Responsive Web Design - The Viewport

What is The Viewport?

The viewport is the user's visible area of a web page.

The viewport varies with the device, and will be smaller on a mobile phone than on a computer screen.

Before tablets and mobile phones, web pages were designed only for computer screens, and it was common for web pages to have a static design and a fixed size.

Then, when we started surfing the internet using tablets and mobile phones, fixed size web pages were too large to fit the viewport. To fix this, browsers on those devices scaled down the entire web page to fit the screen.

This was not perfect!! But a quick fix.

Setting The Viewport

HTML5 introduced a method to let web designers take control over the viewport, through the <meta> tag.

You should include the following <meta> viewport element in all your web pages:

<meta name="viewport" content="width=device-width, initial-scale=1.0">

A <meta> viewport element gives the browser instructions on how to control the page's dimensions and scaling.

The width=device-width part sets the width of the page to follow the screen-width of the device (which will vary depending on the device).

The initial-scale=1.0 part sets the initial zoom level when the page is first loaded by the browser.

## Size Content to The Viewport

Users are used to scroll websites vertically on both desktop and mobile devices - but not horizontally!

So, if the user is forced to scroll horizontally, or zoom out, to see the whole web page it results in a poor user experience.

Some additional rules to follow:

**1. Do NOT use large fixed width elements -** For example, if an image is displayed at a width wider than the viewport it can cause the viewport to scroll horizontally. Remember to adjust this content to fit within the width of the viewport.

**2. Do NOT let the content rely on a particular viewport width to render well** - Since screen dimensions and width in CSS pixels vary widely between devices, content should not rely on a particular viewport width to render well.

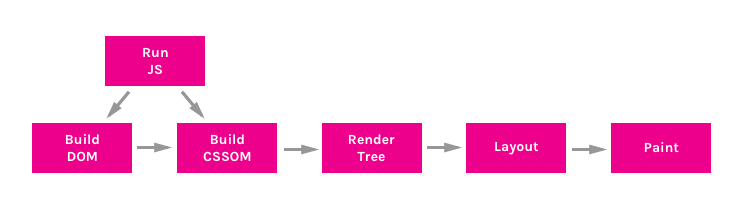
**3. Use CSS media queries to apply different styling for small and large screens** - Setting large absolute CSS widths for page elements will cause the element to be too wide for the viewport on a smaller device. Instead, consider using relative width values, such as width: 100%. Also, be careful of using large absolute positioning values. It may cause the element to fall outside the viewport on small devices.

**Understanding the Critical Rendering Path**

When a browser receives the HTML response for a page from the server, there are a lot of steps to be taken before pixels are drawn on the screen. This sequence the browsers needs to run through for the initial paint of the page is called the "Critical Rendering Path".

Knowledge of the CRP is incredibly useful for understanding how a site's performance can be improved. There are 6 stages to the CRP -

1. Constructing the DOM Tree
2. Constructing the CSSOM Tree
3. Running JavaScript
4. Creating the Render Tree
5. Generating the Layout
6. Painting



## 1. Constructing the DOM Tree

The DOM ([Document Object Model](https://www.w3.org/DOM/)) Tree is an Object representation of the fully parsed HTML page.

A good thing about HTML is that it can be executed in parts. The full document doesn't have to be loaded for content to start appearing on the page. However, other resources, CSS and JavaScript, can block the render of the page.

