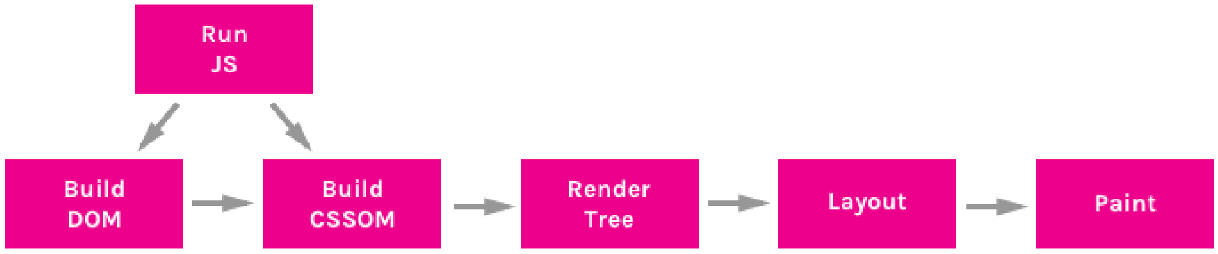
Understanding the Critical Rendering Path

Jan 17, 2017 *performance*

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



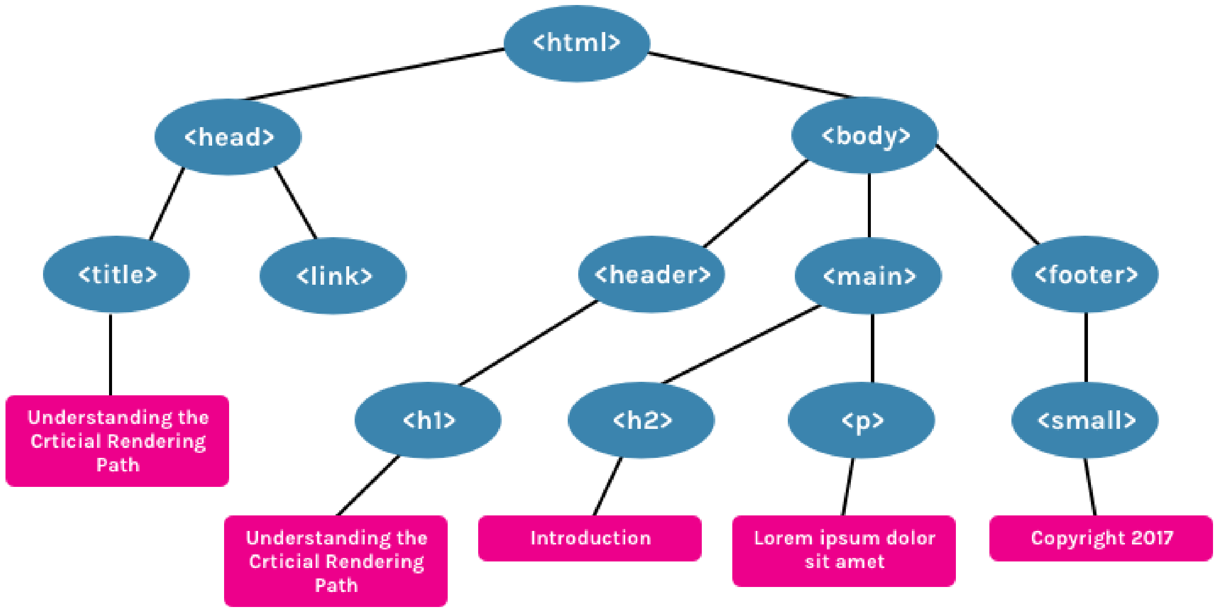
1. Constructing the DOM Tree

The DOM (Document Object Model) Tree is an Object representation of the fully parsed HTML page. Starting with the root element, `<html>`, nodes are created for each element/text on the page. Elements nested within other elements are represented as child nodes and each node contains the full attributes for that element. For example, an **`<a>`** element will have the **`href`** attribute associated with it's node.

Take, for example, this sample document -

```
<html>
<head>
  <title>Understanding the Critical Rendering Path</title>
  <link rel="stylesheet" href="style.css">
</head>
<body>
  <header>
      <h1>Understanding the Critical Rendering Path</h1>
  </header>
  <main>
      <h2>Introduction</h2>
      <p>Lorem ipsum dolor sit amet</p>
  </main>
  <footer>
      <small>Copyright 2017</small>
  </footer>
</body>
</html>
```

This will create the following DOM Tree -


<html>
<head>
<body>
<title>
<link>
<header>
<main>
<footer>
Understanding the Crticial Rendering Path
<h1>
<h2>
<p>
<small>
Understanding the Crticial Rendering Path
Introduction
Lorem ipsum dolor sit amet
Copyright 2017

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.

