negation Pseudo-class - ***:not(x)*** – for example ***p:not(.myclass)*** – using the negation pseudo-class to identify every paragraph element without the class **myclass**. You can also have something as ***:not(section)*** and this will select all the elements that are not section elements. Take a look at **negation.html** and try to recognize the **:not** pseudo-classes that are being applied.

Pseudo Elements

Dynamic elements that do not exist in the document tree – they allow unique parts of the page to be stylized – only one pseudo element may be used within a selector at a certain time.

::first-letter, **::first-line**

::before, **::after** – they are generated content pseudo elements – create new inline level inside the selected element – most of the time these are used together with the **content** property to add some content but sometimes these are used to add UI components to the page without having to clutter the document with not-semantic elements.

Fragment Pseudo-element - ::selection – identifies part of the document that has been selected or highlighted by the action from the user. The background-image property, for this case, is ignored but the background-color, text-shadow, color, background properties are ok.

: or **::** - the fragment pseudo element was added with CSS3 and to differentiate pseudo classes from pseudo elements, the double colons were added to pseudo elements – browsers currently support single and double colons except for selection that needs to be written as **::selection** – if you take a look at the website **caniuse.com**, you will notice that you will need to use the **–moz**prefix such as **::-moz-selection** otherwise this will not work in Firefox. In the **negation.html**, add, in the CSS part the following line:
::selection {background-color:yellow;}

Remember also to add ***::-moz-selection {background-color:yellow;}***

Save the file and open it in a browser. Select any part of the text and see what happens.

Look at **pseudoelements.html** and see how you can draw an arrow with pseudo-elements.

Be careful with certain CSS selectors

In regards of performance, it's always better to use text and not text as image. If you need to write text together with an image (over an image, for example), it's better to choose a web font that will look nice and then have the image as a background-image and write the text using real text, using the font-family property to apply the chosen web font. This is also true if you want to create menu items that would look like buttons – instead of having the buttons set as images (gif or png format), you should use the normal HTML to include the items of your menu (navigation) as text and then use CSS properties to set up the format of the text that you might make the menu items look like buttons (using border-radius, gradient (or background-color), box-shadow, etc.).

You just need to pay attention that some CSS properties add to the loading time of a page – it's a minimum addition but should be considered when we talk about performance (take a look at the image below that shows a simple example of how much some CSS properties add to the loading time):

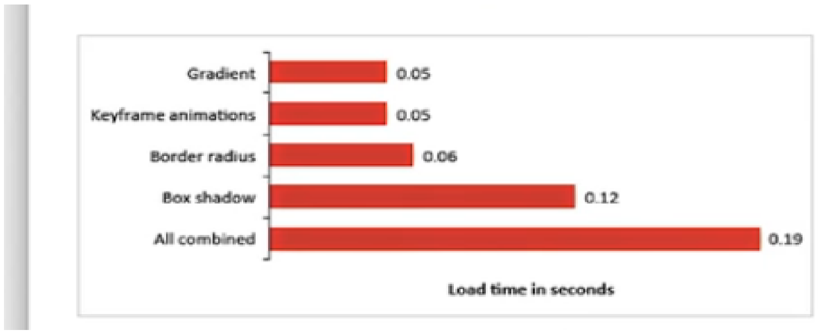


Image copied from Smashing Magazine showing the loading time of some CSS3 properties in iPhone (iOS 5.0)

Resource Hints to optimize performance

What are resource hints? Modern browsers use all types of techniques available trying to guess what the user may like to see next with the main objective of improving performance. The browser, in reality, does not know our application as a whole, the browsers do not know that a certain link, among all other links there might be on the page, will be the one that the user will likely use next. For a browser, all links carry the same weight. What if we, as developers could do something to help the browsers with this “guessing exercise” so the browser would “know” where the user will likely go next and then fetch the next page ahead of time?!? That’s the main principal of **Resource Hints**!