1. **Constructing the CSSOM Tree**

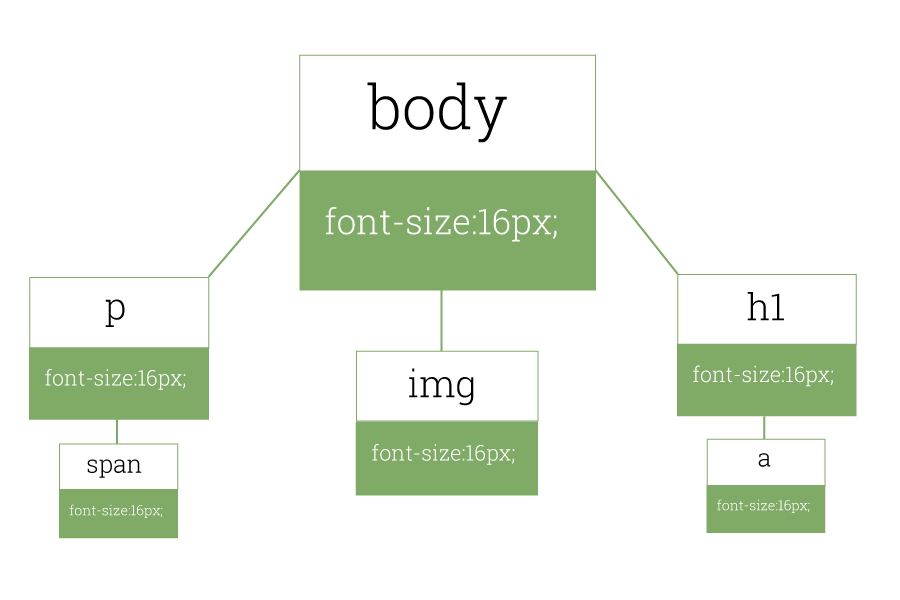
CSS is considered a **"render blocking resource"**. This means that the [Render Tree (see below)](https://bitsofco.de/understanding-the-critical-rendering-path/#4creatingtherendertree) cannot be constructed without first fully parsing the resource. Unlike HTML, CSS cannot be used in parts because of its inherit cascading nature. Styles defined later in the document can override and change styles that were previously defined. So, if we start using CSS styles defined earlier in the stylesheet before the entirety of the stylesheet has been parsed, we may get a situation where the wrong CSS is being applied. This means that CSS must be fully parsed before we can move on to the next stage.

CSS files are only considered render blocking if they apply to the current device. The **<link rel="stylesheet">** tag can accept a **media** attribute, in which we can specify any media query which the styles within apply to. If, for example, we have a stylesheet with a media attribute of **orientation:landscape**, and we are viewing the page in portrait mode, that resource will not be considered render blocking.

CSS can also be **"script blocking"**. This is because JavaScript files must wait until the CSSOM has been constructed before it can run.

**What is CSSOM?**

* CSSOM stands for CSS Object Model.
* It is basically a "map" of the CSS styles found on a web page.
* It is much like the DOM, but for the CSS rather than the HTML.
* The CSSOM combined with the DOM are used by browsers to display web pages.



## What does the CSSOM do?

It maps out the rules in your stylesheet to the things on the page which need styling.

The CSSOM takes many complicated steps to do this, but the end function of the CSSOM is to map out styles to the places those styles need to go.

(More precisely, the CSSOM identifies tokens and coverts them into nodes which are linked into a tree structure. The entire map of all the nodes with their associated styles of a page would be the CSS Object Model).

You don't have to understand how the CSSOM actually works in order to optimize your web pages.

There are only a few key points that a webmaster needs to know about the CSSOM to take advantage of the very real page speed optimizations that can be made.

1. Know that the CSSOM blocks anything from rendering.
2. Know that the CSSOM has to be built everytime you load a new page.
3. Know that there is a relationship between the CSS your page loads and the javascript your page loads - The way javascript is used on your web pages may (and often do) block the CSSOM from being built.

To lay it all out in plain english... The CSSOM is required to display anything. Nothing is displayed until it is done. If you block the process of the CSSOM being built, then the CSSOM takes longer, and that means it takes longer for anything to display. If your javascript is blocking the CSSOM from being built, your users are looking at blank pages longer than they should.

1. **Running JavaScript**

JavaScript is considered a **"parser blocking resource"**. This means that the parsing of the HTML document itself is blocked by JavaScript.

When the parser reaches a **<script>** tag, whether that be internal or external, it stops to fetch (if it is external) and run it. This why, if we have a JavaScript file that references elements within the document, it must be placed after the appearance of that document.

