Exercise1.1:1.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.(2-3hours)

a. What is the main functionality of the browser?

b. High Level Components of a browser.

c. Rendering engine and its use.

d. Parsers (HTML, CSS, etc)

e. Script Processors

f. Tree construction

g. Order of script processing

h. Layout and Painting

Answer:

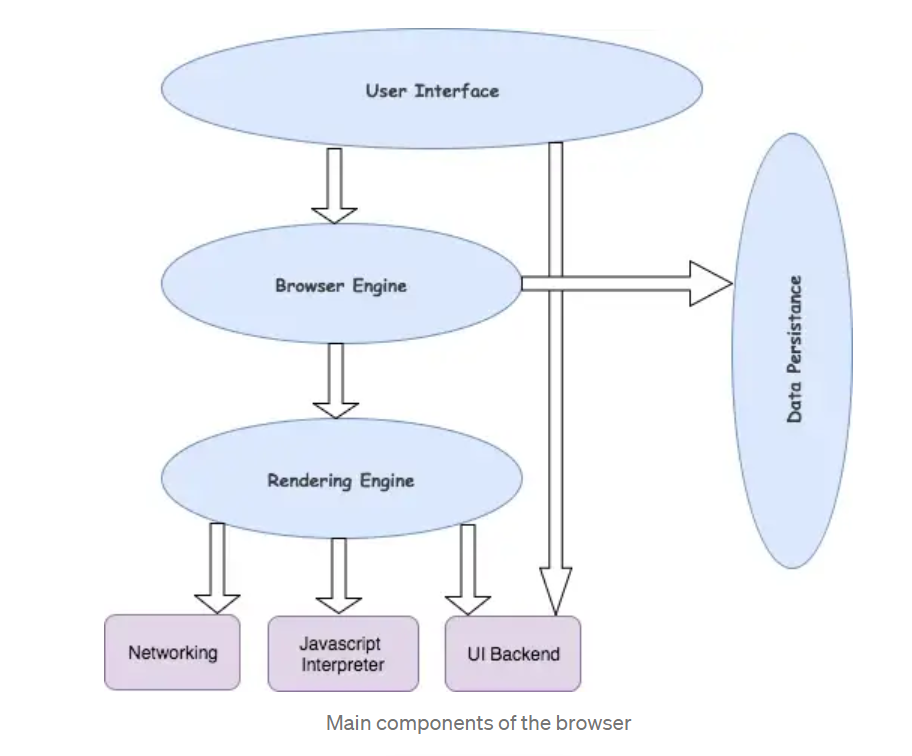
A browser is a software application used to locate, retrieve and display content on the World Wide Web, including Web pages, images, video and other files. As a client/server model, the browser is the client run on a computer that contacts the Web server and requests information. The Web server sends the information back to the Web browser which displays the results on the computer or other Internet-enabled device that supports a browser.

Today’s browsers are fully-functional software suites that can interpret and display HTML Web pages, applications, JavaScript, AJAX and other content hosted on Web servers. Many browsers offer plug-ins which extend the capabilities of the software so it can display multimedia information (including sound and video), or the browser can be used to perform tasks such as videoconferencing, to design web pages or add anti-phishing filters and other security features to the browser.

A browser is a group of structured codes which together performs a series of tasks to display a web page on the screen. According to the tasks they perform, these codes are made as different components.

**High-level architecture of browser**

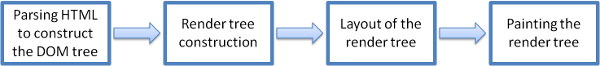
The below image shows the main components of a web browser:



