## **Assignment 1**

The web browser is at the heart of how we experience the internet. Every day, millions visit websites via browsers. Five major browsers — Chrome, Firefox, Internet Explorer, Safari and Opera — account 95% of web traffic.

A major role of a web browser is to accept a web URL via an address bar, fetch resources, and display them on the screen.

Browsers functionality can be classified into four major sections and these include:

1. Fetch
2. Process
3. Display
4. Storage

## **A) The main functionality of the browser**

1. The main task is to collect information from the Internet and make it accessible to users.
2. A web browser can be used to visit any website. When we type a URL into a browser, the web server redirects us to that website.
3. It simplifies Internet surfing because once we arrive at a website, we can quickly check the hyperlinks and access a wealth of information.
4. Internal cache is used by browsers and is saved so that the user can open the same webpage multiple times without losing any data.
5. A web browser can open multiple web pages at the same time. Back, forward, reload, stop reload, home, and other options are available on these web browsers, making them simple and convenient to use.

* **What happens when we type a URL:**

URL stands for Uniform Resource Locator. URL is the address of the website which you can find in the address bar of your web browser. It is a reference to a resource on the internet, be it images, hypertext pages, audio/video files, etc.

For Example - <https://www.pesto.tech/become-a-developer>

DNS is short for Domain Name System. Like a phonebook, DNS maintains and maps the name of the website, i.e. URL, and particular IP address it links to. Every URL on the internet has a unique IP address which is of the computer which hosts the server of the website requested.

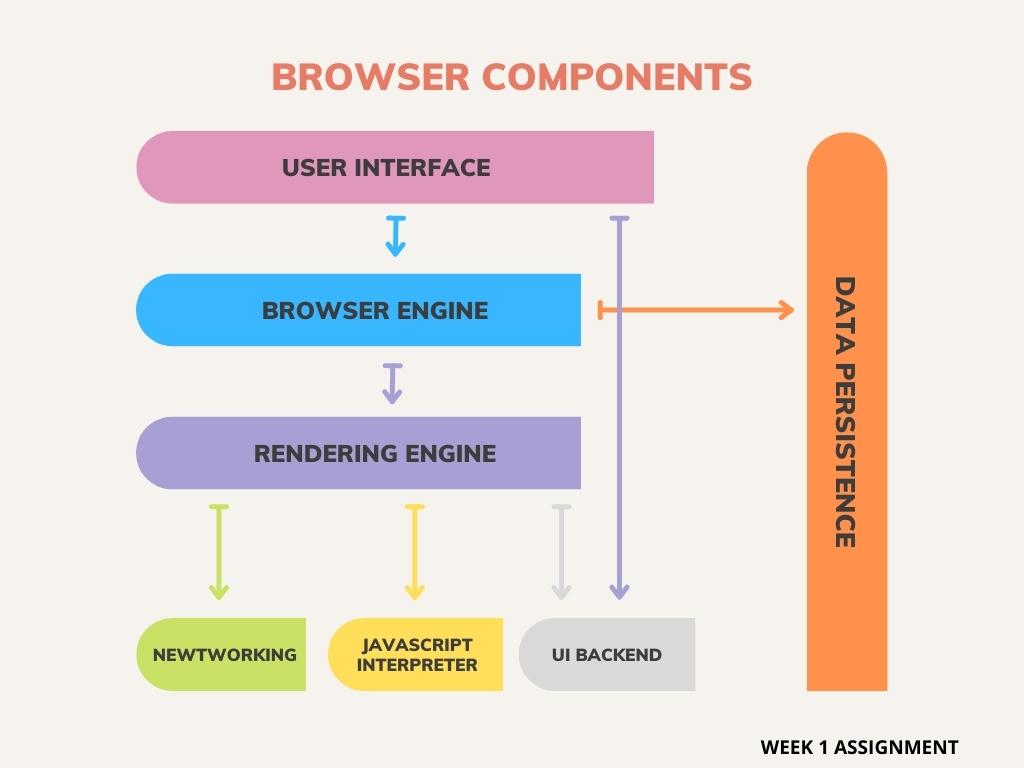
Steps for what happens when we enter a URL :

* 1. Browser checks cache for DNS entry to find the corresponding IP address of website. It looks for following cache. If not found in one, then continues checking to the next until found. a) Browser Cache b) Operating Systems Cache c) Router Cache
  2. If not found in cache, ISP’s (Internet Service Provider) DNS server initiates a DNS query to find IP address of server that hosts the domain name.
  3. The requests are sent using small data packets that contain information content of request and IP address it is destined for.
  4. Browser initiates a TCP (Transfer Control Protocol) connection with the server using synchronize(SYN) and acknowledge(ACK) messages.
  5. Browser sends an HTTP request to the web server. GET or POST request.
  6. Server on the host computer handles that request and sends back a response. It assembles a response in some format like JSON, XML and HTML.
  7. Server sends out an HTTP response along with the status of response.
  8. Browser displays HTML content Finally, Done.