All resource hints use the **rel** attribute of the **<link>** element you generally find in the head part of the HTML document. There are different resource hints that can be used, and we will study some:

DNS prefetching – Every time you type a URL in the address bar of the browser, or click on a link, or load an image from another domain, the browser will do a DNS lookup (transform the human-friendly URL to an IP address) to find the server that holds the requested resource and, depending on the application/page, you might have dozens of DNS lookup that are happening. Although the browser caches those, it’s a slow process and a developer can optimize that by organizing the resources in fewer domains but if this is not possible at all, then you can use the **dns-prefetch** resource hint. It’s well supported by browsers, but even if the browser would not support, no harm will be done as the browser will simply ignore it. For example:

```
<link rel=”dns-prefetch” href=”https://another.domain.com”>
```

You might use the prefetch as:

```
<link rel="prefetch" href="https://mycompany.com/documents/?page=2" as="document">
```

Here you are telling the browser to go ahead and start working on that page 2 and you are basically “guessing” that the user will require that document next – if the user requires, great guess and it will load very quickly but if not, then you wasted resources in that prefetching. By the way, in this example, the as=”document” is an optional attribute just to tell the browser that the resource should be loaded as a document (a web page) – there are other values you can use for this as attribute: audio, video, script, style, font, image, etc.

Pre-connecting – There are certain steps that are followed to establish a connection with a server: DNS Lookup, TCP handshake (a “brief conversation” between the browser and the server to make the connection), TLS negotiation for HTTPS sites (to verify that the certification of information is correct and valid). These steps happen once per server but take some time especially if the server is distant from the browser. We can then pre-connect to a server to make sure that when the browser reaches the part of the page that requests a resource from that server, those steps to establish the connection have been already done. Browser support is great but no harm as the browsers that do not support it will simply ignore it. For example:

```
<link rel="preconnect" href="https://scripts.mycompany.">
```

Pre-rendering – It goes a step further than prefetch and does basically all the necessary work to display the page which might include parsing the resource for any sub-resources such as: Images, JavaScript files, etc. So, the example shown with prefetch above could be changed to:

```
<link rel="prerender" href="https://mycompany.com/documents/?page=2">
```

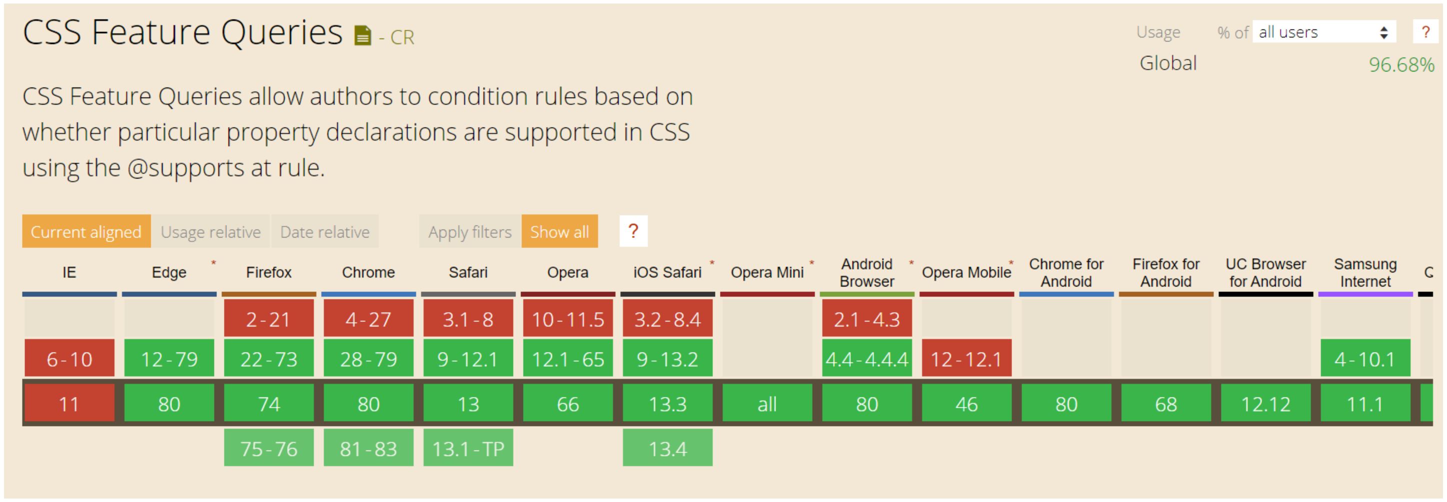
This can really load that page quickly (same type of speed as when the user hits the back button of the browser) but here the gamble is worse because you will not be simply downloading the files, but also executing them (for example, any JavaScript it might have as well). This uses a lot of CPU and memory (battery as well). Besides, you should take a look at caniuse.com and you will see that the browser support for it is very limited!