1. **The User Interface**: The user interface is the space where User interacts with the browser. It includes the address bar, back and next buttons, home button, refresh and stop, bookmark option, etc. Every other part, except the window where requested web page is displayed, comes under it.
2. **The Browser Engine**: The browser engine works as a bridge between the User interface and the rendering engine. According to the inputs from various user interfaces, it queries and manipulates the rendering engine.
3. **The Rendering Engine**: The rendering engine, as the name suggests is responsible for rendering the requested web page on the browser screen. The rendering engine interprets the HTML, XML documents and images that are formatted using CSS and generates the layout that is displayed in the User Interface. However, using plugins or extensions, it can display other types data also. Different browsers user different rendering engines:  
   \* Internet Explorer: Trident  
   \* Firefox & other Mozilla browsers: Gecko  
   \* Chrome & Opera 15+: Blink  
   \* Chrome (iPhone) & Safari: Webkit
4. **Networking**: Component of the browser which retrieves the URLs using the common internet protocols of HTTP or FTP. The networking component handles all aspects of Internet communication and security. The network component may implement a cache of retrieved documents in order to reduce network traffic.
5. **JavaScript Interpreter:**It is the component of the browser which interprets and executes the javascript code embedded in a website. The interpreted results are sent to the rendering engine for display. If the script is external then first the resource is fetched from the network. Parser keeps on hold until the script is executed.
6. **UI Backend**: UI backend is used for drawing basic widgets like combo boxes and windows. This backend exposes a generic interface that is not platform specific. It underneath uses operating system user interface methods.
7. **Data Persistence/Storage:** This is a persistence layer. Browsers support storage mechanisms such as localStorage, IndexedDB, WebSQL and FileSystem. It is a small database created on the local drive of the computer where the browser is installed. It manages user data such as cache, cookies, bookmarks and preferences.

# **Rendering engine**

The networking layer will start sending the contents of the requested documents to the rendering engine in chunks of 8KBs.



Rendering engine basic flow

The rendering engine parses the chunks of HTML document and convert the elements to DOM nodes in a tree called the “c**ontent tree**” or the “**DOM tree**”. It also parses both the external CSS files as well in style elements.

While the DOM tree is being constructed, the browser constructs another tree, the **render tree**. This tree is of visual elements in the order in which they will be displayed. It is the visual representation of the document. The purpose of this tree is to enable painting the contents in their correct order. Firefox calls the elements in the render tree “frames”. WebKit uses the term renderer or render object.

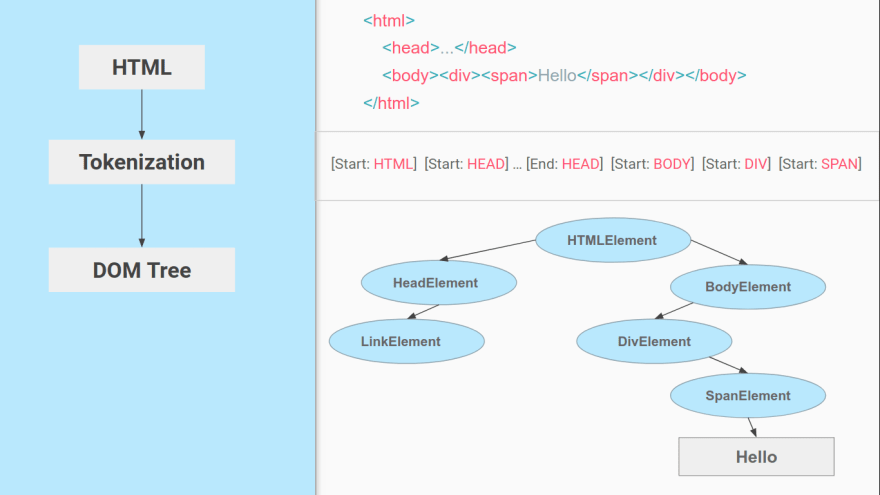
After the construction of the render tree, it goes through a “**layout process**” of the render tree. When the renderer is created and added to the tree, it does not have a position and size. The process of calculating these values is called layout or reflow. This means giving each node the exact coordinates where it should appear on the screen. The position of the root renderer is 0,0 and its dimensions are the viewport–the visible part of the browser window. All renderers have a “layout” or “reflow” method, each renderer invokes the layout method of its children that need layout.

The next stage is **painting**. In the painting stage, the render tree is traversed and the renderer’s “paint()” method is called to display content on the screen. Painting uses the UI backend layer.

The rendering engine always tries to display the contents on the screen as soon as possible for better user experience. It does not wait for the HTML parsing to complete before starting to build and layout the render tree. It parses and displays the content it has received from the network, while rest of the contents stills keeps coming from the networ