## **B) High Level Components of a browser**The user interface: this includes the address bar, back/forward button, bookmarking menu, etc. Every part of the browser display except the window where you see the requested page.

1. The browser engine: marshals actions between the UI and the rendering engine.
2. The rendering engine : responsible for displaying requested content. For example if the requested content is HTML, the rendering engine parses HTML and CSS, and displays the parsed content on the screen.
3. Networking: for network calls such as HTTP requests, using different implementations for different platform behind a platform-independent interface.
4. UI backend: used for drawing basic widgets like combo boxes and windows. This backend exposes a generic interface that is not platform specific. Underneath it uses operating system user interface methods.
5. JavaScript interpreter: Used to parse and execute JavaScript code.
6. Data storage: This is a persistence layer. The browser may need to save all sorts of data locally, such as cookies. Browsers also support storage mechanisms such as localStorage, IndexedDB, WebSQL and FileSystem. It is important to note that browsers such as Chrome run multiple instances of the rendering engine: one for each tab. Each tab runs in a separate process.

**Refer The Image for Explanation:-Assignment-1/images/Browser\_component\_assignment1.jpg**



## **C) Rendering engine and its use**

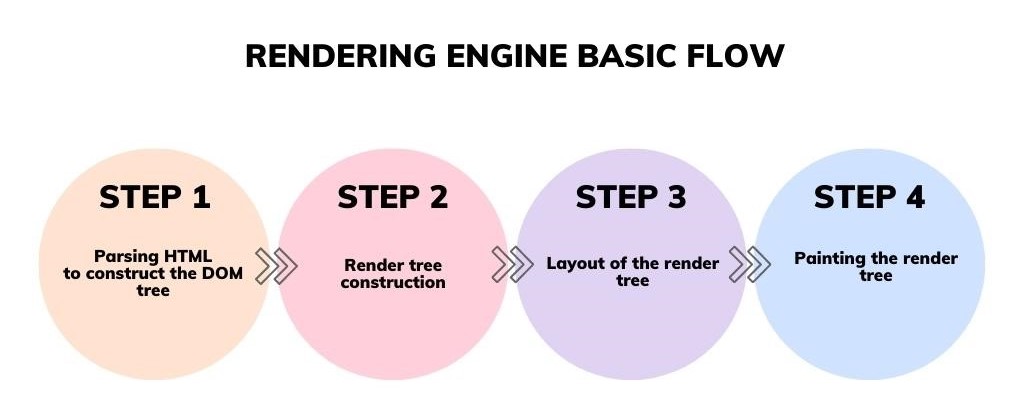
* A rendering engine is software that draws text and images on the screen. The engine draws structured text from a document (often HTML), and formats it properly based on the given style declarations (often given in CSS). Examples of layout engines: Blink, Gecko, EdgeHTML, WebKit.

## **The main flow:**

The rendering engine will start getting the contents of the requested document from the networking layer. This will usually be done in 8kB chunks.

After that, this is the basic flow of the rendering engine:

**Refer The Image for Explanation :: Assignment-1/images /Rendering**  **engine basic flow.jpg**



## **D) Parsers (HTML, CSS, etc):**

HTML or CSS is a lightweight HTML/CSS parser written in C that allows applications to prepare a HTML document for rendering or conversion. HTM or CSS is extremely portable and only requires a C99 compiler like GCC, Clang, Visual C, etc. HTML or CSS is also extremely memory efficient, utilizing a shared string pool and smart CSS cache to minimize the size of a HTML document in memory.

Features include:

1. HTML 5 markup parser
2. CSS 3 stylesheet parser
3. OFC/OFF/TTC/TTF font file parser (metadata only)
4. GIF/JPG/PNG image file parser (metadata only)
5. Functions to calculate CSS properties for a given node in a HTML document
6. Functions to extract HTML “runs” consisting of CSS properties, content strings, and image references that can be rendered directly, including the :before and :after content from a stylesheet

HTML or CSS does not support dynamic HTML content created using Javascript in a HTML document, as such content is typically used for interactive web pages while HTML or CSS is intended for use with static content.

## **E) Script Processors:**

* This is a rundown of how browsers process JavaScript references within HTML.

The original assertion was that concatenating (or bundling) JavaScript and CSS assets[1] might improve performance by reducing load times, but inevitably the conversation ended up including topics such as moving scripts to the bottom, minification, CDNs and HTTP/2.

