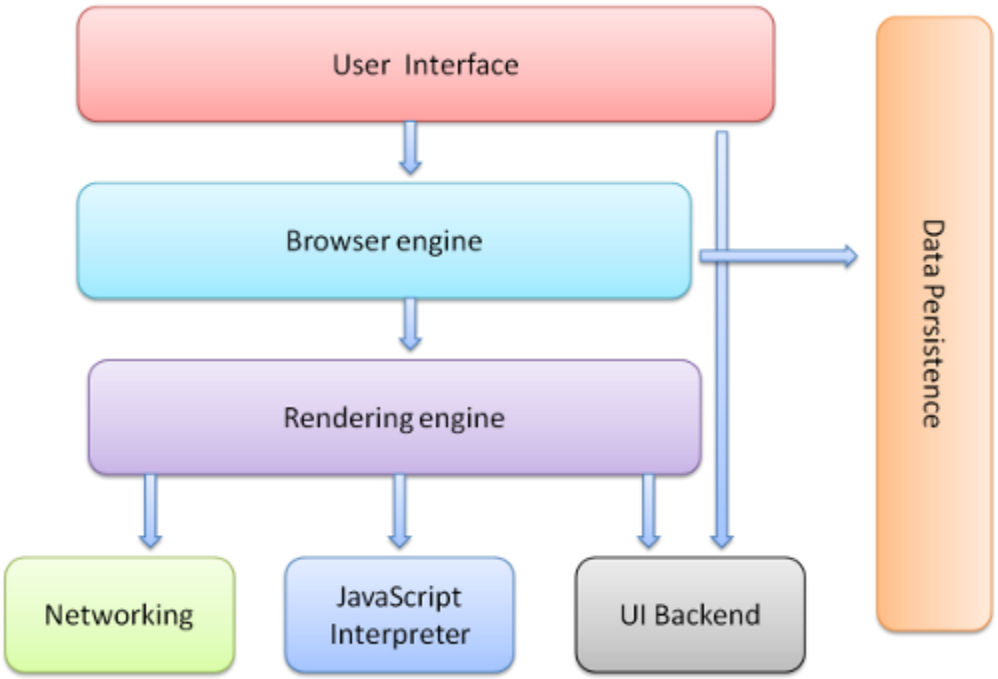
There are several perspectives taken while understanding the browser internal working. Block level or high level view or more technical and introspective view of it.

`Perspective I`



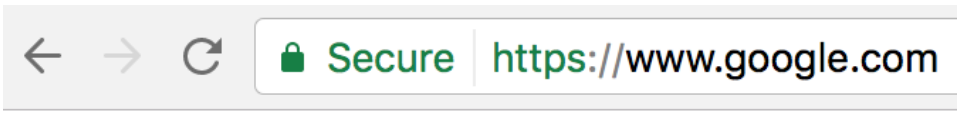
Browser Components

Browsers have a set of components with a particular work flow. Let's have a look at each component in detail, one by one.

User Interface

The address bar is a good example. It interfaces with the user to insert a URI and interacts with various components to render a web page.

There are plenty of user interfaces for various needs, like Back and Forward Buttons, Bookmarks, Reload, and Stop and Home Buttons, which you see below.



Address Bar in Chrome

The interface layer communicates with the data layer to retrieve data. The interface layer also communicates with the UI Backend to draw widgets as per requests by the HTTP Response body (our HTML source code). The browser engine communicates between the UI and the rendering/layout engine.

Browser Engine

It's a bridge between the user interface and the rendering engine. It provides methods to initiate the loading of a URL and other actions like reload, back, and forward.

Layout/Rendering Engine

It's able to render the content of a given URL in the browser screen and interprets the HTML, XML, and CSS. It is single threaded. By default, it displays data according to your specified content type (MIME). For example: HTML, Images, XML, CSS, JSON, PDF, etc.

A key operation of the rendering engine is the HTML Parser. Each browser uses various engines. Chrome and Opera use Blink, Firefox uses Gecko, IE Edge uses EdgeHTML, Internet Explorer uses Trident, and Safari uses WebKit.

The Flow

1. Parsing an HTML document with the HTML Parser converts elements to Node and creates a Content Tree.
2. Parsing styles for code/documents with the CSS Parser and creates a Render Tree.
3. The Render Tree goes through the Layout Process. An element's node gets position coordinates.
4. The Render Tree will be traversed and each node will be painted using the UI Backend Layer.