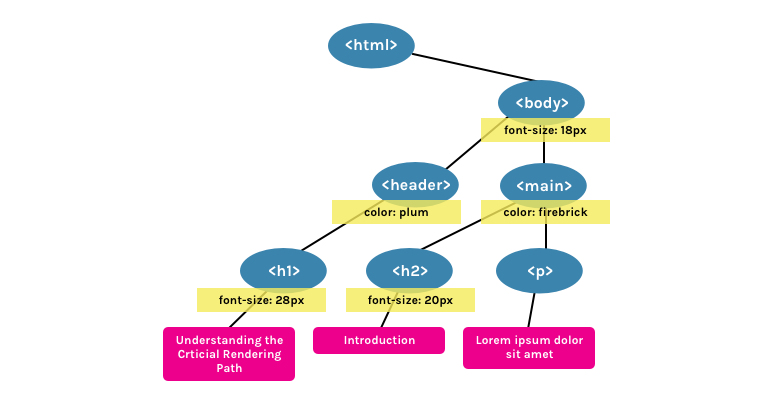
To avoid JavaScript being parser blocking, it can be loaded asynchronously be applying the **async** attribute.

<script async src="script.js">

## 4. Creating the Render Tree

The Render Tree is a combination of both the DOM and CSSOM. It is a Tree that represents what will be eventually rendered on the page. This means that it only captures the visible content and will not include, for example, elements that have been hidden with CSS using **display: none**.

Using the example DOM and CSSOM above, the following Render Tree will be created -



## 5. Generating the Layout

The Layout is what determines what the size of the viewport is, which provides context for CSS styles that are dependent on it, e.g. percentage or viewport units. The viewport size is determined by the meta viewport tag provided in the document head or, if no tag is provided, the default viewport width of 980px is used.

## 6. Painting

Finally, in the Painting step, the visible content of the page can be converted to pixels to be displayed on the screen.

How much time the paint step takes depends on the size of the DOM, as well as what styles are applied. Some styles require more work to execute than others. For example, a complicated gradient background-image will require more time than a simple solid background colour.

## Putting it All Together

To see the Critical Rendering Path in process, we can inspect it in DevTools. In Chrome, it is under the **Timeline** tab

If we look at the Event Log for the page load, this is what we get -



1. **Send Request** - GET request sent for index.html
2. **Parse HTML** and **Send Request** - Begin parsing of HTML and DOM construction. Send GET request for style.css and main.js
3. **Parse Stylesheet** - CSSOM created for style.css
4. **Evaluate Script** - Evaluate main.js
5. **Layout** - Generate Layout based on meta viewport tag in HTML
6. **Paint** - Paint pixels on document

## Web browsers use the CSSOM to render a page

1. The web browser examines your HTML and builds the DOM (Document Object Model).
2. The web browser examines your CSS and builds the CSSOM (CSS Object Model).
3. The web browser combines the DOM and the CSSOM to create a render tree.
4. The web browser displays your webpage.

## Optimizing the critical rendering path

Just to be clear, let's define a few things:

* critical - absolutely needed
* render - display or show (in our case our webpages are "rendered" when they can be seen by a user)
* path - the chain of events that lead to our webpage being displayed in a browser
* initial view - also known as "above the fold", the initial view is the part of a webpage visible to a user before they scroll

## The critical



I have painted a very bleak picture thus far, but the good news is that you can call a million things for your webpage and it can have 12000 pictures, 200 javascript files and the page can still load in a second or so.

## How is this accomplished?

By understanding what is critical for your webpage to display the above the fold / initial view content.

## Optimizing the render path

There are really just three things to concentrate on [1](https://developers.google.com/web/fundamentals/performance/critical-rendering-path/optimizing-critical-rendering-path) ...

* Minimize the number of critical resources.
* Minimize the number of critical bytes.
* Minimize the critical path length.

## Understanding pagespeed measurement

When Google talks about pagespeed, they are not talking about the overall time it takes to download a webpage. What they care about is how quickly does a user start seeing content on that page (initial view) and how quickly can the user start interacting with that content.

