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.

1. Browser checks cache for DNS entry to find the corresponding IP address of website.
   It looks for following cache. If not found in one, then continues checking to the next until found.
   - Browser Cache
   - Operating Systems Cache
   - Router Cache
   - ISP Cache
2. If not found in cache, ISP's (Internet Service Provider) DNS server initiates a DNS query to find IP address of server that hosts the domain name.
   The requests are sent using small data packets that contain information content of request and IP address it is destined for.
3. Browser initiates a TCP (Transfer Control Protocol) connection with the server using synchronize(SYN) and acknowledge(ACK) messages.
4. Browser sends an HTTP request to the web server. GET or POST request.
5. Server on the host computer handles that request and sends back a response. It assembles a response in some format like JSON, XML and HTML.
6. Server sends out an HTTP response along with the status of response.
7. Browser displays HTML content

a) What is the main functionalities of browser??

A software application used to access information on the World Wide Web is called a Web Browser. When a user requests some information, the web browser fetches the data from a web server and then displays the webpage on the user's screen.

b) The browser's main components are:

1. **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.
2. **The browser engine**: marshals actions between the UI 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.

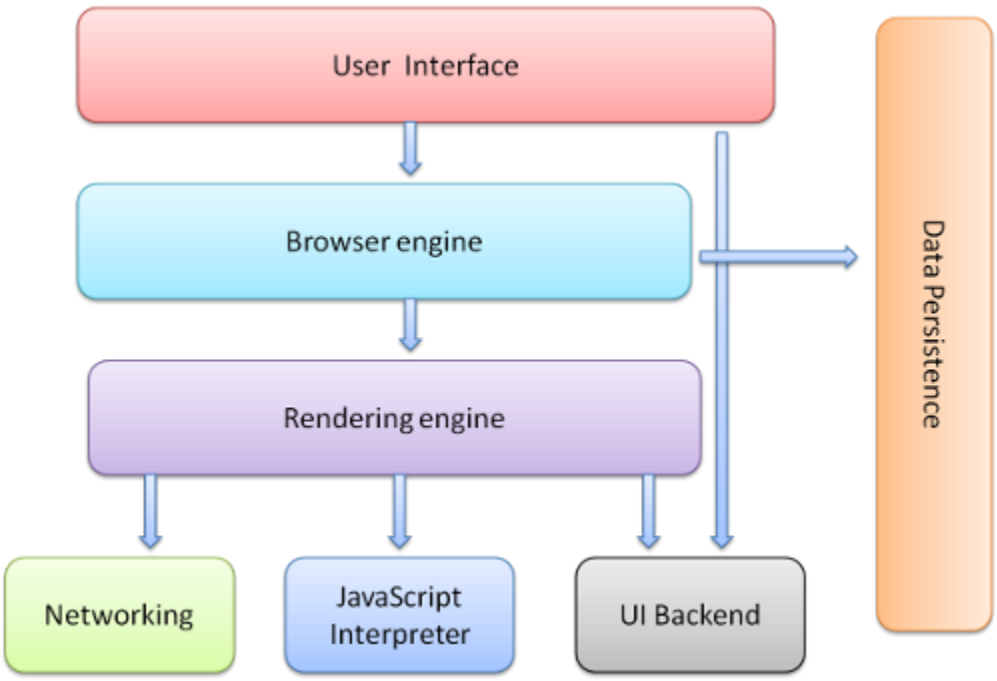

Figure : Browser components

c) Role of Rendering Engine:

Once a user requests a particular document, the rendering engine starts fetching the content of the requested document. This is done via the networking layer. The rendering engine starts receiving the content of that specific document in chunks of 8 KBs from the networking layer. After this, the basic flow of the rendering engine begins.