2. Constructing the CSSOM Tree

The CSSOM (CSS Object Model) is an Object representation of the styles associated with the DOM. It is represented in a similar way to the DOM, but with the associated styles for each node, whether they explicitly declared or implicitly inherited, included.

In the **`style.css`** file from the document mentioned above, we have the folowing styles -

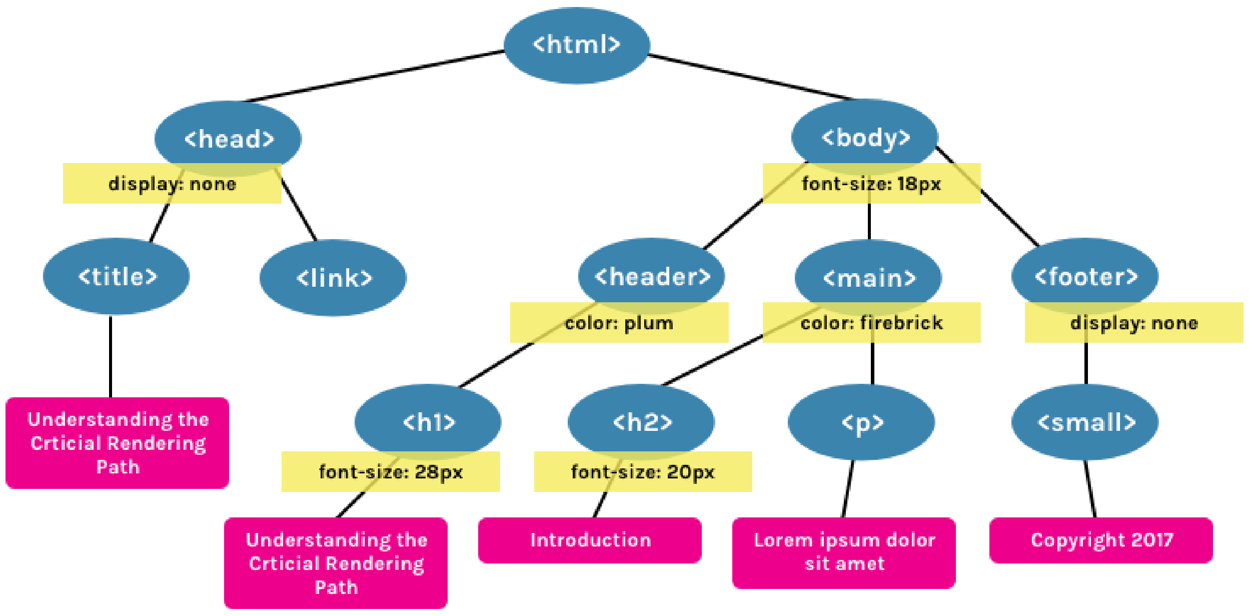
```
body { font-size: 18px; }

header { color: plum; }
h1 { font-size: 28px; }

main { color: firebrick; }
h2 { font-size: 20px; }

footer { display: none; }
```

This will create the following CSSOM Tree -



CSS is considered a **"render blocking resource"**. This means that the Render Tree (see below) 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.

3. 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.

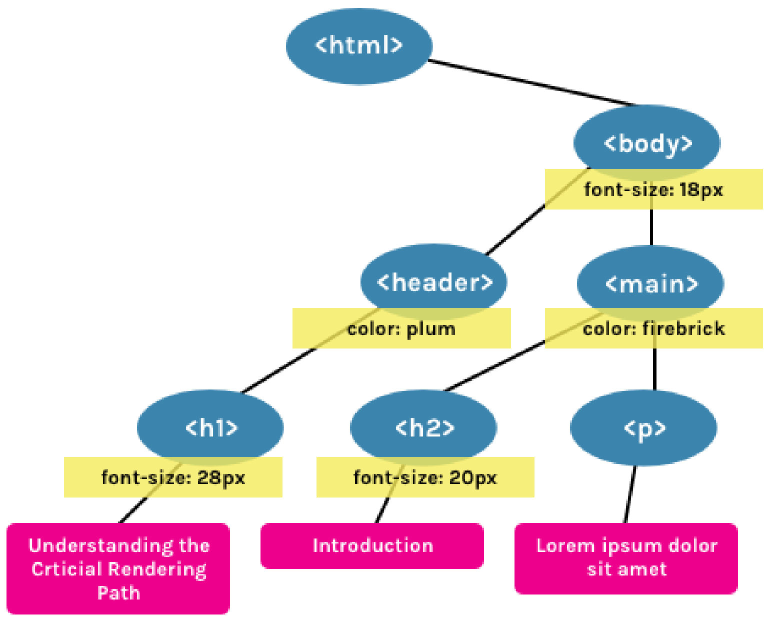
To avoid JavaScript being parser blocking, it can be loaded asynchronously be applying the **`async`** attribute.

```
<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.

For example, the most common meta viewport value is to set the viewport size to correspond to the device width -