The reason Google has started using pagespeed as a ranking factor [2](https://googlewebmastercentral.blogspot.com/2010/04/using-site-speed-in-web-search-ranking.html) is based upon the satisfaction of their users. It is not a good experience for someone searching Google for something when they are sent to a page that takes forever to load.

People complain to Google about this, they say "Why are you sending me to a page that loads soooo slow?". This is known as perceived speed.

If a user is looking at a blank white webpage for 10 seconds waiting for it to load, that is bad and Google doesn't want to show that page in their results. If that same webpage displayed information in the first second, it would then be a great experience, and Google would want it in their results.

Our main concern when we talk about webpage speed is to get content to the user as soon as possible in the initial view.

# How to defer images?

**What are deferred images?**

* A deferred image is downloaded after the initial page load
* Images not initially visible on a page (below the fold) can be deferred allowing what is visible to load faster

## Deferring images for faster pages

This article will describe a simple way to defer images without jQuery or lazy loading.

All these images are competing for the same bandwidth as your page resources such as css and javascript. This means images are getting in the way of providing the initially visible part of your webpage to users as quickly as possible.

## This is a well known issue

The main way developers and webmasters have solved this issue is through a method called lazy loading.

Lazing loading images is a solution where as a user scrolls down the page, images are loaded as needed. There are many wonderful things about lazy loading and I use it often, but it has some issues...

* Lazy loading can cause performance issues.
* Lazyloading does not provide a viable solution for some webpages.
* Lazy loading is not ideal for [mobile](https://varvy.com/mobile/) performance.

**Deferring images without lazy loading or jQuery**

The truth is that lazy loading images is just a more complicated way of deferring images.

To get back to basics we will be talking about deferring images without lazy loading. But let's first define the stuff that lazy loading is actually doing.

1. Observing a scroll position
2. Monitoring scroll position
3. Reacting to a scroll position
4. deferring images

In the four things above, only one of them is the deferring of images.

**Let's discuss the deferring images part**

When a webpage is rendered in a browser the browser will attempt to download all the images it can find on the page. If there are two images on the page, it will download two images. If there are one hundred images on the page it will download all one hundred.

This is default browser behavior. It has to request and download all images.

The only reliable way around this (for all browsers) is to trick the browser into thinking those images aren't there.

The way that is done in lazy loading and elsewhere is to provide a tiny default image in our html, and then switch that default image via javascript to the real image we want to display.

This means that images are marked up something like...

<img src="fake.png" data-src="real-image.png"

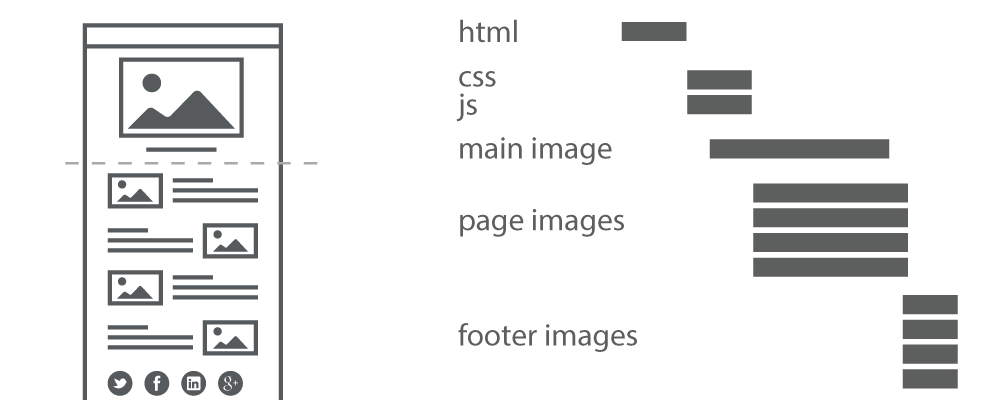
When the page is initially loaded, the browser will get the "fake image" once and then that will be the only image the browser sees, so whether you have one image or a hundred, it won't matter because the browser has already downloaded the fake image.

