Week 1 Assignment

What happens when you type a URL into your browser?

1. You type a URL in your browser and press Enter
2. Browser looks up IP address for the domain
3. Browser initiates TCP connection with the server
4. Browser sends the HTTP request to the server
5. Server processes request and sends back a response
6. Browser renders the content

What is the main functionality of browser?

The main functions of web browser is to fetch or retrieve informative resources from World Wide Web to the client/ user on demand, translate those files received from web server and display those content to the user and allow the client /user to access all other relevant resources & information via hyperlinks.

## When the user inputs any URL (uniform resource locator) in the web browser, the user is navigated to that website by the browser quickly. Let us have a look on its processing. When user type any URL, for example https://pesto.com, the prefix of the URL decide how to retrieve it. The URL prefixes that the web browser is not able to handle directly is sent to related application. Like default email app is responsible to handle mailto: URL prefix.

Plugins are available on web browser that supports flash content and java applets to run smoothly in any device.

Web browser allow users to interact with web pages and other dynamic contents via hyperlinks that provides navigation facility i.e. to go to different locations by clicking on links that makes internet surfing easy.

Web browser provide [security of data](https://msatechnosoft.in/blog/tech-blogs/ip-security-add-authentication-packets-ip-level) and resources over the web through its secure methodologies.

High Level Components of a browser ?

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.



## Rendering Engines and its use

The responsibility of the rendering engine 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.

## Rendering engines

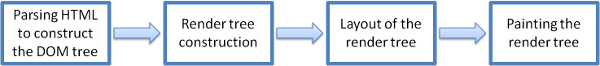
Different browsers use different rendering engines: Internet Explorer uses Trident, Firefox uses Gecko, Safari uses WebKit. Chrome and Opera (from version 15) use Blink, a fork of WebKit.

WebKit is an open source rendering engine which started as an engine for the Linux platform and was modified by Apple to support Mac and Windows.

## 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

The rendering engine will start parsing the HTML document and convert elements to [DOM](https://web.dev/howbrowserswork/#Dom) nodes in a tree called the "content tree". The engine will parse the style data, both in external CSS files and in style elements. Styling information together with visual instructions in the HTML will be used to create another tree: the [render tree](https://web.dev/howbrowserswork/#Render_tree_construction).

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.

After the construction of the render tree it goes through a "[layout](https://web.dev/howbrowserswork/#layout)" process. This means giving each node the exact coordinates where it should appear on the screen. The next stage is [painting](https://web.dev/howbrowserswork/#Painting) - the render tree will be traversed and each node will be painted using the UI backend layer.

It's important to understand that this is a gradual process. For better user experience, the rendering engine will try to display contents on the screen as soon as possible. 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 contents that keeps coming from the network.

Parsing

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 Parser

## The job of the HTML parser is to parse the HTML markup into a parse tree.

## The HTML grammar definition

### The vocabulary and syntax of HTML are defined in specifications created by the W3C organization.

## Not a context free grammar

### As we have seen in the parsing introduction, grammar syntax can be defined formally using formats like BNF.

Unfortunately all the conventional parser topics do not apply to HTML (I didn't bring them up just for fun - they will be used in parsing CSS and JavaScript). HTML cannot easily be defined by a context free grammar that parsers need.

There is a formal format for defining HTML - DTD (Document Type Definition) - but it is not a context free grammar.

This appears strange at first sight; HTML is rather close to XML. There are lots of available XML parsers. There is an XML variation of HTML - XHTML - so what's the big difference?

The difference is that the HTML approach is more "forgiving": it lets you omit certain tags (which are then added implicitly), or sometimes omit start or end tags, and so on. On the whole it's a "soft" syntax, as opposed to XML's stiff and demanding syntax.

This seemingly small detail makes a world of a difference. On one hand this is the main reason why HTML is so popular: it forgives your mistakes and makes life easy for the web author. On the other hand, it makes it difficult to write a formal grammar. So to summarize, HTML cannot be parsed easily by conventional parsers, since its grammar is not context free. HTML cannot be parsed by XML parsers.

## DOM

The output tree (the "parse tree") is a tree of DOM element and attribute nodes. DOM is short for Document Object Model. It is the object presentation of the HTML document and the interface of HTML elements to the outside world like JavaScript.

The root of the tree is the "[Document](http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/level-one-core.html#i-Document)" object.

<html>  
 <body>  
 <p>  
 Hello World  
 </p>  
 <div> <img src="example.png"/></div>  
 </body>  
</html>

This markup would be translated to the following DOM tree:

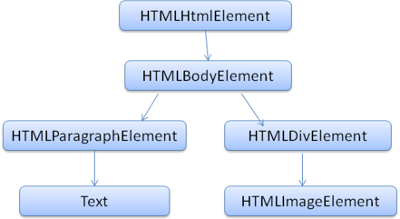


Figure : DOM tree of the example markup

Like HTML, DOM is specified by the W3C organization. It is a generic specification for manipulating documents. A specific module describes HTML specific elements.

