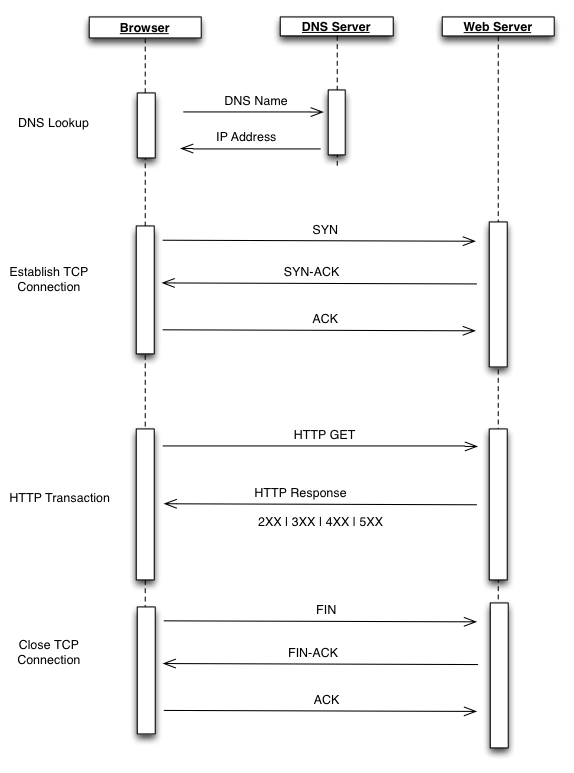
When requesting a web page with a browser the browser creates a thread to handle the request and initiates a DNS lookup at the remote DNS Server. This allows the browser to get the IP address for the URL entered.

Next the browser negotiates a TCP three-way handshake with the remote web server to establish a TCP/IP connection. This handshake consists of a synchronize (SYN), synchronize-acknowledge (SYN-ACK), and acknowledge (ACK) message to be passed between the browser and the remote server.

Once the TCP connection has been established the browser sends an HTTP GET request over the connection to the web server. The web server finds the resource and returns it in an HTTP Response, the status of which is 200 to indicate a good response. If the server cannot find the resource or generates an error when trying to interpret it, or if it is redirected the status of the HTTP Response will reflect these as well. The full list of status codes can be found at http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html but the most common ones are:

* 200 indicates a successful response from the server
* 301 indicates a permanent redirection
* 302 indicates a temporary redirection
* 403 forbidden request
* 404 means that the server could not find the resource requested
* 500 means that there was an error when trying to fulfill the request
* 503 service unavailable
* 504 gateway timeout

See Figure 2.1 for a sequence diagram of this transaction.



Keep in mind that one of these transactions is necessary to not just serve up a single HTML page, but your browser needs to make an HTTP request for each asset that the page links to – all of the images, linked CSS and JavaScript files, and any other type of external asset (though the browser can reuse the TCP connection for each subsequent HTTP request as long as it is connecting to the same origin).

Once the browser has the HTML for the page it begins to parse and render the content.

The browser uses its Rendering engine to parse and render the content. Modern browser architecture consists of several interacting modules:

* The UI Layer that draws the interface or GUI for the browser; think the location bar, the refresh button, basically and UI that is native to the browser.
* The Network Layer that handles network connections, things like establishing TCP connections, handling the HTTP round trips. The Network Layer handles downloading the content and passing it to the Rendering Engine.
* The Rendering Engine handles painting the content to the screen. Browser makers brand and license out their Render and JavaScript engines, so you’ve probably heard the product names for the more popular render engines already. Arguably the most popular Render Engine is WebKit, which is used in Chrome (as a fork named Blink), Safari, and Opera, among many others. When the Render engine encounters JavaScript, it hands it off to the JavaScript interpreter.
* The JavaScript Engine handles parsing and execution of JavaScript. Just like the Render engine, browser makers brand their JavaScript Engines for licensing, and you most likely have heard of them. Some popular JavaScript Engines are Google’s V8 which is used in Chrome, Chromium, as well as the engine that powers Node.js.

See Figure 2.2 for a visualization of this architecture.

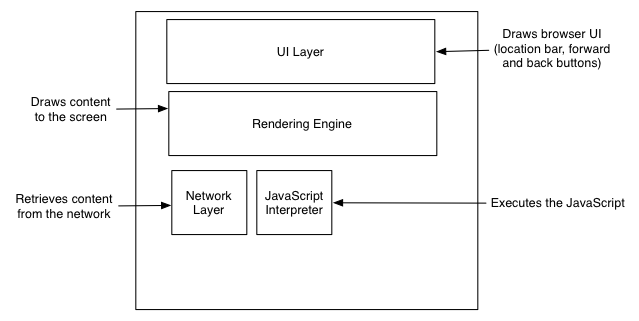


Figure 2.2 Modern browser architecture split into module components

Picture a use case where a user types a URL into the location bar of the browser. The UI layer passes this request to the Network layer, which then establishes the connection and downloads the initial page. As the packets containing chunks of HTML arrive they are passed to the render engine. The render layer assembles the HTML as raw text and begins to perform lexical analysis – or parsing – of the characters in the text. The characters are compared to a rule set – the DTD that we specify in our HTML document – and converted to tokens based on the rule set. The DTD specifies the tags that make up the version of the language that we will use. The tokens are just the characters broken into meaningful segments.