Then via javascript we swap out the fake image with the real one.

When the browser sees that there is a new image in the html, it will now download it.

## Understanding the page load

To understand how, when and what method of deferring to use, we need to understand how a page loads. This is good knowledge to have and will only take a minute or two to read.



The image above shows a small webpage loading. The page has a main image, several other images, a css file and a javascript file.

In this rather typical page, every single resource is competing to be downloaded at the same time.

In this rather typical page, every single resource is competing to be downloaded at the same time.

The truth is though, the only thing that needs to be downloaded is the stuff above the dotted line. This is the "above the fold" or the initially visible part of the page.

This means that for the user to see the above the fold content, we only need the html, css, javascript, and the main image.

Let's look how we can make this page load twice (or more) faster than it is now. We should...

* Only load the main image
* Defer the rest of the images

Once we do that, we see that the pageload event can occur at a much earlier part of the load.

Now we have just cut a page that was loading 12 items before the page load into a page that only loads four things before the pageload event.

## Doesn't lazy loading already do this for me?

It does, but there are many situations where lazy loading does not work ideally.

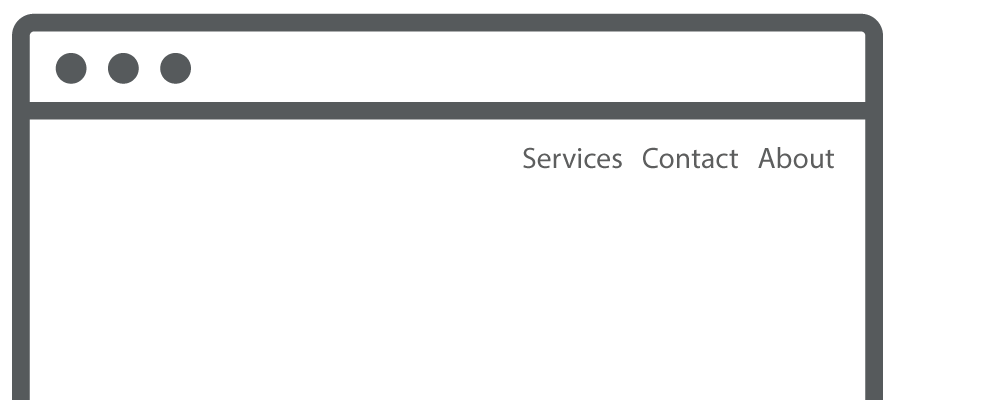
The most common reason people may chose not to defer images via lazy loading is likely the new popular trend of having one page templates or themes.

## One page templates

If you have a one page template (this means that your main navigation is not taking you to other pages it is rather taking you to different parts of the same page) and you choose lazy loading, an interesting problem pops up.

Your page loads, the user sees your main navigation, and if they click, they are taken to part of your page that does not have the images loaded yet.

Ewww. I don't like that.



In this case even though your users just used your navigation, they are taken to a place where they will have to wait for an image.

## This is where we defer the images without lazy loading

In the scenario of a one page template, there is no reason to do all the things that lazy loading does (observe, monitor and react to a scroll postion).

Why not just defer those images and have them load immediately after the page has loaded?

## How to do it

To do this we need to markup our images and add a small and extremely simple javascript. I will show the method I actually use for this site and others. It uses a base 64 image, but do not let that scare you.

The html

<img src="data:image/png;base64,R0lGODlhAQABAAD/ACwAAAAAAQABAAACADs=" data-src="your-image-here">

The javascript

<script>  
function init() {  
var imgDefer = document.getElementsByTagName('img');  
for (var i=0; i<imgDefer.length; i++) {  
if(imgDefer[i].getAttribute('data-src')) {  
imgDefer[i].setAttribute('src',imgDefer[i].getAttribute('data-src'));  
} } }  
window.onload = init;  
</script>

# Defer loading javascript

## How to defer loading of javascript?

Truly deferring javascript means loading or parsing of that javascript only begins after page content has loaded (Meaning it will not affect pagespeed or the critical rendering path).