```
<meta name="viewport" content="width=device-width,initial-scale=1">
```

If the user visits the webpage on a device with a width of, for example, 1000px, then sizes will be based on that unit. Half the viewport will be 500px, 10vw will be 100px, and so on.

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

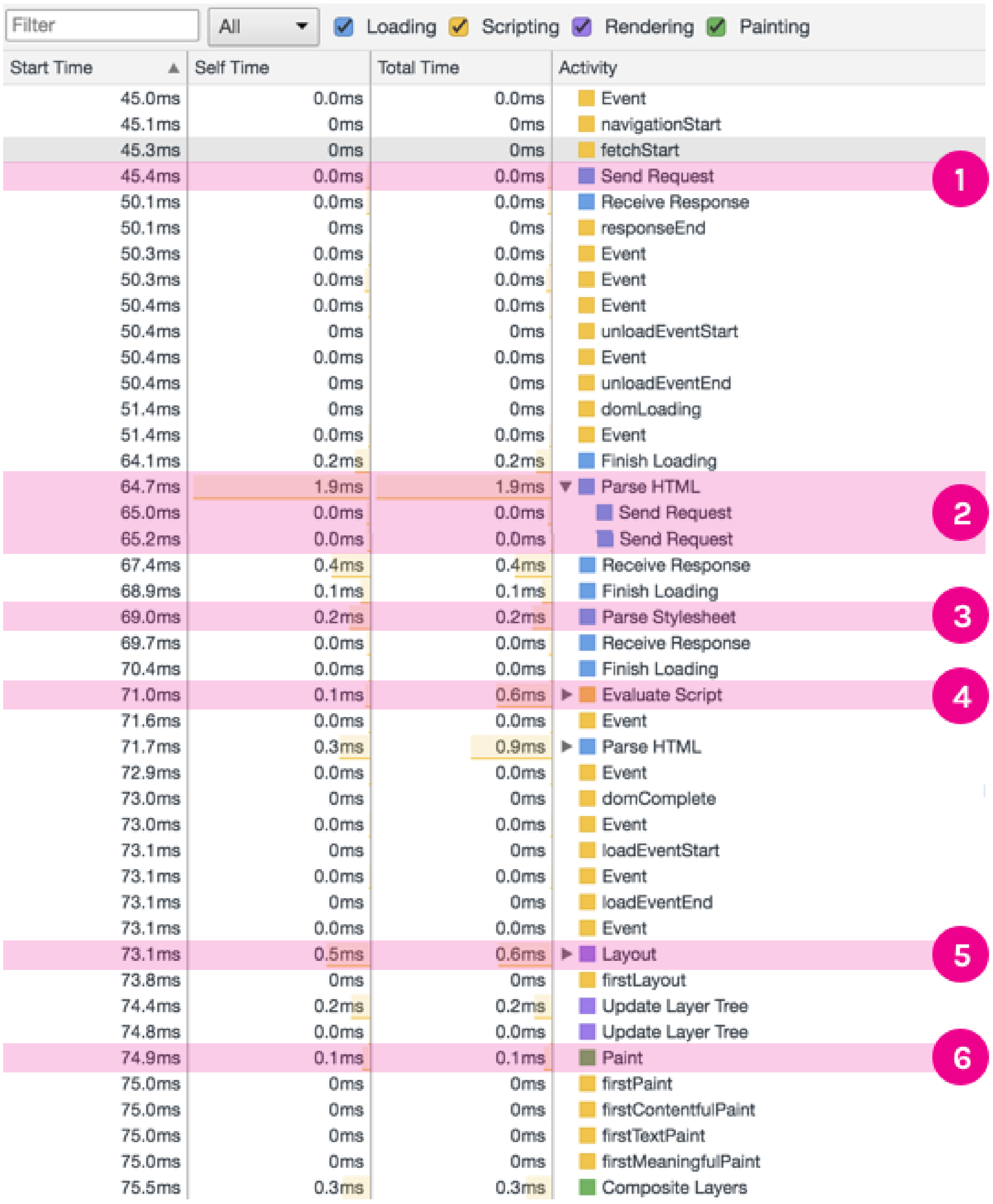
To see the Critical Rendering Path in process, we can inspect it in DevTools. In Chrome, it is under the **Timeline** tab (in Canary, and soon to be Chrome stable, it's renamed **Performance**).

Take for example, our sample HTML from above (with an added `<script>` tag) -