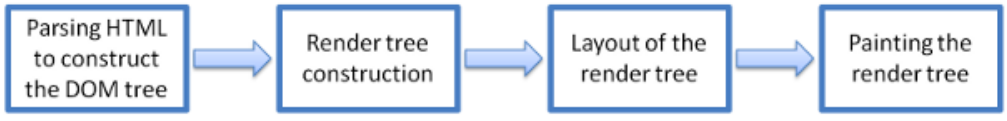

Figure : Rendering engine basic flow

The four basic steps include:

1. The requested HTML page is parsed in chunks, including the external CSS files and in style elements, by the rendering engine. The HTML elements are then converted into DOM nodes to form a **“content tree” or “DOM tree.”**
2. Simultaneously, the browser also creates a **render tree.** This tree includes both the styling information as well as the visual instructions that define the order in which the elements will be displayed. The render tree ensures that the content is displayed in the desired order.
3. Further, the render tree goes through the **layout process.** When a render tree is created, the position or size values are not assigned. The entire process of calculating values for evaluating the desired position is called a layout process. In this process, every node is assigned the exact coordinates. This ensures that every node appears at an accurate position on the screen.
4. The final step is to paint the screen, wherein the render tree is traversed, and the renderer’s **paint()** method is invoked, which paints each node on the screen using the UI backend layer.

d) HTML ,CSS PARSING

The **HTML parser** is a structured markup processing tool. It defines a class called HTMLParser, which is used to parse HTML files. It comes in handy for web crawling.

CSS PARSING

When the browser encounters a CSS stylesheet, be it external or embeded, it needs to parse the text into something it can use for styling the layouts. The data structure that the browser turns the CSS into is called the CSSOM. The DOM and the CSSOM follow similar concepts, in the sense that they are both trees, but they are different data structures. Just like building the DOM out of our HTML, building the CSSOM out of CSS is considered a render-blocking process.

Tokenization & building the CSSOM

Similar to HTML parsing, CSS parsing starts with tokenization. The CSS parser takes the bytes and converts them into characters, then tokens, then nodes and finally

they are linked into the CSSOM. The browser does something called selector machting which means that each set of styles will be matched against all nodes (elements) on the page.

The browser starts with the most general rule applicable to a node (e.g: if a node it's the child of the body element, then all body styles are inherited by that node) and then recursively refines the computed styles by applying more specific rules. This is why we say that the style rules are cascading.

e) Script Processor

The script processor executes Javascript code to process an event. The processor uses a pure Go implementation of ECMAScript 5.1 and has no external dependencies. This can be useful in situations where one of the other processors doesn't provide the functionality you need to filter events.

The processor can be configured by embedding Javascript in your configuration file or by pointing the processor at external file(s).

```
processors:

  - script:

      lang: javascript

      source: >

        function process(event) {

            event.Tag("js");
```

```
}
```

f) Tree Construction

The input to the tree construction stage is a sequence of tokens from the tokenization stage. The tree construction stage is associated with a DOM Document object when a parser is created. The "output" of this stage consists of dynamically modifying or extending that document's DOM tree.

This specification does not define when an interactive user agent has to render the Document so that it is available to the user, or when it has to begin accepting user input.

f) Order of script processing

The scripts in the Process pages are processed in the following order: Pre-Process page and Process page.

...

Script Processing

- Execution Stages.
- Activation.
- Generation.
- Processing.
- Completion.

h) Layout and printing

The step in the critical rendering path is running layout on the render tree to compute the geometry of each node. *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.

The first time the size and position of nodes are determined is called *layout*. Subsequent recalculations of node size and locations are called *reflows*. In our example, suppose the initial layout occurs before the image is returned. Since we didn't declare the size of our image, there will be a reflow once the image size is known.

Paint

The last step in the critical rendering path is painting the individual nodes to the screen, the first occurrence of which is called the 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.

To ensure smooth scrolling and animation, everything occupying the main thread, including calculating styles, along with reflow and paint, must take the browser less than 16.67ms to accomplish. At 2048 X 1536, the iPad has over 3,145,000 pixels to be painted to the screen. That is a lot of pixels that have to be painted very quickly. To ensure repainting can be done even faster than the initial paint, the drawing to the screen is generally broken down into several layers. If this occurs, then compositing is necessary.

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> 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 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.