1. Using the "onload" event we call an external javascript
2. The external javascript will not load before page content is loaded
3. External javascript will then run and affect page

## Script to call external javascript file

This code should be placed in your HTML just before the </body> tag (near the bottom of your HTML file). I highlighted the name of the external JS file.

<script type="text/javascript">  
function downloadJSAtOnload() {  
var element = document.createElement("script");  
element.src = "defer.js";  
document.body.appendChild(element);  
}

if (window.addEventListener)  
window.addEventListener("load", downloadJSAtOnload, false);  
else if (window.attachEvent)  
window.attachEvent("onload", downloadJSAtOnload);  
else window.onload = downloadJSAtOnload;  
</script>

Note- attachEvent is supported by IE8 or lower versions. addEventListener are supported by IE9.

## Why does it matter?

It matters because Google is judging page speed as a ranking factor and because users want fast loading pages. It also matters a great deal for your [mobile seo](https://varvy.com/mobile/). Google measures your [page speed](https://varvy.com/pagespeed/) from the the time it is called to when the **page is initially loaded**. This means you want to get to the page load event as quickly as possible. That initial page load time is what Google is using to judge your webpage (and let's not forget your users waiting for it to load). Google actively promotes and recommends [prioritizing above the fold content](https://varvy.com/pagespeed/prioritize-visible-content.html) so getting any resources at all (js, css, images, etc.) out of the [critical rendering path](https://varvy.com/pagespeed/critical-render-path.html) is well worth the effort. If it makes your users happy, and it makes Google happy, you should probably do it.

# **What is Lazy Loading?**

**Lazy loading**(also called on-demand loading) is an optimization technique for the online content, be it a website or a web app.  
Instead of loading the entire web page and rendering it to the user in one go as in bulk loading, the concept of lazy loading assists in loading only the required section and delays the remaining, until it is needed by the user.



For example, say a user requests for the logo of GeeksForGeeks from a search engine. The entire web page, populated with the requested content, is loaded. Now if the user opens the first image and is satisfied with it, he will probably close the web page thus, rest of the images so loaded will be left unseen. This will result in the wastage of the resources so consumed in the bulk load of that page. Thus the solution to this is Lazy Loading.  
  
One form of lazy loading is **infinity scroll**, in which, the content of the web page is loaded as and when the user scrolls down the page. It is a popular technique being used by various websites.

**Advantages of Lazy loading:**

* On-demand loading reduces time consumption and memory usage thereby optimizing content delivery. As only a fraction of the web page, which is required, is loaded first thus, the time taken is less and the loading of rest of the section is delayed which saves storage. All of this enhances the user’s experience as the requested content is fed in no time.
* Unnecessary code execution is avoided.
* Optimal usage of time and space resources makes it a cost-effective approach from the point of view of business persons. (website owners)

**Disadvantages of Lazy loading:**

* Firstly, the extra lines of code, to be added to the existing ones, to implement lazy load makes the code a bit complicated.
* Secondly, lazy loading may affect the website’s ranking on search engines sometimes, due to improper indexing of the unloaded content.

**Conclusion:** Though there are certain pitfalls of lazy loading yet the big advantages, as optimal utilization of the two major resources (time & space) and many more make us overlook its disadvantages.

**What, exactly, is the DOM?**

The DOM is an object-based representation of the source HTML document. It is essentially an API to the page, allowing programs to read and manipulate the page’s content, structure, and styles. It has some differences, as we will see below, but it is essentially an attempt to convert the structure and content of the HTML document into an object model that can be used by various programs.

## How is a web page built?

How a browser goes from a source HTML document to displaying a styled and interactive page in the viewport is called the “Critical Rendering Path”. Although this process can be broken down into several steps, as I cover in my article on [Understanding the Critical Rendering Path](https://bitsofco.de/understanding-the-critical-rendering-path/), these steps can be roughly grouped into two stages. The first stage involves the browser parsing the document to determine what will ultimately be rendered on the page, and the second stage involves the browser performing the render.

