# Google Chrome's Conceptual Architecture

19 October 2018

Course Code: CISC 322
Instructor: Dr. Ahmed Hassan
TA: Hongkai Chen

Cameron Raymond – 15cjkr@queensu.ca
Brendan Russell – 15br18@queensu.ca
Brenden Forbes – 15bpf@queens.ca
Christopher Molloy – chris.molloy@queensu.ca
Michael Wrana – 16mmw@queensu.ca
Ross Hill – ross.hill@queensu.ca

# Table of Contents

## Abstract

This report analyzes the web browser Google Chrome, and attempts to provide a conceptual architecture encapsulating the system. Our team proposes a strict, object-oriented system that puts an emphasis on modularity, and protection of data. To properly dissect such a massive system, there needs to be a solid line of reasoning. Therefore, to provide contextual knowledge of the system, an overview of the architecture's derivation process will be given. Following that, an in-depth analysis of the architecture as a whole, and its specific subsystems, will illustrate how each object interacts with one another to provide a holistic and detailed view of Google Chrome's functionality. After the conceptual architecture is established, pragmatic issues such as possible development issues, concurrency, and the use of external interfaces will be discussed. Finally, the illustration of specific use cases, followed by a reflection on the architecture as a whole, will provide the base necessary to determine the concrete architecture of Google Chrome.

# Introduction and Overview

As the internet becomes more and more central in everyday life, the way we interact with it becomes extremely important. End users not only want the correct information - they want it fast, secure, and to be wrapped in a pleasing user experience. This has also given rise to the importance of the web browser.  Web browsers are expected to be able to browse the internet for web pages, send or receive information to others, and store information online. Web pages have become increasingly complex, moving well past the static HTML of the past. They often rely on intricate JavaScript, Flash, and support complex user experiences with videos and interactivity. This demand has made Google one of the world's most exciting companies of the 21st century – in no small part due to their web browser, Google Chrome. To properly understand Google Chrome as a piece of software, an analysis of its conceptual architecture must be given.

Google Chrome, launched in 2008, is the world's most used web browser accounting for around 66% of all desktop browsers, giving it a massive lead over its closest competitor, Mozilla Firefox (Elson). It has a simple user interface, and an intuitive user experience, allowing the user to get the information that they want without the browser getting in the way. While Chrome itself is not open source, Google released a large chunk of Chrome's source code under the Chromium project. This open source web browser shares the majority of its code and features with Chrome, and is an invaluable reference when analyzing Chrome's architecture. This may beg the question – why analyze Chrome's conceptual architecture at all? While there is inherent value in understanding a piece of software that a massive amount of people interact with every day, the same principles used in this report to derive Chrome's architecture are applicable to a wide number of different, complex systems.

To dissect Chrome's architecture, we first examined other reference architectures for web browsers, giving us some contextual knowledge to stand on. After that, we looked at what makes Chrome unique, namely its multi-process architecture and use of sandboxing.

At the heart of Chrome is its multi-process architecture, this means that every tab has its own process, which runs independently of the browser, plugins and other tabs. This has performance and safety implications; since every site instance per tab has its own process this can increase the browsers overall performance. As well, if a site happens to crash on one tab, it is contained in its own process and will not affect the rest of the browser.

Another one of the Chrome's core tenants is security - users store everything from their passwords, to their bank account information, to their home addresses in Chrome, so they must be able to trust the system's security. This is why Chrome came up with a sandboxing principle, which are processes that execute within a very restricted environment. For example, a sandboxed process is not able to write to disk or read files from the user's machine. This gives Chrome more control over what parts of the system can actually access system resources.

This report uses these two insights to help derive our conceptual architecture. An object-oriented approach that tries to maximise cohesion and minimize coupling amongst its components.

To gain a deeper understanding of the system, after deriving an adequate conceptual architecture of Chrome there has to be an analysis of its components and subsystems. Chrome is composed of millions of lines of code so organizing this system into the high level components; browser engine, rendering engine, display backend, networking, data persistence, and JavaScript Interpreter can give us a fuller understanding of the system that would not be possible by diving into the source code.

By deriving Chrome's conceptual architecture, we will be able to gain a better understanding of how Chrome's many features are implemented.

