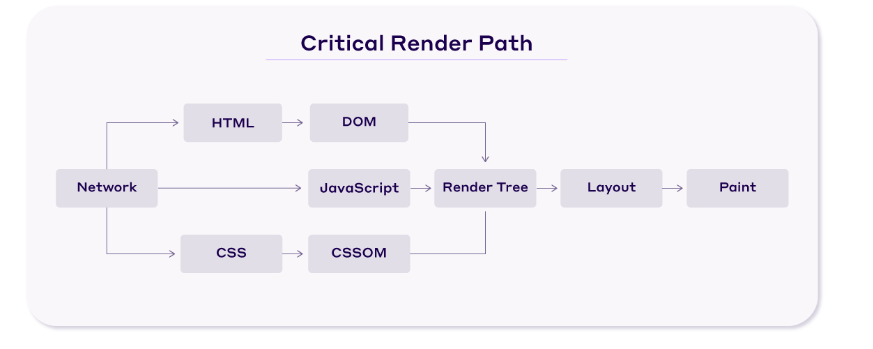
**Critical Rendering Path (CRP)**

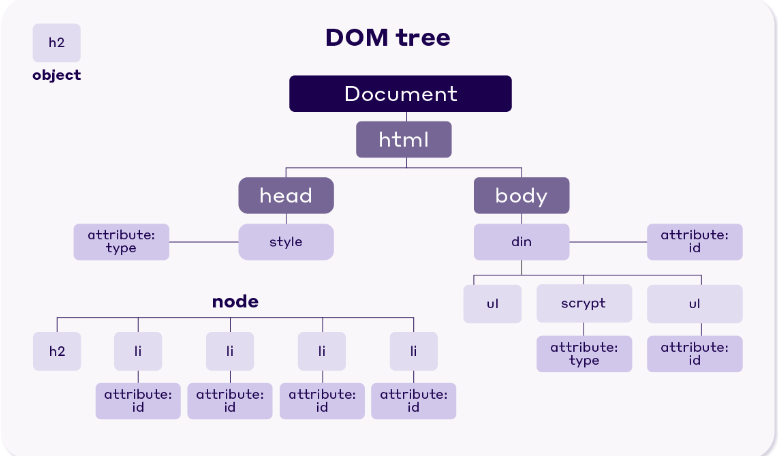
The Critical Rendering Path refers to the sequence of steps that a web browser takes to convert HTML, CSS, and JavaScript code into a visual representation on a user's screen. 

Here’s a quick overview of the steps performed by the browser when rendering a page:

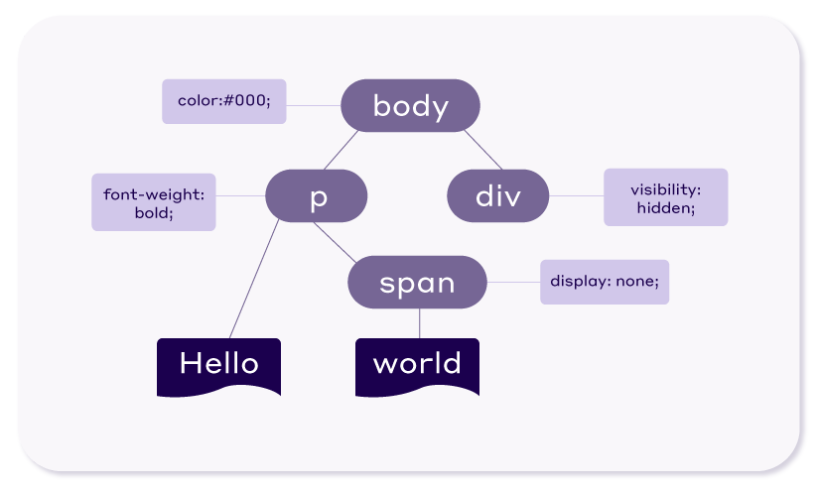
1. The browser downloads and parses the HTML markup and creates the DOM.
2. Next, it downloads and processes the CSS markup and constructs the CSS Object Model (CSSOM).
3. Then, it combines the necessary nodes from the DOM and CSSOM to create the Render Tree, a tree structure of all visible nodes required to render the page.
4. It calculates the dimensions and position of every element on the page through the Layout process.
5. Finally, the browser paints the pixels on the screen.

1. DOM

When a web page is loaded, the browser parses the HTML and creates a tree-like structure of nodes that represent the elements in the document.



2. CSSOM



When a website is loaded, the browser has to process the CSS to apply the styles. Unlike HTML, which can be processed bit by bit, CSS needs to be processed all at once. This is because some styles might be overwritten by others later in the CSS file, so the browser needs to wait until it has read the whole CSS file before deciding which styles to apply.

The browser blocks the rendering process until it receives and parses all the CSS. That’s why CSS is considered a render-blocking resource.