Note: All of those are NOT directives to browsers, they are suggestions – on a busy device the browser might not respond to hints at all. If the memory is low, the browser might not fetch the next page until the current one has been offloaded. Most likely the hints are followed as suggested by the developer but you need to know that it will be up to the browser!

Feature Queries

What is it? Can I start using it?

Feature Queries is something that has been mentioned in many meetings (MeetUps), seminars, etc. but people were kind of afraid to use for lack of support from browsers. Now, it seems they are here. If you go to caniuse.com and type “feature queries”, you might receive something similar to the image below:



CSS Feature Queries - CR

CSS Feature Queries allow authors to condition rules based on whether particular property declarations are supported in CSS using the @supports at rule.

Usage % of all users ?
Global 96.68%

Current aligned | Usage relative | Date relative | Apply filters | Show all | ?

IE	Edge *	Firefox	Chrome	Safari	Opera	iOS Safari *	Opera Mini *	Android Browser *	Opera Mobile *	Chrome for Android	Firefox for Android	UC Browser for Android	Samsung Internet	C
		2-21	4-27	3.1-8	10-11.5	3.2-8.4		2.1-4.3						
6-10	12-79	22-73	28-79	9-12.1	12.1-65	9-13.2		4.4-4.4.4	12-12.1				4-10.1	
11	80	74	80	13	66	13.3	all	80	46	80	68	12.12	11.1	
		75-76	81-83	13.1-TP		13.4								

Image made from caniuse.com in March/2020 - https://caniuse.com/#search=feature%20queries

As you can see, this feature is now supported in every major browser (do not count Internet Explorer in that, but Edge is there)!

Feature queries use the **@supports** rule that allows you to wrap CSS in a conditional block that will only be applied if the browser (the user agent) supports the CSS property-value pair. For example, you might have your CSS as:

```
@supports (display: flex) {
      .myclass { display: flex; }
}
```

So, the class **myclass** will only have the property-value ***display: flex;*** applied if the browser supports it!

You can even use operators such as AND, NOT to create more complex feature queries such as the one you see below:

```
@supports (display: flexbox)
   and
   (not (display:flex)) {
        .myclass { display: flexbox; }
}
```

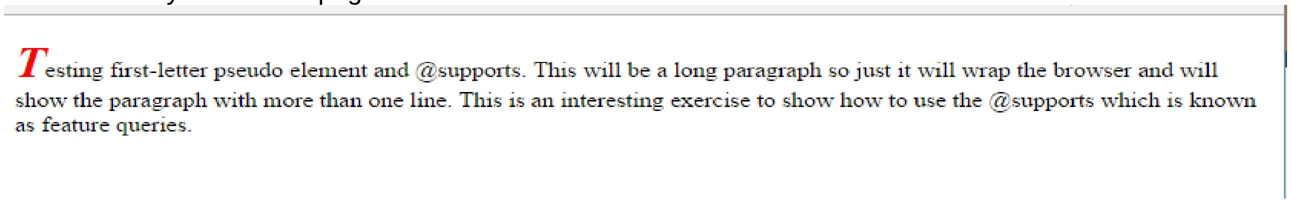
Now, of course, you will not be including the @supports for every CSS property-value pair (especially when using some new CSS3 features), for example, if you have something as:

```
.myclass { border: 2px solid red;
                  border-radius: 5px; }
```

You will not be using the @supports in that case because the browser that does not support border-radius will simply not put it but the element will still have the 2px-red border around it anyway. So, you should use the feature queries when you want to apply a mix of old and new CSS properties but only when the new CSS properties are supported. A good example is with the pseudo-element first-letter – if you write the following CSS:

```
p::first-letter { font-size:2em;
                      color: red;
                      font-weight:bold;
                      font-style:italic; }
```

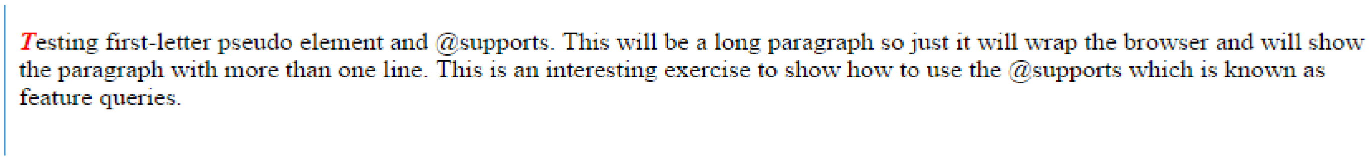
This is the way that a web page with that code would be shown:


Testing first-letter pseudo element and @supports. This will be a long paragraph so just it will wrap the browser and will show the paragraph with more than one line. This is an interesting exercise to show how to use the @supports which is known as feature queries.