A Few Other Stages

Width calculation - It's calculated using the container width attribute/style attribute.

Line Breaking - While a user is scrolling, the layout parent creates the extra renderers and calls the layout on them.

Painting

1. A render tree is traversed and the renderer's "paint()" method is called.
2. The paint method is called to display content on the screen.
3. It uses the UI infrastructure component.
4. Painting order (background color, background image, border, children, outline).

Network

1. Networking handles all aspects of Internet communication and handles URLs to use with HTTP, FTP.

2. Implements a cache of retrieved documents to minimize network traffic.

JavaScript Interpreter — Scripting Engines

JavaScript Interpreter executes the JS code and sends the results to the rendering engine.

Each browser uses various scripting engines, for example, Chrome uses V8, Firefox uses Spider Monkey, IE Edge uses Chakra (JavaScript Engine), Internet Explorer uses Chakra (JScript Engine), Safari uses Nitro (JavaScript Core)

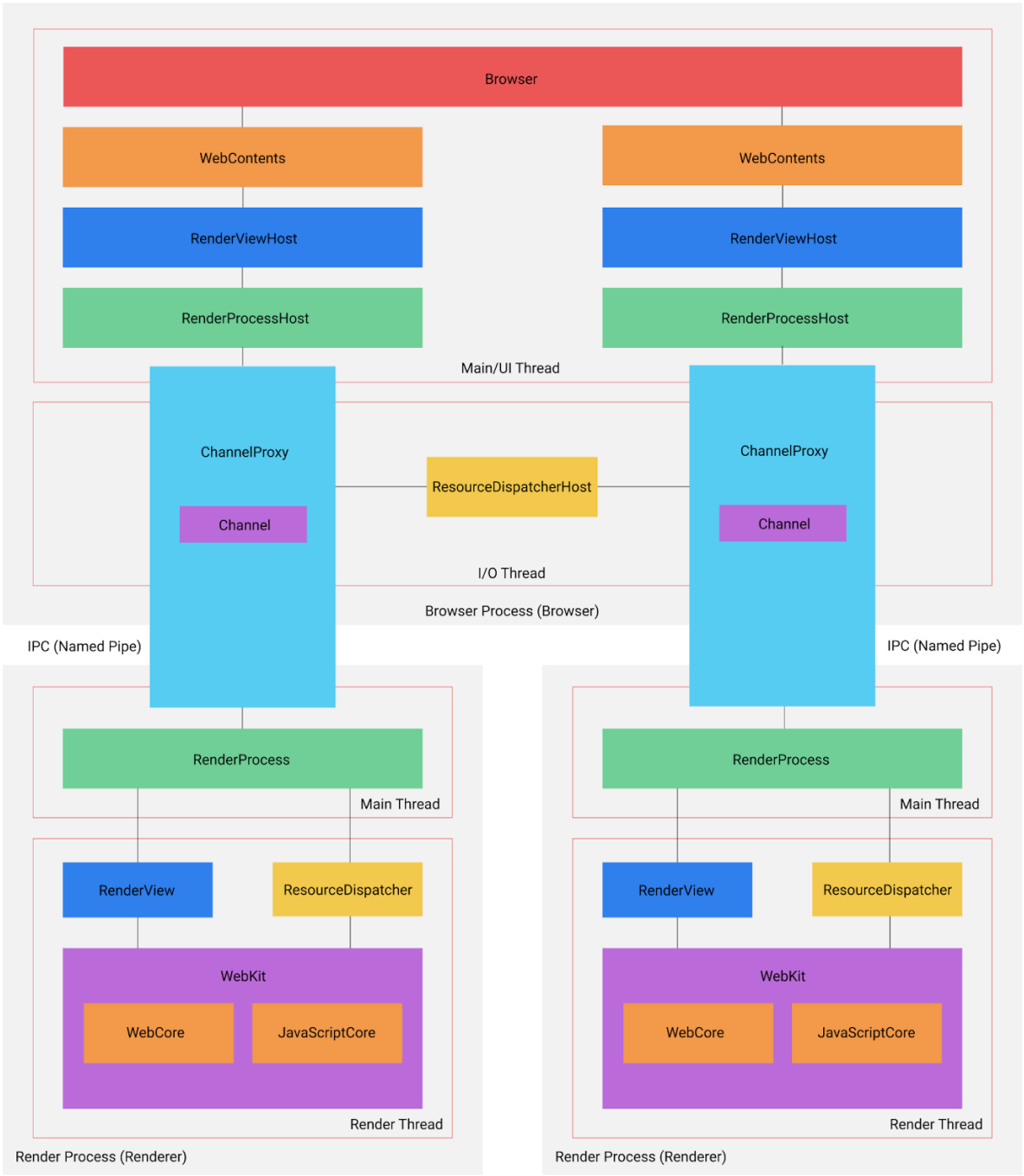
UI Backend

Backend helps to draw widgets like a select box, an input box, a check box, etc.