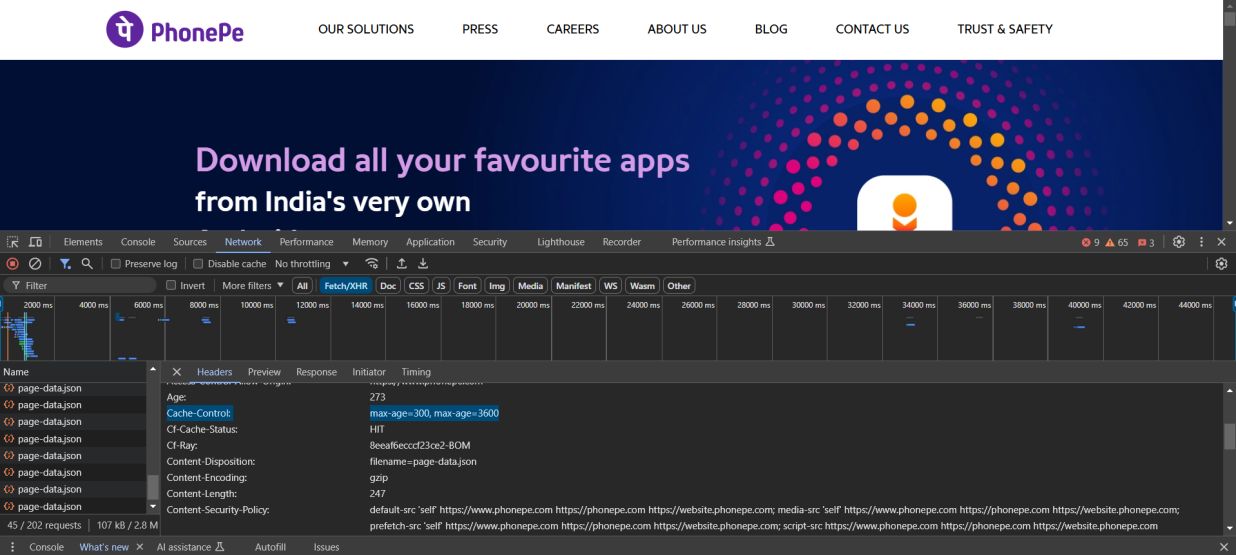
**How to improve Critical Rendering Path?**

1. **Use CDN**

In CDN latency is reduced so performance Improves

**Cache-Control**

We can control the cache basically send a response header from server by **Cache-Control send a parameters in it max-age=3600 means to cache this file till 3600ms(60mins)**



**max-age**

The max-age=N response directive indicates that the response remains [fresh](https://developer.mozilla.org/en-US/docs/Web/HTTP/Caching" \l "fresh_and_stale_based_on_age) until *N* seconds after the response is generated.

Cache-Control: max-age=604800

**Cache-Control: Private**

** What It Does:**

* **Specifies that the resource is intended for a single user (e.g., a specific browser).**
* **The resource cannot be cached by shared caches like a proxy server or CDN.**
* **It is still cached by the browser.**

** Use Case:**

* **When the resource contains sensitive or user-specific data that should not be shared.**
* **Examples:**
  + **Personalized account details.**
  + **Private dashboards or user-specific reports.**

Cache-Control: private, max-age=3600

**Cache-Control: Public**

 **What It Does**:

* Indicates the resource **can** be cached by both browsers and shared caches like proxies and CDNs.
* Often used for static resources that are not user-specific.

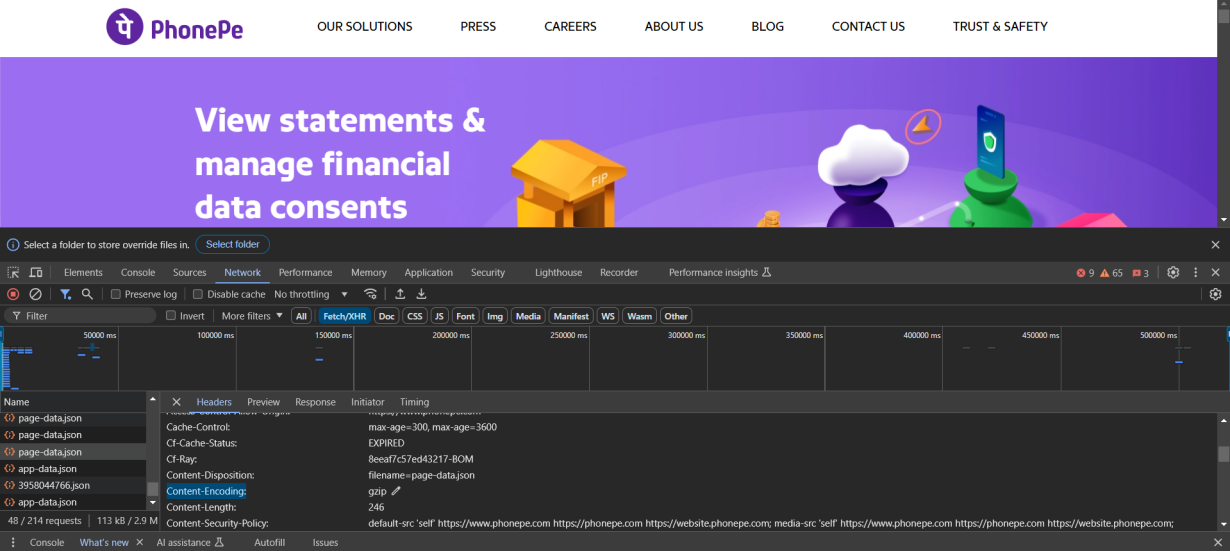
 **Use Case**:

* When the resource is general and safe to be shared across users.
* Examples:
  + Stylesheets (style.css).
  + JavaScript libraries (app.js).
  + Images, fonts, and other static assets.

Cache-Control: public, max-age=86400

**content-encoding**

gzip will basically make your js file an encoded file so it will reduce the size of your file by at least 30-50% without removing any code



**ETag**

An **ETag (Entity Tag)** is an HTTP response header used to identify a specific version of a resource. It helps with web caching and conditional requests, ensuring that only updated resources are transferred between a client and server, thus saving bandwidth and improving performance.

**Minification**

Minification is the process of removing all unnecessary code or unUsed code (like whitespaces, comments, and redundant formatting) without changing its functionality. This reduces the size of files such as HTML, CSS, and JavaScript, resulting in faster download times and improved website performance.

<https://www.toptal.com/developers/javascript-minifier>

**name mangling**

**Name mangling** in JavaScript refers to the process of transforming variable, function, or property names in your code into shorter, less descriptive names during a build process. This is done primarily to reduce file size and improve loading speed, but it can also provide a layer of obfuscation.

**Original Code**:

function calculateSum(a, b) {

const total = a + b;

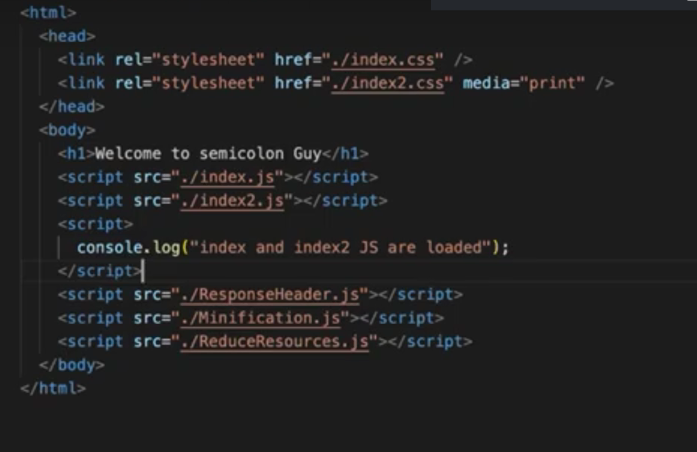
return total;

}

**After Name Mangling**:

function a(b,c){const d=b+c;return d;}

JavaScript has parser blocking means so first index.js will load then index2.js will load



async and defer

The async and defer attributes in the <script> tag are used to control how JavaScript files are loaded and executed in an HTML document. Both attributes improve page performance by preventing scripts from blocking the parsing of HTML, but they behave differently in terms of when scripts are executed.

**Details of async**

1. **Loading Behavior**:
   * The script is downloaded asynchronously, meaning it does not block HTML parsing.
   * The browser continues parsing the HTML while the script is being fetched.
2. **Execution Timing**:
   * The script is executed **immediately** after it is downloaded, even if HTML parsing is not complete.
3. **Use Case**:
   * Best for scripts that **do not depend on other scripts** or the HTML structure (e.g., analytics scripts).
4. **Example**:

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

**Details of defer**

1. **Loading Behavior**:
   * The script is downloaded asynchronously, like async, without blocking HTML parsing.
2. **Execution Timing**:
   * The script is executed **after the HTML parsing is complete**, in the order they appear in the document.
3. **Use Case**:
   * Best for scripts that **depend on the DOM being fully parsed** or need to be executed in sequence.

<script src="script.js" defer></script>

**Which Should You Use?**

* **Use async**:
  + For scripts that are independent and do not rely on other scripts or the DOM.
  + Examples: Analytics, advertisements, and tracking scripts.
* **Use defer**:
  + For scripts that depend on the DOM being fully parsed or must run in a specific order.
  + Examples: Application logic or scripts interacting with HTML content.