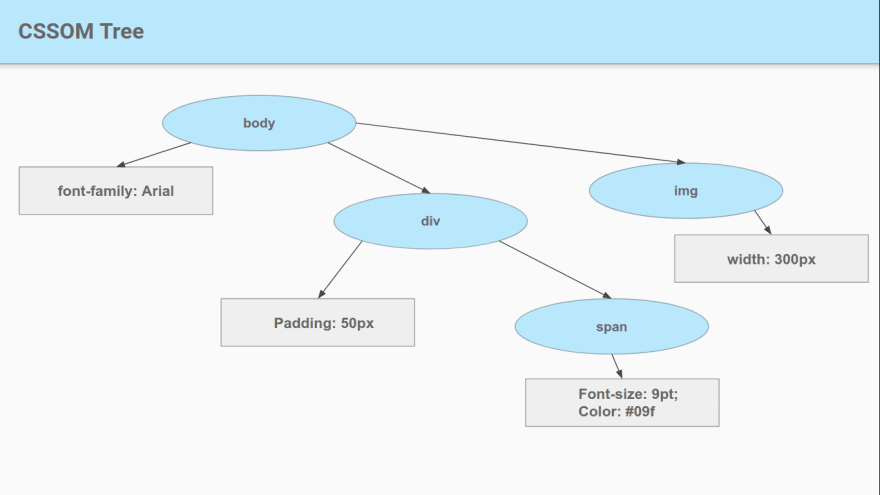
# HTML Parsing

So we have HTML content at the beginning which goes through a process called tokenization, tokenization is a common process in almost every programming language where code is split into several tokens which are easier to understand while parsing. This is where the HTML's parser understands which is the start and which is the end of the tag, which tag it is and what is inside the tag.

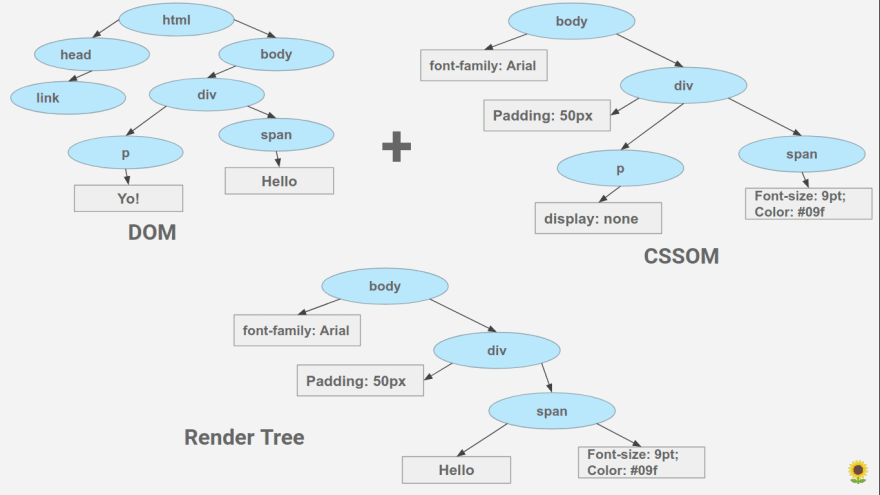
Now we know, html tag starts at the top and then the head tag starts before the html ends so we can figure out that the head is inside html and create a tree out of it. Thus we then get something called a parse tree which eventually becomes a DOM tree as shown in the image below:  
[](https://res.cloudinary.com/practicaldev/image/fetch/s--40NGH5el--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/o91r8lupx8elero5djh3.png)

DOM tree is what we access when we do document.getElementById or document.querySelector in JavaScript.

Just like HTML, CSS goes through a similar process where we have the CSS text and then the tokenization of CSS to eventually create something called a CSSOM or CSS Object Model.

This is what a CSS Object Model looks like:  
[](https://res.cloudinary.com/practicaldev/image/fetch/s--maAtq7ut--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/vg9595umg17jzkcdvm7f.png)

now we have DOM and CSSOM so we got every information that is required to get our screens painted.