# Derivation Process

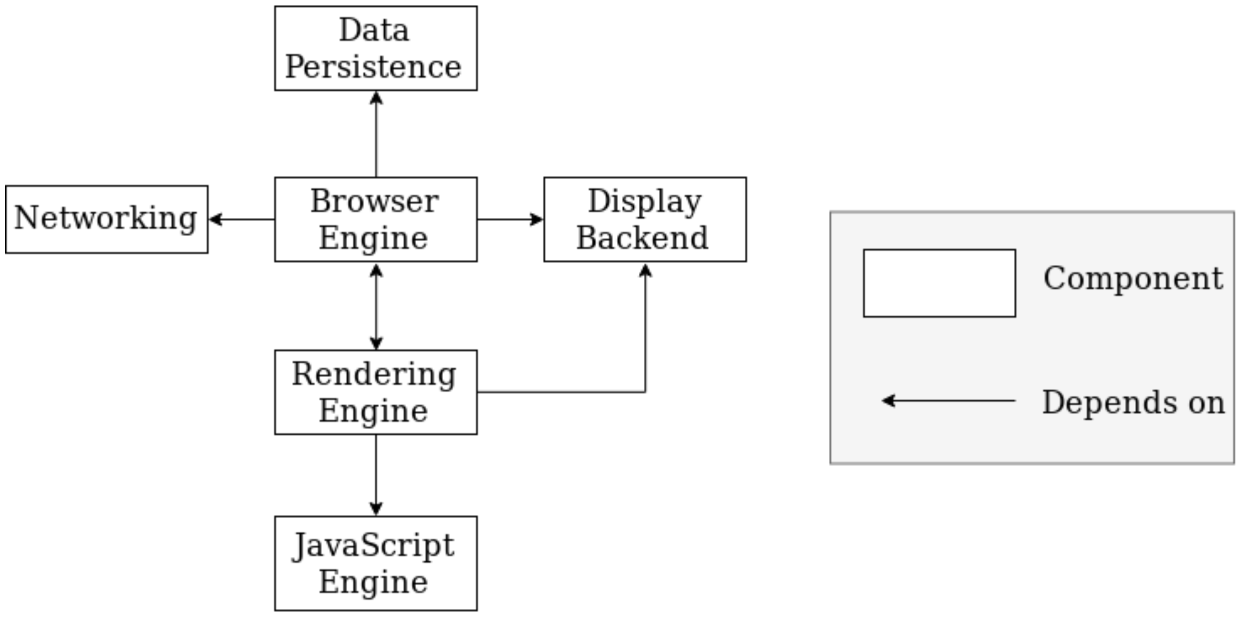
One of the most difficult parts of the architecture derivation process was starting out. The first piece of documentation we looked at was a paper[13] that gave some basic information about web browsers, which we used as our reference architecture. Although outdated, this paper was a great starting point which we built upon later. Next, we looked at the Chromium documentation and any information related to conceptual architecture. We compared this documentation to the paper, and tried to determine which old browser architecture was most similar to Chrome's. After collecting all this information, we began to understand how Chrome worked at a very high level.

At this point, we felt our understanding was good enough to start creating our own conceptual architecture for Chrome. One of the biggest challenges we faced while trying to develop it was zooming out. Many of our early designs had loads of boxes which contained one minor component and their own dependencies. Although these designs had very high cohesion, they also had very high coupling. This is where our research from earlier came in handy, it helped us understand which sub-components could be combined. Our later designs reduced both coupling and cohesion, so it was important to strike a balance . Although it is not perfect, after some deliberation we arrived at the design below.

# Conceptual Architecture

## Architecture Summary

Using the reference architecture given, we were able to derive what we believe is a clear, and concise, object oriented conceptual architecture. This is when you have a system of distributed objects that can interface with each other. Each object represents data, and its associated operations, in an encapsulated abstract data type[10]. Each object is responsible for preserving its data's integrity, making it suitable for a web browser - where corrupted data leads to an unsecure system (data breaches) and poor user experience (corrupted HTML). Below is an image of our conceptual architecture.



This modular system is able to reduce unnecessary coupling, while giving clear guidelines on how to maximize cohesion amongst components. By separating our main components into different objects, we can ensure that the integrity of the data is maintained, and if it's not the problem would reside in a single object, increasing its testability.

