**EXERCISE 1.1 HTML:**

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 constructing.

g.Order of script processing

h.Layout and Painting

ANS: When a user enters a URL(Uniform Resource Locator), browser receive this request and search the requested content on the server and server responds with html file. **However, an HTML content is not what we see when we visit a website... we see a web page with colors and backgrounds and animations and pictures. So there's a process that turns the HTML content to a pretty webpage, and that is parsing and rendering**!

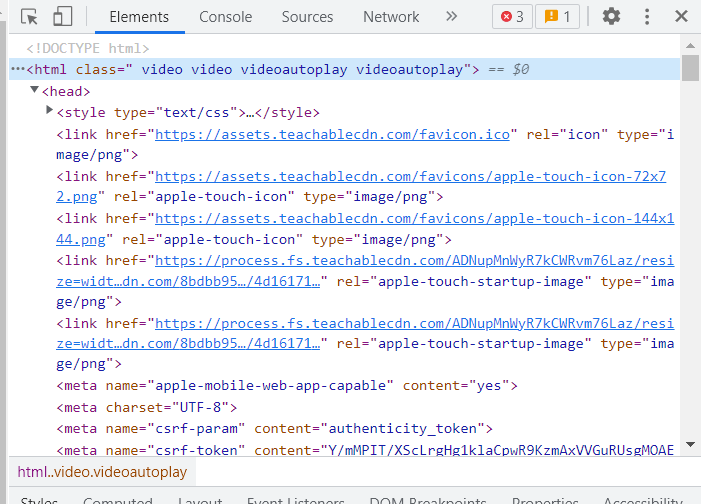
**Diagram:**

request

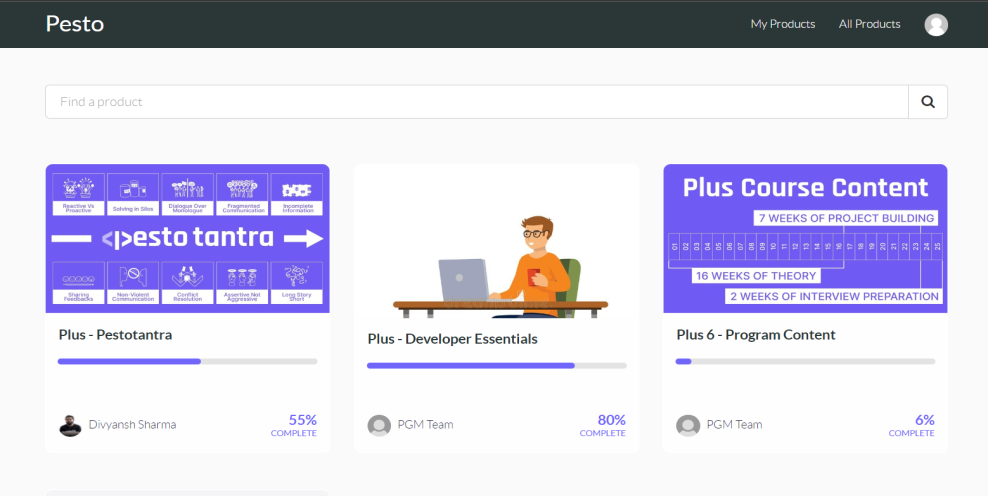
response

SERVER

https://pestotech.teachable.com/courses/enrolled



PARSING/RENDERING



**that’s how browser fetch the desired result.**

1. **Main Functionality of browser:**The main function of the browser is to represent the web resources to the user by requesting it from the server and display it in the browser through the process of parsing or rendering. It takes input from users in the form of URL and using HTTP(Hyper Text Transfer Protocol ) ,finds the desired content on web server.Once found display that desired content on the screen. A **web browser** (also referred to as an **Internet browser** or simply a **browser**) is [application software](https://en.wikipedia.org/wiki/Application_software) for accessing the [World Wide Web](https://en.wikipedia.org/wiki/World_Wide_Web) or a local website. When a [user](https://en.wikipedia.org/wiki/User_(computing)) requests a [web page](https://en.wikipedia.org/wiki/Web_page) from a particular [website](https://en.wikipedia.org/wiki/Website), the web browser retrieves its [files](https://en.wikipedia.org/wiki/Computer_file) from a [web server](https://en.wikipedia.org/wiki/Web_server) and then [graphically](https://en.wikipedia.org/wiki/Computer_graphics) [renders the page](https://en.wikipedia.org/wiki/Browser_engine) on the user's screen. **The way the browser interprets and displays HTML files is specified in the HTML and CSS specifications. These specifications are maintained by the W3C (World Wide Web Consortium) organization, which is the standards organization for the web. For years browsers conformed to only a part of the specifications and developed their own extensions. That caused serious compatibility issues for web authors. Today most of the browsers more or less conform to the specifications.**
2. **Highlevel Components of the browser are :-**
3. **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.
4. **The browser engine**: marshals actions between the UI and the rendering engine.