The result of the first stage is what is called a “render tree”. The render tree is a representation of the HTML elements that will be rendered on the page and their related styles. In order to build this tree, the browser needs two things:

1. The CSSOM, a representation of the styles associated with elements
2. The DOM, a representation of the elements

## What the DOM is not

It seems like the DOM is a 1-to-1 mapping of the source HTML document or what you see your DevTools. However, as I mentioned, there are differences. In order to fully understand what the DOM is, we need to look at what it is not.

### The DOM is not your source HTML

Although the DOM is created from the source HTML document, it is not always exactly the same. There are two instances in which the DOM can be different from the source HTML.

#### 1. When the HTML is not valid

The DOM is an interface for **valid** HTML documents. During the process of creating the DOM, the browser may correct some invalidities in the HTML code.

Let’s take this HTML document for example:

<!doctype html>

<html>

Hello, world!

</html>

The document is missing a **<head>** and **<body>** element, which is a requirement for valid HTML. If we look at the resulting DOM tree, we will see that this has been corrected:

html

head

body

Hello, world!

#### 2. When the DOM is modified by Javascript

Besides being an interface to viewing the content of an HTML document, the DOM can also be modified, making it a living resource.

We can, for example, create additional nodes to the DOM using Javascript.

var newParagraph = document.createElement("p");

var paragraphContent = document.createTextNode("I'm new!");

newParagraph.appendChild(paragraphContent);

document.body.appendChild(newParagraph);

This will update the DOM, but of course not our HTML document.

### The DOM is not what you see in the browser (i.e., the render tree)

What you see in the browser viewport is the render tree which, as I mentioned, is a combination of the DOM and the CSSOM. What really separates the DOM from the render tree, is that the latter only consists of what will eventually be painted on the screen.

Because the render tree is only concerned with what is rendered, it excludes elements that are visually hidden. For example, elements that have **display: none** styles associated to them.

<!doctype html>

<html lang="en">

<head></head>

<body>

<h1>Hello, world!</h1>

<p style="display: none;">How are you?</p>

</body>

</html>

The DOM will include the **<p>** element:

html

head

body

h1

Hello, world!

p

How are you?

However, the render tree, and therefore what is seen in the viewport, will not include that element.

html

body

h1

Hello, world!

### The DOM is not what is in DevTools

This difference is a bit more minuscule because the DevTools element inspector provides the closest approximation to the DOM that we have in the browser. However, the DevTools inspector includes additional information that isn’t in the DOM.

The best example of this is CSS pseudo-elements. Pseudo-elements created using the **::before** and **::after** selectors form part of the CSSOM and render tree, but are not technically part of the DOM. This is because the DOM is built from the source HTML document alone, not including the styles applied to the element.

Despite the fact that pseudo-elements are not part of the DOM, they are in our devtools element inspector.

This is why pseudo-elements cannot be targetted by Javascript, because they are not part of the DOM.

# What is the Shadow DOM?

## A DOM within a DOM

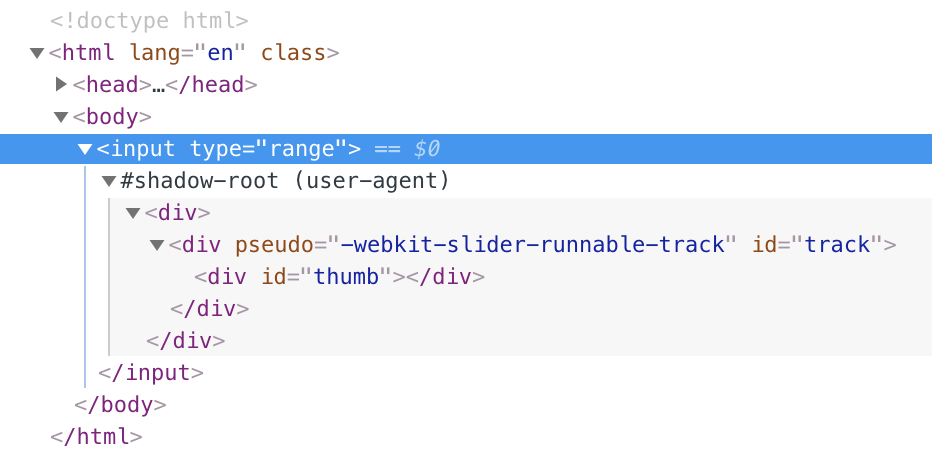
You can think of the shadow DOM as a “DOM within a DOM”. It is its own isolated DOM tree with its own elements and styles, completely isolated from the original DOM.