Data Persistence

This layer is persistent which helps the browser to store data (like Cookies, Local Storage, Session Storage, IndexedDB, WebSQL, and FileSystem) locally.

Perspective II



There are two primary types of processes in Chromium multi-process architecture: **the browser process** and **the render process**. Generally speaking, there can be only one browser process and multiple render processes.

The Browser Process

A] What is the main functionality of the browser?

The Browser is main process responsible for managing all render processes and displaying UI. It maintains two threads: main/UI thread and I/O thread. Main/UI thread is largely responsible for rendering web pages on screen. I/O thread takes care of IPC communication between the browser process and render processes, and any network communication is also handled in this thread. Each thread runs several different objects. In the section below, I will explain details of all major objects categorized by the thread they belong to.

B] What is the main functionality of the browser?

Objects in the main/UI thread

RenderProcessHost: The browser process might have more than one RenderProcessHost. Each one connects to a render process with one-to-one relationship. The primary responsibility of RenderProcessHost is to dispatch view-specific messages to RenderViewHost and digest the rest non-view-specific messages. This object is also needed to uniquely identify a RenderView along with a RenderView Id because a render process might have more than one RenderView and the Id is only unique inside a render process but not within the browser process.

RenderViewHost: as I mentioned above, RenderViewHost receives view-specific messages from RenderProcessHost. The relationship between RenderProcessHost and RenderViewHost is one-to-many because a render process might generate more than one RenderView (e.g. a web page and a popup in the same tab) and RenderProcessHost needs to pass that information to corresponding

RenderViewHost. This object is responsible for navigational commands, receiving input events, and painting web pages.

WebContents: this object represents a tab with a web page in it. It is responsible for displaying a web page in a rectangular view.

Browser: this object represents a browser window and might contain multiple WebContents.

Objects in the I/O thread

Channel: this object defines methods for communicating across pipes. Its main responsibility is to communicate with render processes over IPC.

ResourceDispatcherHost: This object is responsible for sending network request to the internet. It acts as a gate to the internet. If a render process wants to send a request to the wild, it needs to send this request to RenderProcessHost first. Then RenderProcessHost forwards this request to ResourceDispatcherHost to actually send the request.

Objects in the main thread

RenderProcess: The render process maintains two threads: **main thread** and **render thread**. Main thread has only one object RenderProcess whose solely responsibility is to facilitate cross-thread communication between render thread in which RenderView lives and I/O thread of the browser process. Render thread, on the other hand, takes care of generating web pages. In the section below, I will illustrate major objects in the render process.

Objects in the render thread

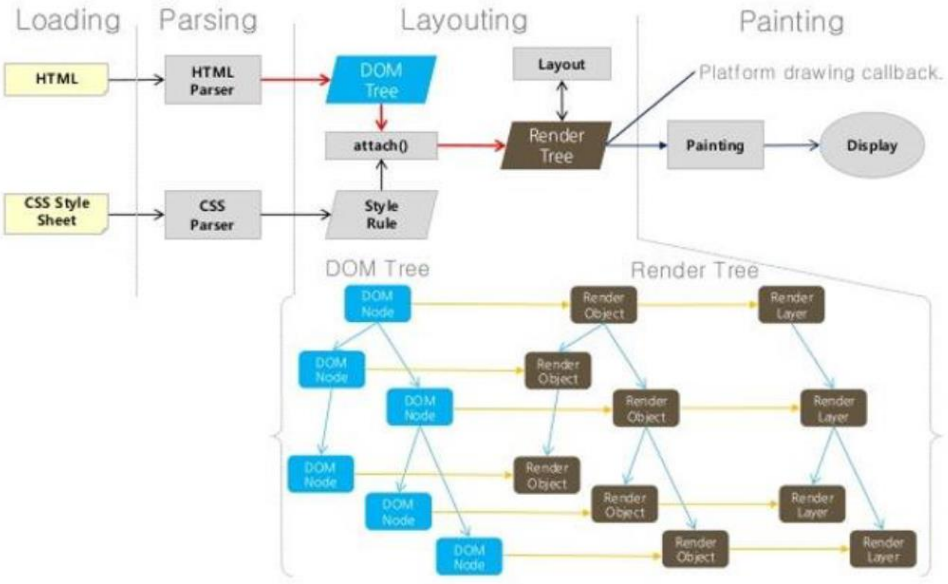
ResourceDispatcher: sometimes a web page needs to make request to server to fetch content. The render process does not have access to the internet. It has to rely on ResourceDispatcher to forward a request to the browser via IPC.

