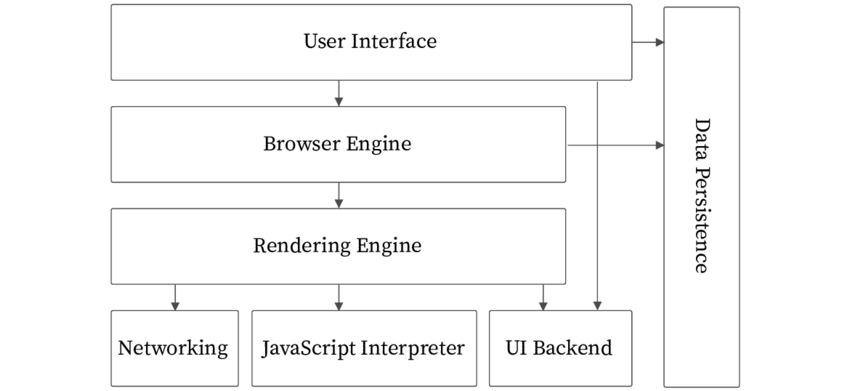


When a user enters a URL in the browser, the browser follows these steps to fetch the desired result:

1. Parsing the URL: The browser parses the URL to extract information such as the protocol (e.g., HTTP or HTTPS), domain name (e.g., https://www.pesto.tech/), and path (e.g., /path/to/resource).
2. DNS resolution: The browser checks its cache to see if it has the IP address of the domain name. If it doesn't have it, it sends a DNS (Domain Name System) query to a DNS server to obtain the IP address.
3. Establishing a TCP connection: The browser establishes a TCP (Transmission Control Protocol) connection with the web server at the IP address obtained in the previous step.
4. Sending an HTTP request: The browser sends an HTTP (Hypertext Transfer Protocol) request to the web server. The request includes information such as the HTTP method (e.g., GET or POST), headers (e.g., Accept-Encoding), and any data that needs to be sent (e.g., form data).
5. Receiving an HTTP response: The web server sends an HTTP response back to the browser. The response includes information such as the HTTP status code (e.g., 200 OK or 404 Not Found), headers (e.g., Content-Type), and any data that needs to be sent (e.g., HTML, CSS, JavaScript, or images).
6. Rendering the web page: The browser receives the response and starts rendering the web page. It parses the HTML, CSS, and JavaScript, and creates a Document Object Model (DOM) tree. It then uses the DOM tree to render the web page on the screen.

**High level components of a browser:**

1. 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.
3. **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.
4. **Networking**: for network calls such as HTTP requests, using different implementations for different platform behind a platform-independent interface.
5. **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.
6. **JavaScript interpreter**. Used to parse and execute JavaScript code.
7. **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.



**Rendering engine and it’s use**: The rendering engine is responsible for parsing HTML, CSS, and JavaScript code and displaying the content of web pages on the screen. It interprets the HTML code and builds the Document Object Model (DOM) tree, which represents the structure of the web page. It then applies the CSS styles to the DOM tree to determine the layout and appearance of the web page. Finally, it executes any JavaScript code embedded in the web page to add interactivity and dynamic behavior.

**Parser and scripts**: Parsing means analyzing and converting a program into internal format so that it can run on a runtime environment. Example: Javascript engine inside the browser.Browser parses the HTML code into a Document object model tree. HTML parsing involves tokenization and tree construction. HTML tokens include start and end tags, as well as attribute names and values. If the document is well-formed, parsing it is straightforward and faster. The parser parses tokenized input into the document, building up the document tree. When the HTML parser finds non-blocking resources, such as an image, the browser will request those resources and continue parsing. Parsing can continue when a CSS file is encountered, but <script> tags—particularly those without an [async](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function) or defer attribute—blocks rendering, and pauses parsing of HTML.

When the browser encounters CSS styles, it parses the text into the CSS Object Model (or CSSOM), a data structure it then uses 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. JavaScript is also downloaded, parsed, and then executed.

# Render-tree Construction, Layout, and Paint :

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.



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 soon),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.

1. For each visible node, find the appropriate matching CSSOM rules and apply them.
2. Emit visible nodes with content and their computed styles
3. 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.**
4. 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 of the device---that's the "layout" stage.
5. 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.
6. Finally, now that we know which nodes are visible, and their computed styles and geometry, we can pass this information to the final stage, which converts each node in the render tree to actual pixels on the screen. This step is often referred to as "painting" or "rasterizing."
7. The "Layout" event captures the render tree construction, position, and size calculation in the Timeline.
8. When layout is complete, the browser issues "Paint Setup" and "Paint" events, which convert the render tree to pixels on the screen.

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

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