**Resource Hints**

Resource Hints can improve page performance by giving the browser extra information that it can't infer from the document HTML. For example, you can tell the browser to preload certain network resources that the page will need to render later on.

There are three main types of resource hints:

* **preload** – load content that's required for the intial render
* **prefetch** - load content that may be needed to render the next page
* **preconnect** - establish a [server connection](https://www.debugbear.com/blog/http-server-connections) without loading a specific resource yet

## What are preload hints?[​](https://www.debugbear.com/blog/resource-hints-rel-preload-prefetch-preconnect" \l "what-are-preload-hints" \o "Direct link to What are preload hints?)

When browsers load a page they don't just sequentially download resources as they need them to render the page. Instead they look at the entire HTML document, decide what resources they'll need to render the page, and start loading those resources. So while a style tag in the document head is [render-blocking](https://www.debugbear.com/blog/render-blocking-resources), it does not block subsequent network requests from being made.

The preload resource hint lets you tell the browser about additional URLs that it should preload. That's useful if the initial page HTML does not already contain the URL. Chrome also has something called a "preload scanner", which can detect preload hints even before Chrome starts parsing the document.

If you have a CSS file that sets a background image you can make sure that the CSS file and the background image are loaded at the same time, rather than sequentially. If a Fetch request is made by JavaScript you can start loading the resource before the JS code is executed.

Here's an example of a rel preload link tag:

<link rel="preload" href="/public/home.js" as="script" />

Preload requests for JavaScript files are made by the browser with [high priority](https://www.debugbear.com/blog/request-priorities), so you should only use them for resources that are needed for the initial rendering of the page. If you don't use a preloaded resource within 3 seconds Chrome will complain:

**Fetchpriority[​](https://www.debugbear.com/blog/resource-hints-rel-preload-prefetch-preconnect" \l "fetchpriority" \o "Direct link to Fetchpriority)**

You can use [Priority Hints](https://www.debugbear.com/blog/fetchpriority-attribute) in combination with preload hints. For example, preloaded images will be loaded with low priority by default, but you can change the priority to high like this:

<link rel="preload" href="/image.png" as="image" fetchpriority="high" />

**The crossorigin attribute[​](https://www.debugbear.com/blog/resource-hints-rel-preload-prefetch-preconnect" \l "the-crossorigin-attribute" \o "Direct link to the-crossorigin-attribute)**

If you preload cross-origin resources you need to add the crossorigin attribute to the link tag. Fonts always need the attribute:

<link rel="preload" href="fonts/Roboto-regular.woff2" as="font" crossorigin />

**What are prefetch hints?[​](https://www.debugbear.com/blog/resource-hints-rel-preload-prefetch-preconnect" \l "what-are-prefetch-hints" \o "Direct link to What are prefetch hints?)**

Prefetched resources might be needed when the user navigates to the next page, or after the user starts interacting with the page. So loading them before the user starts the navigation will make the page load faster for them.

For example, at DebugBear we prefetch the JavaScript application code on the homepage. When a user logs in or views the demo they don't need to wait as long.

<link rel="prefetch" href="/public/app.08343a72.js" as="script" />

Prefetched resources are loaded with a much lower priority than preloaded ones, so that the user's experience of the current page isn't negatively impacted.

**What are preconnect hints?[​](https://www.debugbear.com/blog/resource-hints-rel-preload-prefetch-preconnect" \l "what-are-preconnect-hints" \o "Direct link to What are preconnect hints?)**

Preconnecting allows you to set up an [HTTP connection](https://www.debugbear.com/blog/http-server-connections) to a server, even if you don't know the exact request URL yet. That way, when a request is made, the existing connection can be used immediately.

For example, you may not know the what requests a third-party script will make, or it may be hard for your backend to predict what resources the front-end app will need.

Preconnect resource hints speed up [load times](https://www.debugbear.com/docs/page-load-time) by reducing the number of network round-trips needed when the browser actually loads a resource from the server.

You can see in the screenshot below that making the initial connection to a server requires several round-trips. The browser needs to resolve the domain name and establish connections over TCP and SSL. Once that's all done fetching a particular resource from a server only requires a single round-trip.

So most of those round-trips can be made with a preconnect resource hint, before you know the exact resources you need to fetch.

<link rel="preconnect" href="https://storage.googleapis.com" />

**Reflow**

Old (Directly changing in real DOM)

****

New

Document Fragment (will append in real dom only one time)



**Image Lazy Loding**

Lazy loading is a strategy to identify resources as non-blocking (non-critical) and load these only when needed. It's a way to shorten the length of the [critical rendering path](https://developer.mozilla.org/en-US/docs/Web/Performance/Critical_rendering_path), which translates into reduced page load times.

<img loading="lazy" src="image.jpg" alt="..." />

<iframe loading="lazy" src="video-player.html" title="..."></iframe>