**When a user enters an URL in the browser, how does the browser fetch the desired result? Explain this with the below in mind and Demonstrate this by drawing a diagram for the same.**

**1. Main functionality of the browser**

The main functionality of the browser is to serve the requested resource from the server and display it to the user. There are different forms of data the browser can show to a user. Mostly it is an HTML document, but the data can be PDF, Image, or some other data content type.  
The location of the resource can be specified by the user using a URI (Uniform Resource Identifier).  
The way the browser interprets and displays HTML files is specified in the HTML and CSS specifications. These specifications are maintained by W3C(World Wide Web Consortium) organization.

**2. High Level Components of a Browser**

**The Browser's main components are:**

1. **The UI (User Interface)**

UI includes the address bar, back/forward button, bookmarking menu, reloading button, and a display where you see the requested page.

1. **The Browser Engine**

It assembles and arranges actions between the UI and the rendering engine.

1. **The Rendering Engine**

It is responsible for displaying the requested content.

The rendering engine parses the HTML and CSS and displays the parsed content on the screen/browser's display.

1. **Networking**

For network calls such as HTTP requests. It uses different implementations for different platforms behind a platform-independent interface.

1. **UI Backend**

It is for drawing basic widgets like Combo boxed and windows. It is not platform-specific. Underneath it uses OS (Operating system) UI methods.

1. **JavaScript Interpreter**

As the name suggests, it is for parsing and executing JS code.

1. **Data Storage**

It is a persistence layer for saving and storing data locally as cookies, session storage, local storage, etc.

[](https://user-images.githubusercontent.com/101351789/160363782-1ddaf12a-428f-4b2a-ba4c-ce20be2a62f2.png)

**3. Rendering Engine & its use**

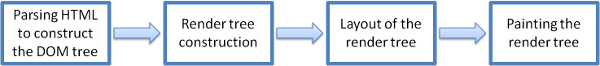
The responsibility of the rendering engine is to display the requested page on the browser screen.

The browsers use different rendering engines.

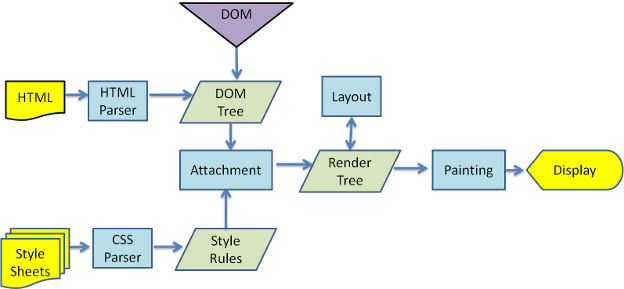
a. Firefox -> Gecko

b. Safari -> WebKit

c. Chrome & Opera -> Blink (a fork of WebKit).

[](https://user-images.githubusercontent.com/101351789/160368334-12eaf79a-6b4d-4ed3-a5be-15d32c0cb2ca.png)

***Figure: The Basic Flow of a Rendering Engine.***

[](https://user-images.githubusercontent.com/101351789/160368915-f92ee8cc-0f1e-491c-87ef-d57c3e5bd3bc.png)

***Figure: The main flow of WebKit Rendering Engine.***

**4. Parser(HTML & CSS etc.)**

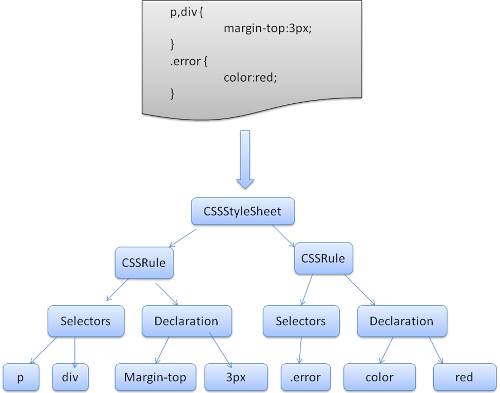
Parsing means analyzing and converting a program into an internal format that a runtime environment can run, for example, the JavaScript engine inside browsers.

The browser parses HTML into a DOM tree, HTML parsing involves tokenization and tree construction. If the document is well-formed, parsing it is straightforward and faster.

When the HTML parser finds non-blocking resources, such as an image, the browser will request those and continue parsing but <script> tags--particularly those without an async or defer attribute—block rendering, and pauses parsing of HTML.

**CSS Parsing**

When the browser encounters CSS styles, it parses the text into the CSS Object Model (CSSOM), a data structure. It is then used for styling layouts and painting. The browser then creates a render tree from both these structures to be able to paint the content to the screen.