In order to assess the consequences of any such decision, it helps to understand how browsers work: When the browser processes an HTML document, it does so from top to bottom. Upon encountering a <script> tag, it halts (“blocks”) further processing[2] in order to download the referenced script file. Only after that download has completed and the respective JavaScript code has been processed, HTML processing continues.

**Let’s see the following document:**

<!DOCTYPE html>  
<html>  
 <head>  
 <meta charset="utf-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1">  
 <title>Hello World</title>  
 <link rel="stylesheet" href="main.css">  
 <script src="foo.js"></script>  
 </head>  
 <body>  
 <h1>Hello World</h1>  
 <script src="bar.js"></script>  
 <p>lorem ipsum dolor sit amet</p>  
 <script src="baz.js"></script>  
 </body>  
</html>

The browser might actually begin rendering the page even before it has fully downloaded the HTML file. Thus you might see the browser window reading “Hello World” (thanks to the <title> tag) while the page is still blank.

Once we arrive at <script src="foo.js">, processing halts as described above. Afterwards, we continue to <script src="bar.js">, repeat the same procedure, and then move on to <script src="baz.js"> for the final piece. That leaves us with the following sequence: |-foo.js-| |-bar.js-| |-baz.js-|

+---------------------------------------------------------------------------------------------> Concatenation would mean combining these files into a single one: |-foo-bar-baz.js-| +---------------------------------------------------------------------------------------------> While the amount of content transferred remains identical[3], this is generally faster because there’s less networking overhead. (Obviously I’m simplifying a bit here.)

As you might have guessed from this (poor man’s) visualization, there’s another approach. We could parallelize the retrieval of JavaScript files: |-foo.js-| |-bar.js-| |-baz.js-| +---------------------------------------------------------------------------------------------------> Browsers these days support this with the simple addition of a dedicated attribute: <script defer> (implied by <script type="module">). In fact, there’s also another, similar attribute: async — except this one doesn’t guarantee order of execution; see Asynchronous vs Deferred JavaScript for details.[4] However, these attributes don’t work for inline scripts (of which, unfortunately, there were a few in the project at hand), so those would likely execute before the deferred external scripts they depend on become available.

Now, you might argue that HTTP/2 makes all of this a non-issue because it reduces protocol overhead — but in fact, even HTTP/2 is still prone to the laws of physics:

As described above, <script> tags are processed sequentially — which means that the browser doesn’t know it should retrieve bar.js until after foo.js has been fully loaded. Thus it actually has to wait before even requesting that file from the server:

|-- → foo.js --| |-- ← foo.js --| |-- → bar.js --| |-- ← bar.js --| +---------------------------------------------------------------> t

→ request ← response Depending on connectivity, that latency can be significant.

However, if we were using defer, those <script> tags would be non-blocking, which means the browser could request both files simultaneously: |-- → foo.js --| |-- ← foo.js --| |-- → bar.js --| |-- ← bar.js --| +---------------------------------------------------------------> t

→ request ← response This is why concatenation can actually be a net negative with HTTP/2, as it prevents parallel downloads:

| | |-- → foo-bar.js --| | |-- ← foo-bar.js --| | +---------------------------------------------------------------> t

→ request ← response Network protocols aside, it’s generally good practice to relegate script tags to the bottom in order to avoid unnecessarily blocking static HTML content. In the example above, even if the entire HTML document has already been downloaded, if foo.js and/or bar.js are slow to load (for which there are myriad potential reasons), they’d prevent the content below from being displayed.

## **F) Tree construction:**

**To construct the render tree, the browser roughly does the following:**

The CSSOM and DOM trees are combined into a render tree, which is then used to compute the layout of each visible element and serves as an input to the paint process that renders the pixels to screen. Optimizing each of these steps is critical to achieving optimal rendering performance.

In the previous section on constructing the object model, we built the DOM and the CSSOM trees based on the HTML and CSS input. However, both of these are independent objects that capture different aspects of the document: one describes the content, and the other describes the style rules that need to be applied to the document.