```
<html>
<head>
  <title>Understanding the Critical Rendering Path</title>
  <link rel="stylesheet" href="style.css">
</head>
<body>
  <header>
      <h1>Understanding the Critical Rendering Path</h1>
  </header>
  <main>
      <h2>Introduction</h2>
      <p>Lorem ipsum dolor sit amet</p>
  </main>
  <footer>
```

```
        <small>Copyright 2017</small>
    </footer>
    <script src="main.js"></script>
  </body>
  </html>
```

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



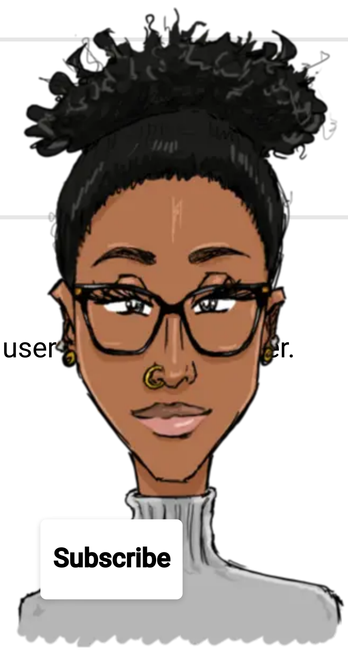
Filter | All | ✓ Loading ✓ Scripting ✓ Rendering ✓ Painting

Start Time ▲	Self Time	Total Time	Activity
45.0ms	0.0ms	0.0ms	Event
45.1ms	0ms	0ms	navigationStart
45.3ms	0ms	0ms	fetchStart
45.4ms	0.0ms	0.0ms	Send Request
50.1ms	0.0ms	0.0ms	Receive Response
50.1ms	0ms	0ms	responseEnd
50.3ms	0.0ms	0.0ms	Event
50.3ms	0.0ms	0.0ms	Event
50.4ms	0.0ms	0.0ms	Event
50.4ms	0ms	0ms	unloadEventStart
50.4ms	0.0ms	0.0ms	Event
50.4ms	0ms	0ms	unloadEventEnd
51.4ms	0ms	0ms	domLoading
51.4ms	0.0ms	0.0ms	Event
64.1ms	0.2ms	0.2ms	Finish Loading
64.7ms	1.9ms	1.9ms	▼ Parse HTML
65.0ms	0.0ms	0.0ms	Send Request
65.2ms	0.0ms	0.0ms	Send Request
67.4ms	0.4ms	0.4ms	Receive Response
68.9ms	0.1ms	0.1ms	Finish Loading
69.0ms	0.2ms	0.2ms	Parse Stylesheet
69.7ms	0.0ms	0.0ms	Receive Response
70.4ms	0.0ms	0.0ms	Finish Loading
71.0ms	0.1ms	0.6ms	▶ Evaluate Script
71.6ms	0.0ms	0.0ms	Event
71.7ms	0.3ms	0.9ms	▶ Parse HTML
72.9ms	0.0ms	0.0ms	Event
73.0ms	0ms	0ms	domComplete
73.0ms	0.0ms	0.0ms	Event
73.1ms	0ms	0ms	loadEventStart
73.1ms	0.0ms	0.0ms	Event
73.1ms	0ms	0ms	loadEventEnd
73.1ms	0.0ms	0.0ms	Event
73.1ms	0.5ms	0.6ms	▶ Layout
73.8ms	0ms	0ms	firstLayout
74.4ms	0.2ms	0.2ms	Update Layer Tree
74.8ms	0.0ms	0.0ms	Update Layer Tree
74.9ms	0.1ms	0.1ms	Paint
75.0ms	0ms	0ms	firstPaint
75.0ms	0ms	0ms	firstContentfulPaint
75.0ms	0ms	0ms	firstTextPaint
75.0ms	0ms	0ms	firstMeaningfulPaint
75.5ms	0.3ms	0.3ms	Composite Layers

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

Based on this information, we can make decisions on how to optimize the Critical Rendering Path. I will go into some of these techniques in later articles.



Subscribe to the Newsletter

Receive quality articles written by Ire Aderinokun, frontend developer and user[illegible]r.

Email Address*

First Name

Subscribe

←

3 New CSS Features to Learn in 2017

css

→

Rules for Using ARIA in HTML

accessibility, *html*

27 Comments bitsofcode Disqus' Privacy Policy 1 Login

Recommend 25 Tweet Share Sort by Best

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?