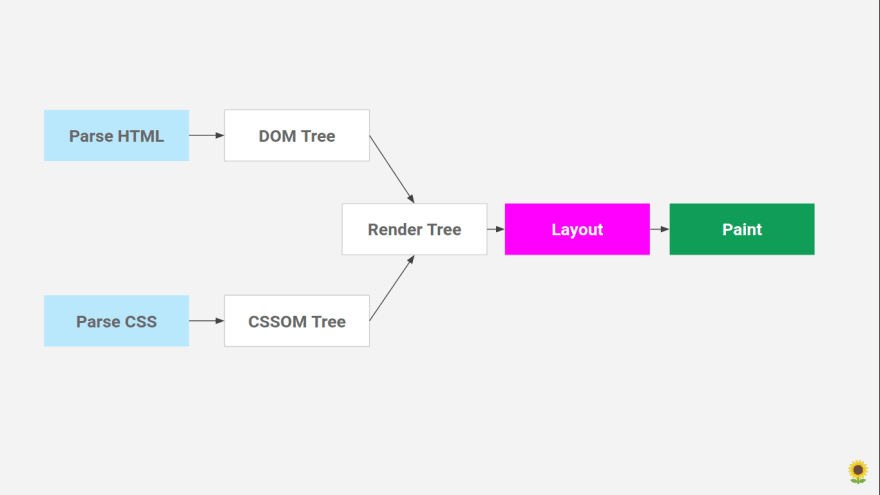
[](https://res.cloudinary.com/practicaldev/image/fetch/s--T2Lw75n3--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/0av60eim9egeoihfdfo1.png)**Rendering of Web Page:** For rendering, a DOM and CSSOM are merged to form something called a Render Tree. Render Tree has the information required to mark and paint elements on the screen

Also while forming a Render Tree, elements like <head>, <link>, <script>, and elements with 'display: none' in CSS are ignored since they are not rendered on the screen.

Note that the elements with 'opacity:0' or 'visibility: none' are included in the render tree, even though they are not painted on the screen they do take their positions and render as an empty space and thus are required for calculations.

Huushh! you've come a long way! drink a glass of water maybe?

So now we have a render tree with all the information that is needed to create a visual page. Now, the renderer will use this information to create a Layout and then a Paint, we will talk about Layout and Paint in next point before that here's what the overall process looks like:

[](https://res.cloudinary.com/practicaldev/image/fetch/s--uJ44oL2X--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/s0cj1gr1srugpkc34985.png)

## Layout

The layout is where the elements are marked on the screen. The layout includes all the calculations and mathematics behind an element's position so it takes all the properties related to the position (height, width, position, top left right bottom, etc) from The Render Tree and places the elements on the screen.

## Paint

After Layout, a Paint happens. Paint takes properties like color, background-color, border-color, box-shadow, etc. to paint the screen with colors.

After the paint, we see the content on the screen and the first time we see something other than a white screen is called 'First Paint'. The term First Paint is used in performance reports to show how long your website took to show something on the screen.

Order of Processing

Depending on the type of object, the task may have more than one Process page on which you can write scripts. The scripts in the Process pages are processed in the following order:

1. **Pre-Process** page and **Process** page
2. **Child Post Process** page
3. **Post Process** page

* The last step is paint, which takes in the final render tree and renders the pixels to the screen.

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

1. 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.
2. For each visible node, find the appropriate matching CSSOM rules and apply them.
3. Emit visible nodes with content and their computed styles.
4. The final output is a render tree that contains both the content and style information of all the visible content on the screen. **With the render tree in place, we can proceed to the "layout" stage.**
5. Up to this point we've calculated which nodes should be visible and their computed styles, but we have not calculated their exact position and size within the [viewport](https://web.dev/web/fundamentals/design-and-ux/responsive/#set-the-viewport) of the device---that's the "layout" stage, also known as "reflow."
6. To figure out the exact size and position of each object on the page, the browser begins at the root of the render tree and traverses it. Let's consider a simple, hands-on example:

## Script Processor

Allows inline and stored scripts to be executed within ingest pipelines.

See [How to use scripts](http://man.hubwiz.com/docset/ElasticSearch.docset/Contents/Resources/Documents/www.elastic.co/guide/en/elasticsearch/reference/current/modules-scripting-using.html) to learn more about writing scripts. The Script Processor leverages caching of compiled scripts for improved performance. Since the script specified within the processor is potentially re-compiled per document, it is important to understand how script caching works. To learn more about caching see [Script Caching](http://man.hubwiz.com/docset/ElasticSearch.docset/Contents/Resources/Documents/www.elastic.co/guide/en/elasticsearch/reference/current/modules-scripting-using.html#modules-scripting-using-caching)

# TLDR;

* Parsing and Rendering turn the HTML content into a web page with colors and backgrounds and pictures.
* **HTML Parsing:** HTML Text -> Tokenization -> DOM Tree
* **CSS Parsing:** CSS Text -> Tokenization -> CSSOM Tree
* DOM and CSSOM are merged to form a Render Tree
* Render Tree has all the information required to mark and paint the screen.
* Render Tree -> Layout -> Paint
* The layout does the maths for placing the elements
* Paint paints the elements with colors, backgrounds, shadows, etc.