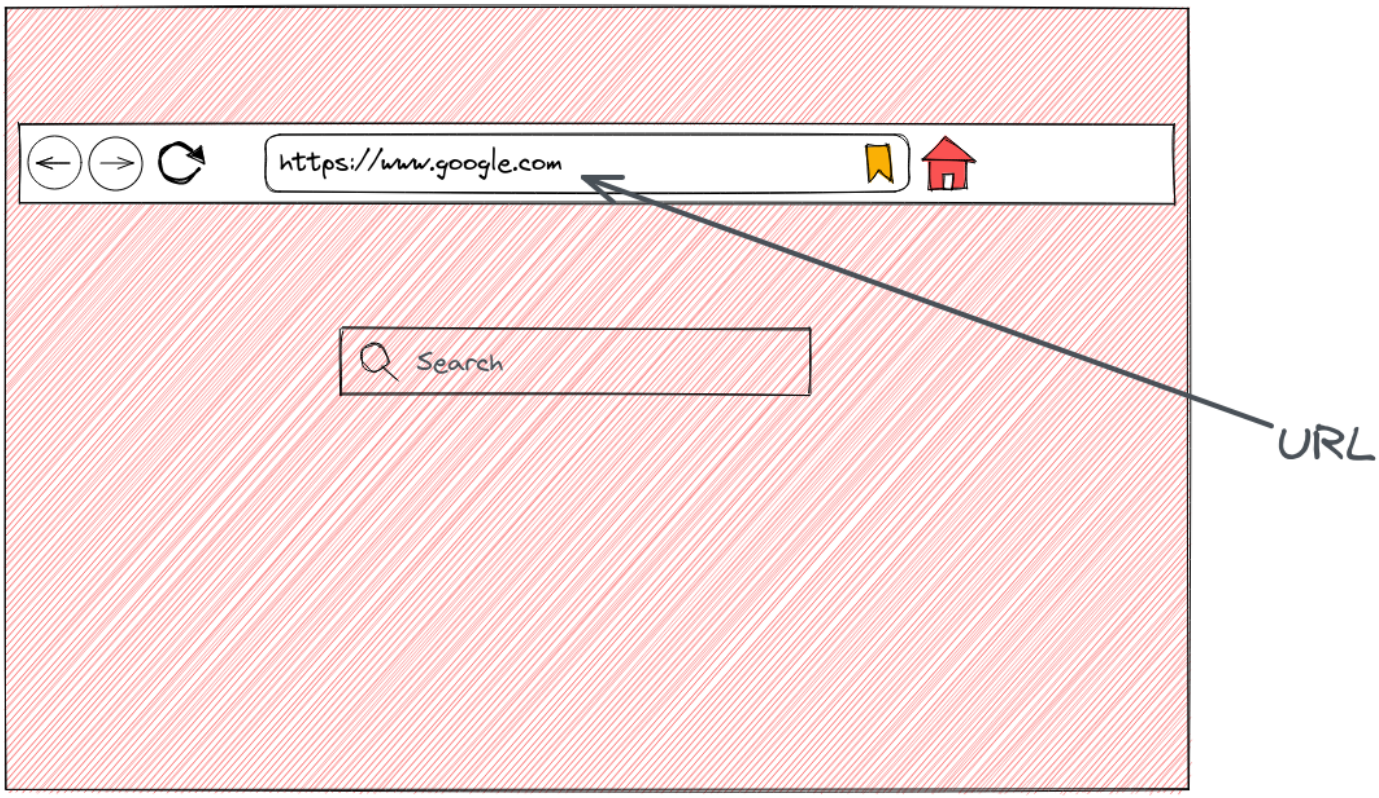
A browser is an application with a graphical user interface which is used to view & navigate web pages. The main function of a browser is to request a resource on behalf of a user & present it to the user by displaying it in the browser window. HTML files are the most commonly requested resource, but it can also be a PDF, image, audio or a video file.