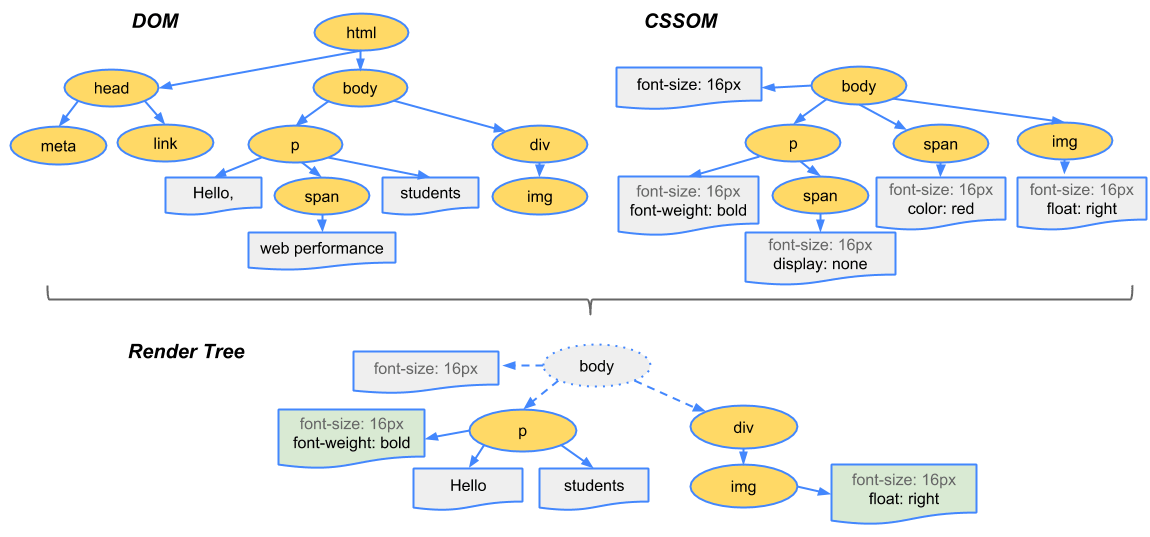
a) The DOM and CSSOM trees are combined to form the render tree. b) Render tree contains only the nodes required to render the page. c) Layout computes the exact position and size of each object. d) The last step is paint, which takes in the final render tree and renders the pixels to the screen. e) First, the browser combines the DOM and CSSOM into a "render tree," which captures all the visible DOM content on the page and all the CSSOM style information for each node.

f) Starting at the root of the DOM tree, traverse each visible node.

* + Some nodes are not visible (for example, script tags, meta tags, and so on), and are omitted since they are not reflected in the rendered output.
  + Some nodes are hidden via CSS and are also omitted from the render tree; for example, the span node in the example above is missing from the render tree because we have an explicit rule that sets the "display: none" property on it. g) For each visible node, find the appropriate matching CSSOM rules and apply them. h) Emit visible nodes with content and their computed styles.

**Here's a quick recap of the browser's steps:**

* Process HTML markup and build the DOM tree.
* Process CSS markup and build the CSSOM tree.
* Combine the DOM and CSSOM into a render tree.
* Run layout on the render tree to compute geometry of each node.
* Paint the individual nodes to the screen.



## **G) Order of script processing:**

* + Designing your web page using JavaScript requires attention to the order in which your code appears and whether you are encapsulating code into functions or objects, all of which impact the order in which the code runs.
  + The Location of JavaScript on Your Web Page Since the JavaScript on your page executes based on certain factors, let's consider where and how to add JavaScript to a web page.

**There are basically three locations into which we can attach JavaScript:**

Directly into the head of the page Directly into the body of the page From an event handler/listener It doesn't make any difference whether the JavaScript is within the web page itself or in external files linked to the page. It also doesn't matter whether the event handlers are hard-coded into the page or added by the JavaScript itself (except that they can't be triggered before they are added).

## **H) Layout and Painting:**

The process of a web browser turning HTML, CSS, and JavaScript into a finished visual representation is quite complex and involves a good bit of magic. Here’s a simplified set of steps the browser goes through:

Browser creates the DOM and CSSOM. Browser creates the render tree, where the DOM and styles from the CSSOM are taken into account (display: none elements are avoided). Browser computes the geometry of the layout and its elements based on the render tree. Browser paints pixel by pixel to create the visual representation we see on the screen. In this article, I’d like to focus on the last part: painting.

All of those steps combined is a lot of work for a browser to do on load… and actually, not just on load, but any time the DOM (or CSSOM) is changed. That’s why many web developers tend to partially solve this by using some sort of frontend framework, such as React which, apart from many other advantages, can help to highly optimize changes in the DOM to avoid unnecessary recalculating or rendering.

You may have heard terms such as state, component rendering, or immutability. All of those have something to do with the optimization of DOM changes, or in other words, to only make changes to the DOM when it’s necessary.

To give an example, the state of a web application may change, and that would lead to a change in UI. However, certain (or many) components are not affected by this change. What React helps to do is limit the writing to the DOM for elements that are actually affected by a change in state and ultimately limit the rendering to the smallest part of the web application possible:

DOM/CSSOM → render tree → layout → painting However, browser painting is special in its own way, as it can happen even without any changes to the DOM and/or CSSOM.