For example the Network Layer may return the below string.

<!DOCTYPE html><html><head><meta charset="UTF-8"/>

This string would get tokenized into meaningful chunks.

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8"/>

The Render Engine then takes the tokens and converts them to DOM elements. The DOM elements are laid out in a render tree that the Render Engine then iterates over. In the first iteration the Render Engine lays out the positioning of the DOM elements, and in the next it paints them to the screen.

If the render layer identifies a script tag during the parsing and tokenization phase it pauses and evaluates what to do next. If the script tag points to an external JavaScript file then parsing is paused, and the network layer is used to download this file before initializing the JavaScript engine to interpret and execute the JavaScript. If the script tag contains inline JavaScript then the rendering is paused and the JavaScript engine is initialized and the JavaScript is interpreted and executed. Once execution is complete then parsing resumes. This is an important nuance that has implications not just on when DOM elements are available to JavaScript (our code might be trying to access an element on the page that has not yet been parsed and tokenized, let alone rendered) but as well as performance (do we want to block the parsing of the page until this code is downloaded and run, or can the page be functional if we show the content first then load the page).

See Figure 2.3 for this workflow.

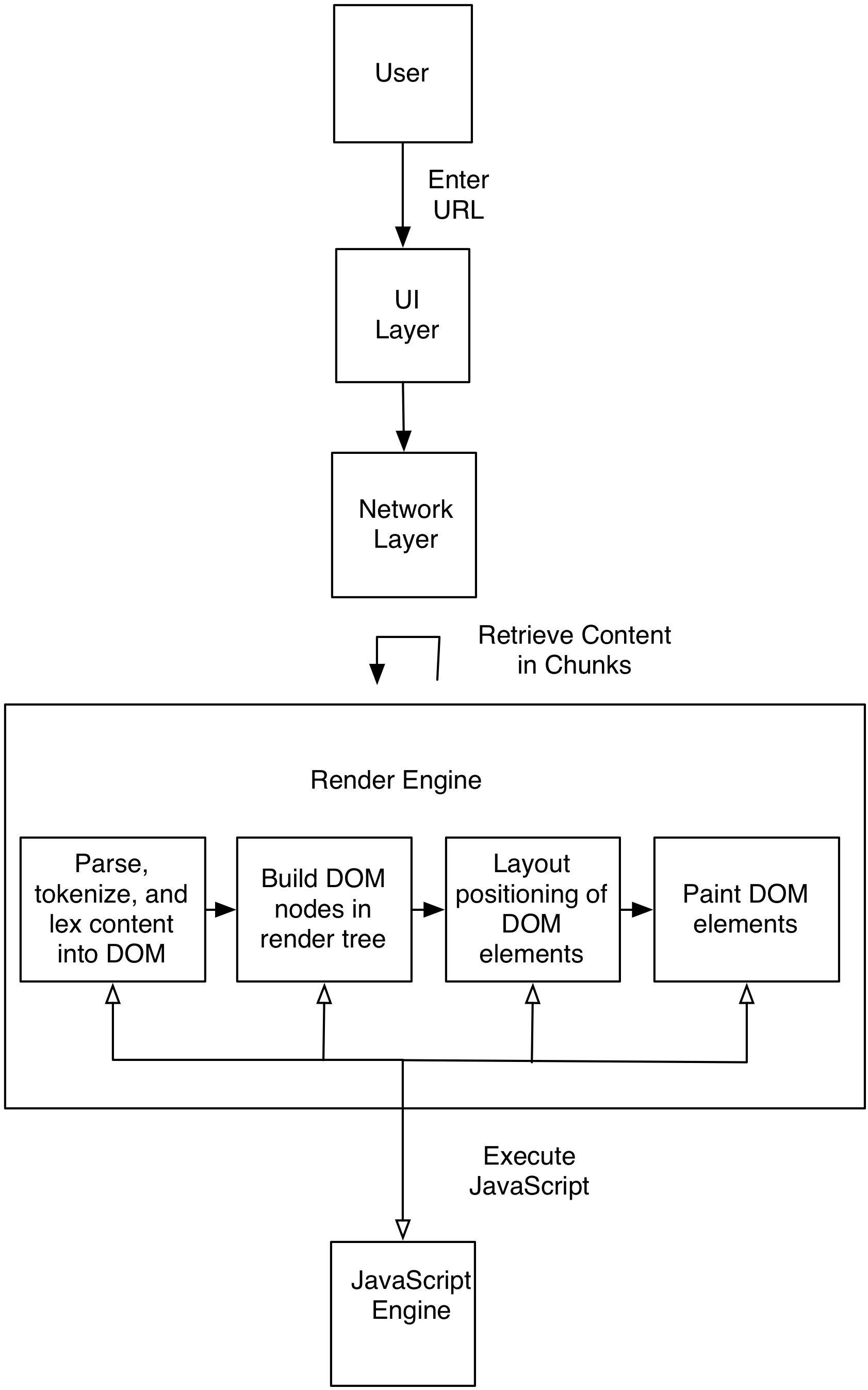


Figure 2.3 Workflow describing the loading and rendering of content in the browser