There are many different browsers which are freely available for people to use - some of the widely used ones are Chrome, Firefox, Brave, Safari (iOS users), Tor among others. Though the look & feel of each browser is different, all the browsers have some basic standard features in their user interfaces -



1) Address Bar for entering URI
2) Back & Forward buttons to navigate a website
3) Reload button to reload a webpage & Stop button to stop the loading of a webpage
4) Bookmarking option
5) Home Button to return to a specified home page in the browser

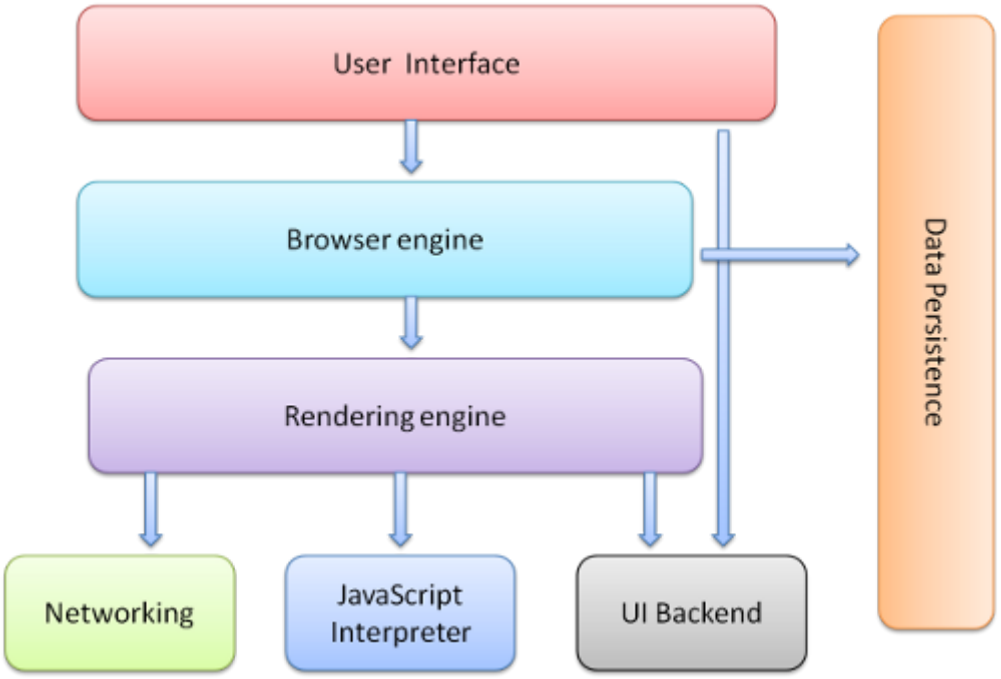
URL is short for Uniform Resource Locator. It is a unique identifier required to locate & fetch any resource on the internet. In the above image is the URL of the most commonly accessed resource or webpage on the internet - Google Homepage.

Let's take the above example in which the user enters the above URL and hits the enter key. The first thing the browser will do is that it'll look for the resource on the internet (since Google Homepage is commonly accessed, it might be stored in the cache).

The browser after coordinating & communicating through various complex processes & protocols of computer networks finally initiates communication with the server. It sends a **request** to the server and if everything goes fine, receives the requested resource as a **response** from the server.

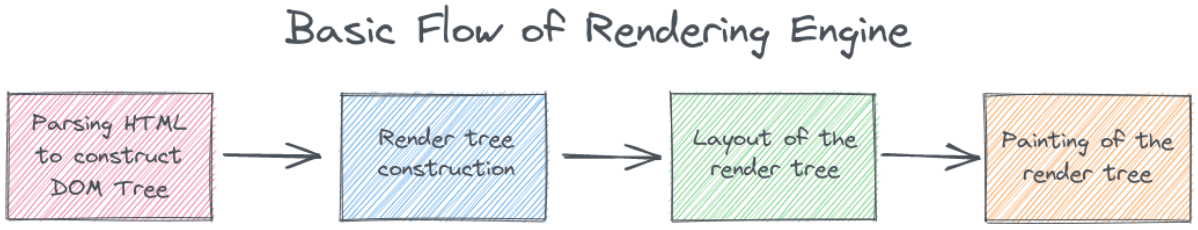
At this point, it is imperative to know a bit about the high level components of a browser and their individual roles:

1. **User interface**: This is the part where the user interacts with the browser. The webpage is not part of the UI.
2. **Browser engine**: Coordinates between UI and the rendering engine.
3. **Rendering engine**: responsible for displaying the requested resource/ content. For example if the requested content is HTML, the rendering engine parses HTML and CSS, and displays the parsed content on the screen.
4. **Networking**: for network calls such as HTTP requests.
5. **UI backend**: uses the Operating System's graphic tools to build the interface.
6. **JavaScript Interpreter**. Used to parse and execute JavaScript code.
7. **Data Storage**. This is a persistence layer to save all sorts of data locally, such as cookies.