Webkit: this is the rendering engine used in Chromium. It is used for constructing DOM and laying out web pages. Webkit consists of two primary components: WebCore which contains core layout functionality and JavaScriptCore where JavaScript interpreter V8 lives.

C] What is the main functionality of the browser?

The rendering engine refers to a program that constructs DOM, executes JavaScript, and layout out web pages, e.g. Webkit, Gecko, Trident. A rendering engine consists of two primary components: WebCore which contains core layout functionality and JavaScriptCore where JavaScript interpreter V8 lives.

D] Parsers (HTML, CSS, etc.)



Parsing is the process of reading HTML content and constructing a DOM tree from it. Hence the process is also called DOM parsing and the program that does that is called the DOM parser. The DOM parsing normally happens on the main thread. So if the main JavaScript execution thread is busy, DOM parsing will not progress until the thread is free.

E] Script Processor

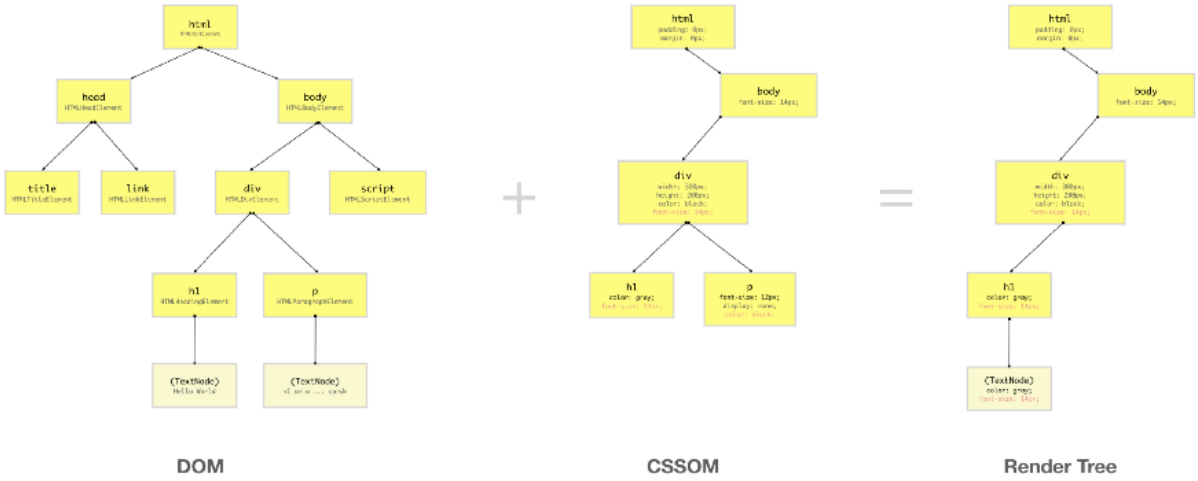
A process scans the web document, identifies scripts, and initiates the downloading of the scripts. As the scripts are downloaded, an HTML parser generates an identifier for each script and the sends the scripts and associated identifiers to a script engine. The script engine parses, analyses, compiles, and otherwise prepares the scripts for execution in an order that may be different than the execution order of the scripts.

F] Tree Construction

When the browser reads HTML code, whenever it encounters an HTML element like html, body, div etc., it creates a JavaScript object called a Node. Eventually, all HTML elements will be converted to JavaScript objects. Since every HTML element has different properties, the Node object will be created from different classes (constructor functions). For example, the Node object for the div element is created from HTMLDivElement which inherits Node class.