5. **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.
6. **Networking**: for network calls such as HTTP requests, using different implementations for different platform behind a platform-independent interface.
7. **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.
8. **JavaScript interpreter**. Used to parse and execute JavaScript code.
9. **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.



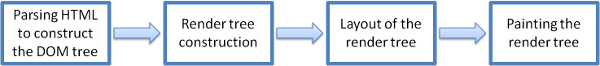
1. **Rendering engine and its use:-**The responsibility of the rendering engine is well… Rendering, that is display of the requested contents on the browser screen.

By default the rendering engine can display HTML and XML documents and images. It can display other types of data via plug-ins or extension; for example, displaying PDF documents using a PDF viewer plug-in. However, in this chapter we will focus on the main use case: displaying HTML and images that are formatted using CSS.

## 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:

Figure : Rendering engine basic flow

1. **Parsing:** Parsing means analyzing and converting a program into an internal format that a runtime environment can actually run, for example the [JavaScript](https://developer.mozilla.org/en-US/docs/Glossary/JavaScript) engine inside browsers.

The [browser parses HTML](https://developer.mozilla.org/en-US/docs/Learn/HTML) into a [DOM](https://developer.mozilla.org/en-US/docs/Glossary/DOM) tree. HTML parsing involves [tokenization](https://developer.mozilla.org/en-US/docs/Web/API/DOMTokenList) and tree construction.

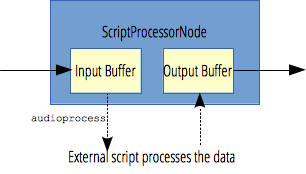
HTML tokens include start and end tags, as well as attribute names and values. If the document is well-formed, parsing it is straightforward and faster. The parser parses tokenized input into the document, building up the document tree.

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.

1. **Script Processor:-**

The Script Processor provides interface that allows the generation, processing, or analyzing of audio using JavaScript.

The ScriptProcessorNode interface is an [AudioNode](https://developer.mozilla.org/en-US/docs/Web/API/AudioNode) audio- processing module that is linked to two buffers, one containing the input audio data, one containing the processed output audio data. An event, implementing the [AudioProcessingEvent](https://developer.mozilla.org/en-US/docs/Web/API/AudioProcessingEvent) interface, is sent to the object each time the input buffer contains new data, and the event handler terminates when it has filled the output buffer with data.