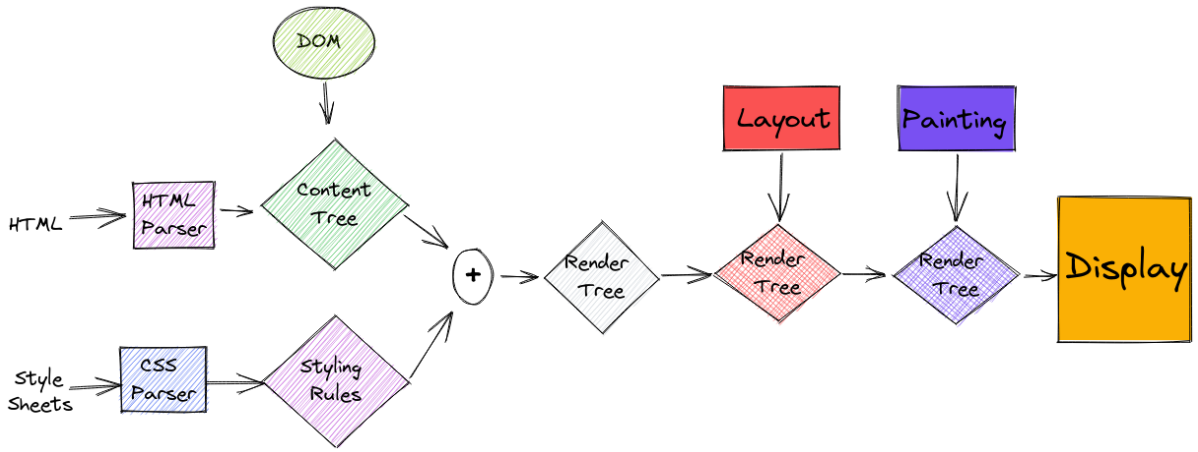


All of the above are responsible for the user experience of modern browsers. Out of those, Rendering Engine is what mainly concerns Frontend Development.

Rendering Engine in Detail



Once the browser receives the content from the internet, the rendering engine starts parsing the HTML document and converts elements to DOM nodes in a tree called the "**content tree**". Simultaneously, the engine also parses the style data using CSS Parser, both in external CSS files and in style elements.



Parsing a document means translating it to a structure the code can use. The result of parsing is usually a tree of nodes that represent the structure of the document. This is called a parse tree or a syntax tree. HTML & CSS Parsers parse HTML & CSS files respectively.

Content Tree & the styling information is then combined to form **Render Tree**. The render tree contains rectangles with visual attributes like color and dimensions. The rectangles are in the right order to be displayed on the screen.

Then the render tree goes through the Layout process in which each node is given the exact coordinates where they should be appearing on the screen. Next is the **Painting** stage in which the render tree is traversed and each node painted using the backend layer.

All these processes might take some time & the rendering engine might display to the user the content which is already processed and not wait for the entire process to be completed. It will not wait until all HTML is parsed before starting to build and layout the render tree. Parts of the content will be parsed and displayed, while the process continues with the rest of the content that keeps coming from the network.

Script Processors & Order of Script Processing

Since Script Processing is not exclusively mentioned in the above flow, I'll shed some light on it as it forms an important process with respect to order in which content is processed and how it'll affect content being displayed.

For eg: there might be some image to be fetched from a URL which might be encountered by the browser while creating the render tree. Of course, it won't be logical for the entire process to be halted while the image is fetched from the network. So there are some optimizations which are done by browsers.

The model of the web is synchronous. By default, parsing of the document halts until the script has been executed. 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.

Speculative parsing done by some browsers

While executing scripts, another thread parses the rest of the document and finds out what other resources need to be loaded from the network and loads them. In this way, resources can be loaded on parallel connections and overall speed is improved. Note: the speculative parser only parses references to external resources like external scripts, style sheets and images: it doesn't modify the DOM tree - that is left to the main parser.

The above process may vary for Style Sheets since they don't change the DOM tree. But in some cases when scripts ask for style information during document parsing stage, it might be a problem if style is not loaded and parsed yet. Firefox blocks all scripts when there is a style sheet that is still being loaded and parsed. WebKit blocks scripts only when they try to access certain style properties that may be affected by unloaded style sheets.