When I say the tree contains DOM nodes, I mean the tree is constructed of elements that implement one of the DOM interfaces. Browsers use concrete implementations that have other attributes used by the browser internally.

## The parsing algorithm

## As we saw in the previous sections, HTML cannot be parsed using the regular top down or bottom up parsers.

The reasons are:

1. The forgiving nature of the language.

The fact that browsers have traditional error tolerance to support well known cases of invalid HTML.

1. The parsing process is reentrant. For other languages, the source doesn't change during parsing, but in HTML, dynamic code (such as script elements containing document.write() calls) can add extra tokens, so the parsing process actually modifies the input.

Unable to use the regular parsing techniques, browsers create custom parsers for parsing HTML.

## Tree construction algorithm

When the parser is created the Document object is created. During the tree construction stage the DOM tree with the Document in its root will be modified and elements will be added to it. Each node emitted by the tokenizer will be processed by the tree constructor. For each token the specification defines which DOM element is relevant to it and will be created for this token. The element is added to the DOM tree, and also the stack of open elements. This stack is used to correct nesting mismatches and unclosed tags. The algorithm is also described as a state machine. The states are called "insertion modes".

Let's see the tree construction process for the example input:

<html>  
 <body>  
 Hello world  
 </body>  
</html>

The input to the tree construction stage is a sequence of tokens from the tokenization stage. The first mode is the **"initial mode"**. Receiving the "html" token will cause a move to the **"before html"** mode and a reprocessing of the token in that mode. This will cause creation of the HTMLHtmlElement element, which will be appended to the root Document object.

The state will be changed to **"before head"**. The "body" token is then received. An HTMLHeadElement will be created implicitly although we don't have a "head" token and it will be added to the tree.

We now move to the **"in head"** mode and then to **"after head"**. The body token is reprocessed, an HTMLBodyElement is created and inserted and the mode is transferred to **"in body"**.

The character tokens of the "Hello world" string are now received. The first one will cause creation and insertion of a "Text" node and the other characters will be appended to that node.

The receiving of the body end token will cause a transfer to **"after body"** mode. We will now receive the html end tag which will move us to **"after after body"** mode. Receiving the end of file token will end the parsing.

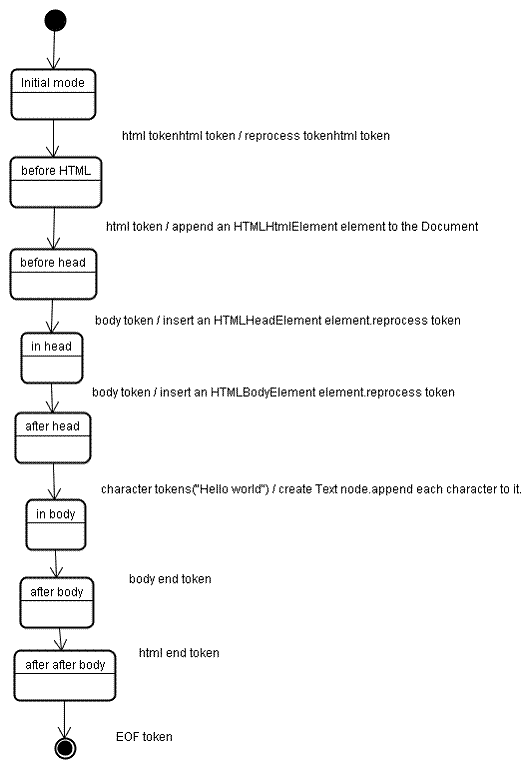


Figure : tree construction of example html

## The order of processing scripts and style sheets

## Scripts

The model of the web is synchronous. Authors expect scripts to be parsed and executed immediately when the parser reaches a <script> tag. The parsing of the document halts until the script has been executed. If the script is external then the resource must first be fetched from the network - this is also done synchronously, and parsing halts until the resource is fetched. 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

## Both WebKit and Firefox do this optimization. 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.

### Style sheets

### Style sheets on the other hand have a different model. Conceptually it seems that since style sheets don't change the DOM tree, there is no reason to wait for them and stop the document parsing. There is an issue, though, of scripts asking for style information during the document parsing stage. If the style is not loaded and parsed yet, the script will get wrong answers and apparently this caused lots of problems. It seems to be an edge case but is quite common. 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.

### Layout

### When the renderer is created and added to the tree, it does not have a position and size. Calculating these values is called layout or reflow.

HTML uses a flow based layout model, meaning that most of the time it is possible to compute the geometry in a single pass. Elements later "in the flow" typically do not affect the geometry of elements that are earlier "in the flow", so layout can proceed left-to-right, top-to-bottom through the document. There are exceptions: for example, HTML tables may require more than one pass.

The coordinate system is relative to the root frame. Top and left coordinates are used.

Layout is a recursive process. It begins at the root renderer, which corresponds to the <html> element of the HTML document. Layout continues recursively through some or all of the frame hierarchy, computing geometric information for each renderer that requires it.

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.

## 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 infrastructure component.