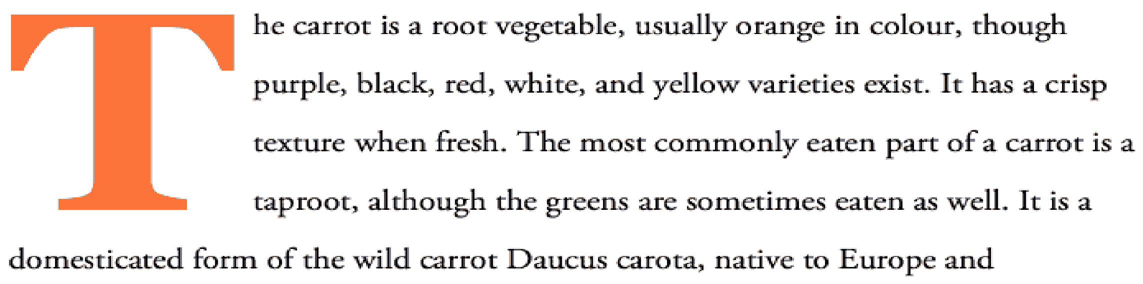
If we use the initial-letter new CSS property such as:

```
p::first-letter { initial-letter: 2;
                     -webkit-initial-letter: 2;
                      color: red;
                      font-weight:bold;
                      font-style:italic; }
```

In browsers that do not support the initial-letter CSS property, the page would show as


Testing first-letter pseudo element and @supports. This will be a long paragraph so just it will wrap the browser and will show the paragraph with more than one line. This is an interesting exercise to show how to use the @supports which is known as feature queries.

Instead of the big letter T taking all the lines of the paragraph – such as in this example here (this was an example seen in Safari version 9):


The carrot is a root vegetable, usually orange in colour, though purple, black, red, white, and yellow varieties exist. It has a crisp texture when fresh. The most commonly eaten part of a carrot is a taproot, although the greens are sometimes eaten as well. It is a domesticated form of the wild carrot Daucus carota, native to Europe and

So, in this case, it would not be ok to have the letter T in red, bold, and italicized unless the initial-letter CSS was also supported by the browser. That's then the case to use @supports and then your code would be:

```
@supports (initial-letter: 2) or (-webkit-initial-letter: 2) {
    p::first-letter { initial-letter: 2;
                     -webkit-initial-letter: 2;
                      color: red;
                      font-weight:bold;
                      font-style:italic; }
}
```

Avoid writing something like:

```
@supports not (display:grid) { … }
```

Although most browsers currently support feature queries, the ones that do not support will not go inside those curly brackets ({ and }) because they do not understand/support feature queries. So, it would be better to write something as:

```
/* writing here the normal CSS for browsers that do not support display: grid */

@supports (display:grid) {
   /* writing here the CSS for browser that support display: grid */
}
```

When writing the CSS code, you will generally end up with some situations to think about (shown in the table below) based on a structure of code as shown above:

Browser supports feature queries	Browser supports new CSS property	You will get the new CSS property displayed
Browser supports feature queries	Browser does not support new CSS property	You will not get the new CSS property displayed, instead the CSS before @supports
Browser does not support feature queries	Browser does not support new CSS property	You will get the CSS before @supports (browser will not even enter the @supports)
Browser does not support feature queries	Browser supports new CSS property	You will get the CSS before @supports – **THERE MIGHT BE A PROBLEM!**

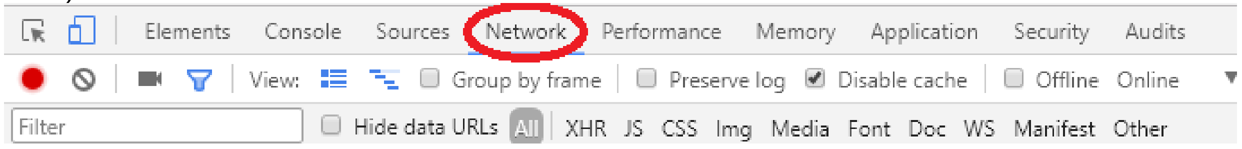
The 4th line of the table above can or cannot happen and you need to decide what will cause the less negative impact to most of your users – for example, if I'm using display:flex, I would probably not use the feature queries because if I look at the caniuse.com, I will see that some browsers support display:flex but they do not support feature queries and, instead, I may use what is called CSS overrides – which means that I would leave the element with display:flex and if a browser does not support it, it will simply not be applied.