After the browser has created Nodes from the HTML document, it has to create a tree-like structure of these node objects. Since our HTML elements in the HTML file

are nested inside each other, the browser needs to replicate that but using Node objects it has previously created. This will help the browser efficiently render and manage the webpage throughout its lifecycle.



DOM tree starts from the topmost element which is html element and branches out as per the occurrence and nesting of HTML elements in the document. Whenever an HTML element is found, it creates a DOM node (Node) object from its respective class (constructor function). A DOM node doesn't always have to be an HTML element. When the browser creates a DOM tree, it also saves things like comments, attributes, text as separate nodes in the tree. But for the simplicity, we will just consider DOM nodes for HTML elements

---

G] Order of script processing

The browser will execute the scripts in the order it finds them. If you call an external script, it will block the page until the script has been loaded and executed. Dynamically added scripts are executed as soon as they are appended to the document.

	Loading priority (network/Blink)	Execution priority	Where should this be used?
`<script>` in `<head>`	Medium/High	VeryHigh - Blocks parser	◦ Scripts that affect layout of First Meaningful Paint (FMP) / First Contentful Paint (FCP) content ◦ Scripts that must be run before other scripts Examples: ◦ Framework runtime (if not static-rendered) ◦ Polyfills ◦ A/B testing that affects DOM structure of the entire page
`<link rel=preload>` + `<script async>` hack or `<script type=module async>`	Medium/High	High - Interrupts parser	◦ Scripts that generate critical content (needed for FMP) ◦ But shouldn't affect above-the-fold layout of the page ◦ Scripts that trigger network fetches of dynamically inserted content ◦ Scripts that need to execute as soon as their imports are fetched, use `<script async type=module>` Examples: ◦ Draw something on `<canvas>`
`<script async>`	Lowest/Low	High - Interrupts parser	Be careful when considering `<script async>`. Today it is often used to indicate non-critical scripts, but is inconsistent in being loaded at low priority and executed at high priority.
`<script defer>`	Lowest/Low	VeryLow - Runs after `<script>`s at end of `<body>`	◦ Scripts that generate non-critical content ◦ Scripts that provide key interactive features that >50% of page view sessions would use Examples: ◦ Ad frameworks ◦ Framework runtime (if client-side or server rendered)
`<script>` at the end of `<body>`	Medium/High	Low - Waits parser end	Be careful when using `<script>` at the end of `<body>` when you think they are low priority. These scripts are scheduled at Medium/High priority.
`<script defer>` at the end of `<body>`	Lowest/Low - end of the queue	VeryLow - Runs after `<script>`s at end of `<body>`	◦ Scripts that provide interactive features that may be used occasionally Examples: ◦ Load "Related articles" ◦ "Give feedback" feature
`<link rel=prefetch>` + `<script>` in a next-page navigation	Idle / Lowest	Depends on when and how the script is consumed.	Scripts very likely to provide important functionality to a next-page navigation. Examples: ◦ JavaScript bundle for a future route

---

H] Layout and Painting

Layout operation

First browser creates the layout of each individual Render-Tree node. The layout consists of the size of each node in pixels and where (position) it will be printed on the screen. This process is called layout since the browser is calculating the layout

information of each node. This process is also called reflow or browser reflow and it can also occur when you scroll, resize the window or manipulate DOM elements.

Paint operation

Until now we have a list of geometries that need to be printed on the screen. Since elements (or a sub-tree) in the Render-Tree can overlap each other and they can have CSS properties that make them frequently change the look, position, or geometry (such as animations), the browser creates a layer for it. Creating layers helps the browser efficiently perform painting operations throughout the lifecycle of a web page such as while scrolling or resizing the browser window. Having layers also help the browser correctly draw elements in the stacking order (along the z-axis) as they were intended by the developer. Now that we have layers, we can combine them and draw them on the screen. But the browser does not draw all the layers in a single go. Each layer is drawn separately first.

Information Flow

How does information flow move through various components and connectors? To answer this question, I am going to take a life of a “mouse click” message from Chromium official documentation as an example. When a mouse click event is received by main/UI thread of the browser process, it sends to RenderViewHost, which in turn tells RenderProcessHost to send the message to ChannelProxy. Then, ChannelProxy will proxy the message to I/O thread of the browser process and send it to the render process via IPC pipe. In the render process, the message is received by RenderView, which in turns hand this message to WebKit. Finally, WebKit JavaScriptCore will trigger appropriate callback function that handles this mouse click event.