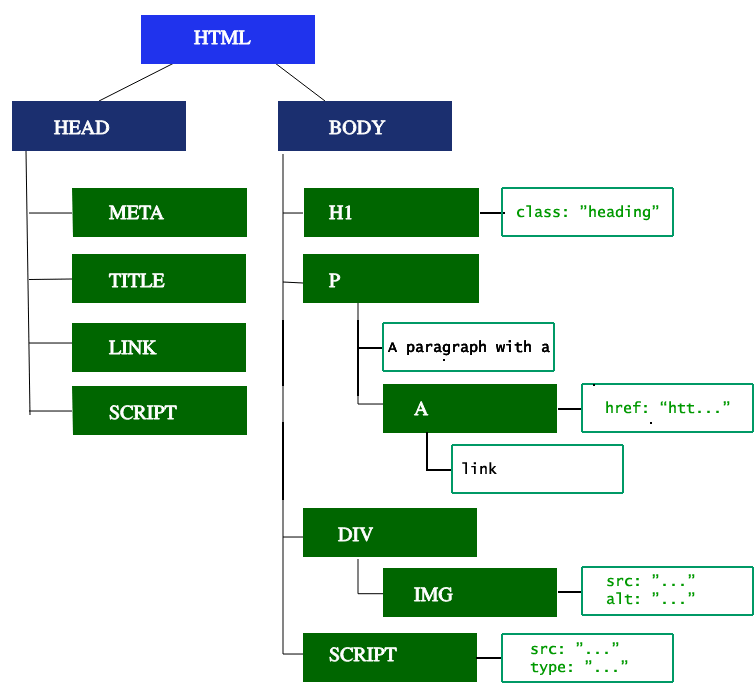
[](https://user-images.githubusercontent.com/101351789/160385803-e5f77df9-5635-4045-909e-eb9a112b470b.png)

**5. Script Processor**

The model of the web is synchronous. Authors expect scripts to be parsed and executed immediately when the parser reaches a <script> tag. The parsing of the document halts until the script has been executed. If the script is external then the resource must first be fetched from the network–this is also done synchronously, and parsing halts until the resource is fetched. This was the model for many years and is also specified in HTML4 and 5 specifications. Authors can add the "defer" attribute to a script, in which case it will not halt document parsing and will execute after the document is parsed. HTML5 adds an option to mark the script as asynchronous so it will be parsed and executed by a different thread.

**6. Tree Construction**

The first step is processing the HTML and building DOM(Document Object Model). The DOM tree describes the content of the document. The <html> element is the first tag and root node of the document tree. The tree reflects the relationship and hierarchies between different tags. Tags nested within other tags are child nodes. The greater the number of DOM nodes, the longer it takes to construct the DOM tree.

[](https://user-images.githubusercontent.com/101351789/160390787-bec22a22-0ab4-41d3-b821-6a54c971dda0.gif)

**Preload Scanner**

While the browser builds the DOM tree, this process occupies the main thread. As this happens, the preload scanner will parse through the content available and also request high-priority resources like CSS, JavaScript, and web fonts. It will retrieve the resources in the background so that by the time the main HTML parser reaches the requested asset, they may already be in flight, or have been downloaded. The optimizations the preload scanner provides reduce blockages.

**Building the CSSOM**

The second step is processing CSS and building a CSSOM tree. The CSS object model is similar to DOM. The DOM and CSSOM are both trees. They are independent data structures. The browser converts the CSS rules into a map of styles it can understand and work with. The browser goes through each rule set in the CSS, creating a tree of nodes with a parent, child, and sibling relationships based on the CSS selectors.

Building the CSSOM is very, very fast and is not displayed in a unique color in current developer tools. In terms of web performance optimization, there is lower hanging fruit, as the total time to create the CSSOM is generally less than the time it takes for one DNS lookup.

**7. JavaScript Compilation**

While the CSS is being parsed and the CSSOM created, other assets, including JavaScript files, are downloading (thanks to the preload scanner). JavaScript is interpreted, compiled, parsed, and executed. The scripts are parsed into abstract syntax trees. Some browser engines take the Abstract Syntax Tree and pass it into an interpreter, outputting bytecode which is executed on the main thread. This is known as JavaScript compilation.

**Building the Accessibility Tree**

The browser also builds an accessibility tree that assistive devices use to parse and interpret content. The accessibility object model (AOM) is like a semantic version of the DOM. The browser updates the accessibility tree when the DOM is updated. The accessibility tree is not modifiable by assistive technologies themselves.

Until the AOM is built, the content is not accessible to screen readers.

**8. Layout**

The fourth step in the critical rendering path is running the layout on the render tree to compute the geometry of each node. The layout is the process by which the width, height, and location of all the nodes in the render tree are determined, plus the determination of the size and position of each object on the page. *Reflow* is any subsequent size and position determination of any part of the page or the entire document.

Once the render tree is built, the layout commences. The render tree identified which nodes are displayed (even if invisible) along with their computed styles, but not the dimensions or location of each node. To determine the exact size and location of each object, the browser starts at the root of the render tree and traverses it.

On the web page, almost everything is in a box. Different devices and different desktop preferences mean an unlimited number of different viewport sizes. In this phase, considering the viewport size, the browser determines what the dimensions of all the different boxes are going to be on the screen. Taking the size of the viewport as its base, the layout generally starts with the body, laying out the dimensions of all the body's descendants, with each element's box model properties.

The first time the size and position of nodes are determined is called *layout*. Subsequent recalculations of node size and locations are called *reflows*.

**9. Painting**

The last step is to paint individual nodes on the screen, the first occurrence of which is called first meaningful paint. In the painting or rasterization phase, the browser converts each box calculated in the layout phase to actual pixels on the screen. Painting involves drawing every visual part of an element to the screen, including text, colors, borders, shadows, and replaced elements like buttons and images. The browser needs to do this super quickly.

Painting can break the elements in the layout tree into layers. Promoting content into layers on the GPU (instead of the main thread on the CPU) improves paint and repaint performance. Some specific properties and elements instantiate a layer, including <video> and <canvas>, and any element which has the CSS properties of opacity, a 3D transform, will-change, and a few others. These nodes will be painted onto their layer, along with their descendants, unless a descendant necessitates its layer for one (or more) of the above reasons.

Layers do improve performance but are expensive when it comes to memory management, so should not be overused as part of web performance optimization strategies