Using or not feature queries might really depend on each situation you will have in your hands.

Using Chrome Developer Tools (CDT) to Analyze Network Performance

It is important to know how to analyze the **Network** tab in the CDT as you might receive reports, especially from mobile users, that a particular page on your web application is slow.

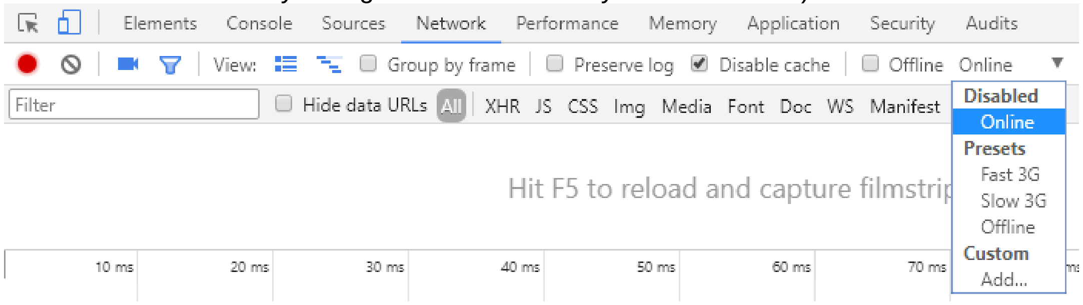
You should then open that page in your Chrome browser and then open the CDT. When the CDT is opened, make sure that you click on the **Network** tab (see that tab shown in the image here after I clicked on the Network marked in a red circle):



In that tab, you should enable the **Capture screenshot** – it's the camera icon that you see in the image above and you see here as well - 📹 - this will become blue when you click on it to show that it's been activated.

Most of the time the issue will happen when users try to access your application from a mobile device which means that testing network performance on a desktop/laptop might be deceiving as your internet connection will be faster than a mobile user's besides the fact that most probably your browser caches resources from previous visits. You should then check the **Disable Cache** checkbox that you can also see in the previous image – when you check this box; the CDT will not serve any resources from the cache which will make your test more accurate for what first-time users might be experiencing when they view your page.

From the drop-down menu (shown in the image below) that by default might show **Online**, you should select **Slow 3G** (or the slowest connection you might have at the time you use the CDT):

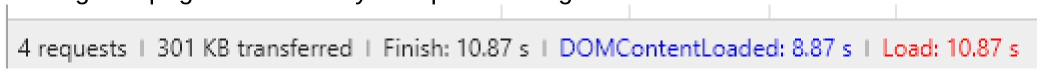


Notice that the **Disable cache** is checked and the video-camera icon is now blue (has been activated before).

Now, you just need to reload the page to analyze what will come in this **Network** tab.

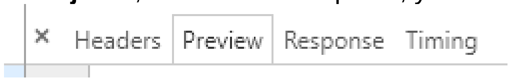
1) When the browser sees a `<script>` tag, it will load and execute the script immediately – you should find scripts that are not necessary during the page load and mark them asynchronous or defer their execution to speed up load time

For example, in the image below, you see the bottom part of the CDT (called Summary pane), in the Network panel, after loading one page that had only 4 requests being done:



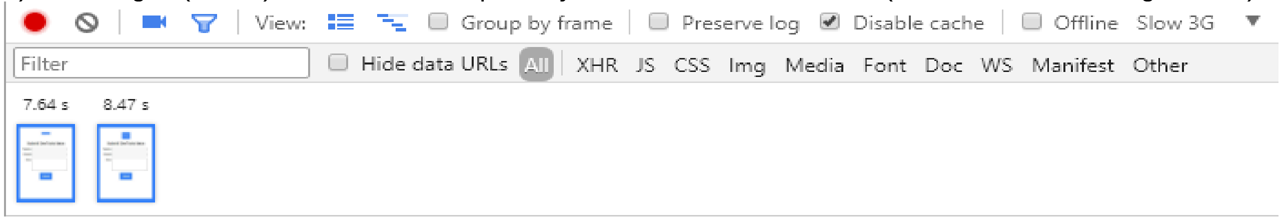
```
4 requests | 301 KB transferred | Finish: 10.87 s | DOMContentLoaded: 8.87 s | Load: 10.87 s
```