Although only recently specified for use by web authors, the shadow DOM has been used by user agents for years to create and style complex components such as form elements. Let’s take the range input element, for example. To create one on the page, all we have to do is add the following element:

<input type="range">

That one element results in the following component:

If we dig deeper, we will see that this one **<input>** element is actually made up of several smaller **<div>** elements, controlling the track and the slider itself.



This is achieved using the shadow DOM. The element that is exposed to the host HTML document the simple **<input>**, but underneath it there are elements and styles related to the component that do not form part of the DOM’s global scope.

## How the shadow DOM works

First, we start with the **shadow host**. This is the regular HTML element within the original DOM that we want to attach the new shadow DOM to. For a component like the Follow button, it could also contain the fallback element that we would want displayed if Javascript was not enabled on the page or shadow DOM wasn't supported.

<span class="shadow-host">

<a href="<https://twitter.com/ireaderinokun>">

Follow @ireaderinokun

</a>

</span>

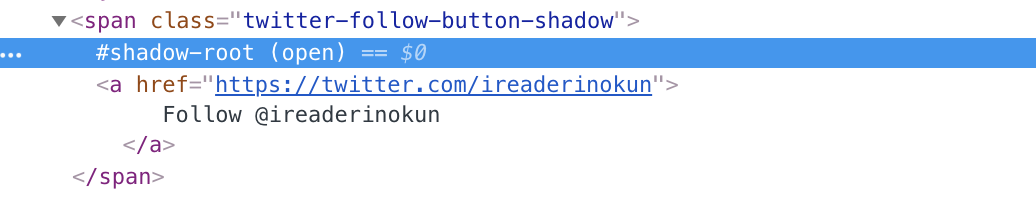
Note that we didn’t just use the **<a>** element as the shadow host, because certain elements, primarily interactive elements, can’t be shadow hosts.

To attach a shadow DOM to our host, we use the **attachShadow()** method.

const shadowEl = document.querySelector(".shadow-host");

const shadow = shadowEl.attachShadow({mode: 'open'});

This will create an empty **shadow root** as a child of our shadow host. The shadow root is the start of a new shadow DOM in the way that the **<html>** element is the start of the original DOM. We can see our shadow root in the devtools inspector by the **#shadow-root**.



Although the regular HTML children are viewable in the inspector, they are no longer visible on the page as the shadow root takes over.

Next, we want to create the content to form our new **shadow tree**. The shadow tree is like a DOM tree, but for a shadow DOM instead of a regular DOM. To create our follow button, all we need is a new **<a>** element, which will be almost exactly the same as the fallback link we already have, but with an icon.

<script>

const shadowEl = document.querySelector(".shadow-host");

const shadow = shadowEl.attachShadow({mode: 'open'});

shadow.appendChild(document.querySelector('a'));

</script>

Finally, we can add some styles by creating a **<style>** element and appending that to the shadow root too.

const styles = document.createElement("style");

styles.textContent = `

a, span {

vertical-align: top;

display: inline-block;

box-sizing: border-box;

}`

shadow.appendChild(styles);

## The DOM vs the shadow DOM

In some ways, the shadow DOM is a "lite" version of the DOM. Like the DOM, it is a representation of HTML elements, used to determine what to render on the page and enables the modification of the elements. But unlike the DOM, the shadow DOM is not based on a full, standalone document. A shadow DOM, as it's name suggests, is always attached to an element within a regular DOM. Without the DOM, a shadow DOM doesn't exist.