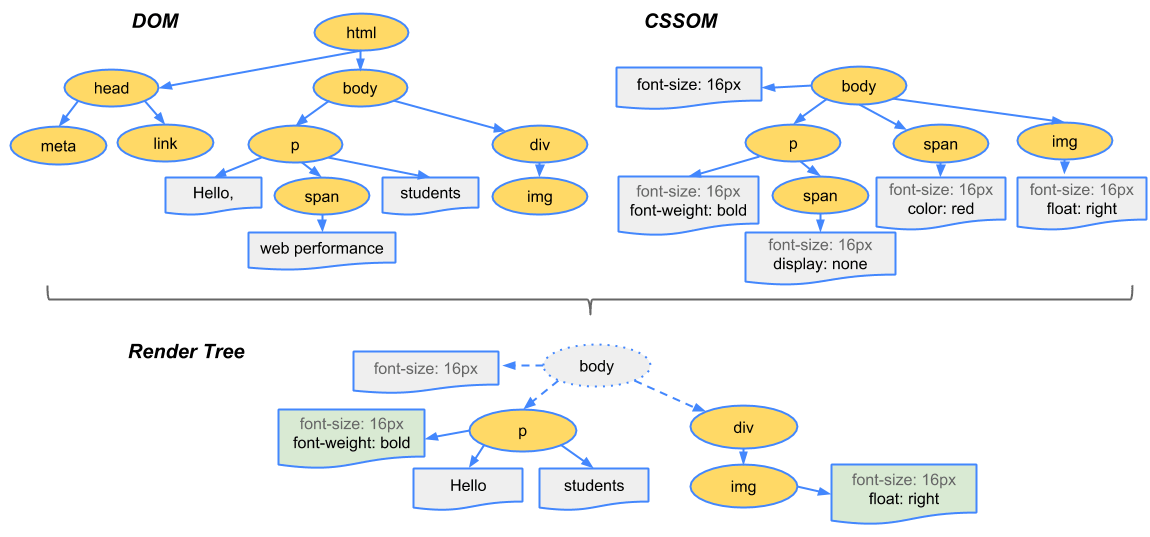


f**) TREE CONSTRUCTING:-**

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.

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.



**G)Order of Script processing:-**

JavaScript performance in the browser is arguably the most important usability issue facing developers. The problem is complex because of the blocking nature of JavaScript, which is to say that nothing else can happen while JavaScript code is being executed. In fact, most browsers use a single process for both user interface (UI) updates and JavaScript execution, so only one can happen at any given moment in time. The longer JavaScript takes to execute, the longer it takes before the browser is free to respond to user input.

Order of execution in JavaScript is dependent on the following components working together to pass and order information.

* The Callstack
* The Event Loop
* The Task Queue
* WebAPIs/External Resources

We can think through the order of execution using the (sometimes headache inducing) example of the asynchronous fetch request.

A very simple fetch request might look something like this

fetch(url)  
.then(res => res.json())  
.then(json => {   
this.setFetchResults(json.items)   
}

You fetch some data, you parse to json, and then you pass that json into another function and do something with it. Fairly straightforward.

Now, if this fetch request happens as part of a larger function, at the beginning of that function, JavaScript will essentially put it to the side, and wait for the response (let’s say in this case from an external API), and continue moving down the function.

Eventually, hopefully, there is either a response from the external API (whether an error, or the data we’re expecting) or a set timeout. Then, this information is passed to the task queue.

As the call stack is cleared, the Event Loop passes information from the task queue back to the call stack to finish executing the function.

**H) LAYOUT AND PAINTING:-**

**Layout:** *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, 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 a box. Different devices and different desktop preferences mean an unlimited number of differing viewport sizes. In this phase, taking the viewport size into consideration, 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, layout generally starts with the body, laying out the dimensions of all the body's descendants, with each element's box model properties, providing placeholder space for replaced elements it doesn't know the dimensions of, such as our image.

**Painting:** 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.

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. There are specific properties and elements that instantiate a layer, including [<video>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/video) and [<canvas>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/canvas), and any element which has the CSS properties of [opacity](https://developer.mozilla.org/en-US/docs/Web/CSS/opacity), a 3D [transform](https://developer.mozilla.org/en-US/docs/Web/CSS/transform), [will-change](https://developer.mozilla.org/en-US/docs/Web/CSS/will-change), and a few others. These nodes will be painted onto their own layer, along with their descendants, unless a descendant necessitates its own 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