Note the **DOMContentLoaded** of more than 8 seconds! If you see something like that, even with more than 4-5 seconds, look at the scripts you have as they might be delaying the load of the page. You can certainly click on the **.js** file that you see in the list of files loaded in the CDT and you might even see more details about each of these files – when you click on a **.js** file, in the **Network** panel, you will see a set of tabs as shown below:

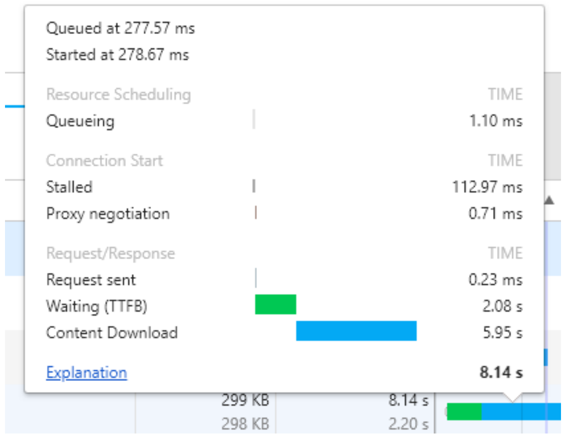


You can click on the **Preview** to see the source code of the **.js** file you clicked on. If you see that this .js file is not important to be loaded before the content of the page, then you can certainly move that script to the bottom of the page (before the closing </body> tag) and also put the async attribute in the <script> tag such as: <script src="app.js" async> - If the app.js is not necessary to be executed immediately, then you should definitely use the async attribute because, by default, all JavaScript is parser blocking – the browser does not know what the script is planning to do so it assumes the worst case scenario and blocks the parser until that external file (app.js) is downloaded and "analyzed". The async attribute signs to the browser that the script does not need to be executed at the exact point where it's being referenced which allows the browser to continue building the DOM and let the script execute when it is ready (in the example of the script tag presented here, after the app.js is fetched from the cache or from a server).

2) The images (media) – in the Network panel, you see some screenshots (like shown in the image below):



If you double-click on those screenshots, you will see how the DOM looks like at a certain moment in time and you might see that some images might be taking too long to download. If you hover over the Waterfall (see that column with some horizontal colored bars), on the bar that represent the image you are downloading for your page, you might get a pop-up screen like the one shown here on the right side – notice the time it took for the Content Download.



Queued at 277.57 ms
Started at 278.67 ms

Resource Scheduling	TIME
Queueing	1.10 ms
Connection Start	**TIME**
Stalled	112.97 ms
Proxy negotiation	0.71 ms
Request/Response	**TIME**
Request sent	0.23 ms
Waiting (TTFB)	2.08 s
Content Download	5.95 s
Explanation	8.14 s

299 KB 8.14 s
298 KB 2.20 s

This will help confirm to you that the image might be too big and you might get a chance to optimize it – IT'S NOT TO REDUCE THE WIDTH AND HEIGHT IN THE HTML OR CSS CODE!!!

3) Avoid using @import in CSS files and avoid also grouping all @media queries in the same CSS file. The browser will download all CSS files anyway but higher priority will be given to the file that is related to the resolution you currently need to fulfill.

So, instead of having the @media queries inside a single CSS file, you can have the general CSS in one single file and then the media-specific CSS in separated files and your <link> tags would be something as:

```
<link rel=”stylesheet” href=”all.css” media=”all”>
<link rel=”stylesheet” href=”mobile.css” media=”(min-width: 200px)”>
<link rel=”stylesheet” href=”desktop.css” media=”(min-width:481px)”>
<link rel=”stylesheet” href=”print.css” media=”print”>
```

The @import should be avoided as it's slow by nature – you are simply creating more “round trips” on the critical path.

4) There are other little tips even for the case when you have <script> tags and <link> to CSS file – which is pretty common, right? But as there are many different situations for this case, here is an article that goes over different situations with some nice tips on how you should code in your page.

After doing some changes (to the code or to some files), make sure you test again the page using the CDT Network panel.

The Google Developers website has a very interesting course about Critical Rendering Path where you can learn how to build your pages in a better way based on how the browsers generally render pages.