Cohesion is reasonably high as well. The internals of each component work independently of one another, and generally communicate with each other through the IPC thread. This improves how the system can evolve over time. If new requirements arise, developers can add in new functionality by building their components and making sure that it interfaces correctly with only the components it needs to interact with. Another example of how the system has evolved is in the case of Chrome switching its rendering engine from WebKit to Blink. With our current architecture, the developers could make this switch without drastically impacting the JS Interpreter, Browser Engine or Display Backend systems.

## Rendering Engine (Blink)

Forked from WebKit in 2013, Blink is an open source layout engine used for interpreting and laying out HTML. Blink's mission statement is: "to improve the open web through technical innovation and good citizenship." Chromium made the decision to fork Webkit's WebCore component as a result of increasing complexity between WebKit and Chromium. This is due to Chromium's multi-process architecture, which is unique from other WebKit based browsers. Blink handles rendering engine services needed by Chrome, including handling web components such as DOM, CSS, and HTML. Furthermore, the forking of WebCore allowed Chromium to immediately reduce unnecessary code , approximately 4.5 MLOC), and gave them greater flexibility when implementing their unique sandboxing and multi-process model.

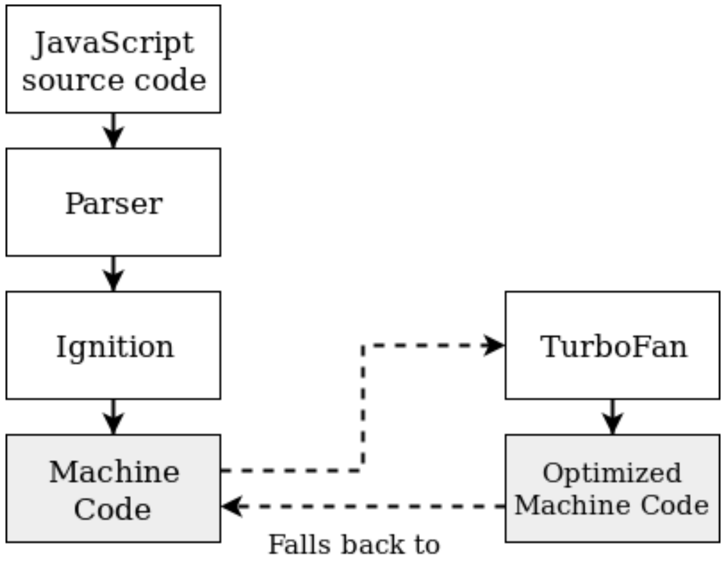
## Javascript Engine (Chrome V8)

In 2008, Chrome V8 was released as Chromium's JavaScript engine, a program that compiles and executes JavaScript source code. Javascript is specified by the EcmaScript Standard created by Ecma Technical Committee 39. The V8 engine uses the rules defined in the standard to run JavaScript source code.[15]

At one point, JavaScript being slow to execute was not detrimental because it was only being used for small website features. That changed in 2004 when Google released Google Maps, a fully featured mapping service based on web technologies. JavaScript engines at the time were unable to handle the resource heavy application, so Google created Chrome V8 as a modern JavaScript engine that is meant to run performance-intensive applications.[11]

Chrome V8 first turns the Javascript source code into an abstract syntax tree (AST), a structure describing the program that is easier for a compiler to work with. V8's compilers produce machine code from the AST.[14]

One feature of JavaScript is that it is dynamically typed, meaning that developers do not have to worry about variable types. For instance, one can define the variable `x` with `let x = 17;` instead of something like `int x = 17;`. This poses a challenge to JavaScript compilers because having type information allows the compiler to generate optimized machine code. The main difference in Google's new JavaScript engine was that it used just-in-time (JIT) compilation in order to optimize the output machine code. First, the baseline compiler (Ignition) produces machine code that looks similar to what prior Javascript engines might produce. As the code is run, the optimizing compiler (TurboFan) recompiles and optimizes "hot functions"—functions that that are reused frequently with the same argument types. When an optimized function gets passed a type that has not been optimized for, the Chrome V8 falls back to the baseline machine code.[14]



In summary, Chrome V8 takes Javascript source code and parses it into an abstract syntax tree. The AST is passed to Ignition which produces the baseline machine code. At the same time, Turbofan watches how the program is executed to look for any hot functions and produces optimized machine code which is used in favour of the baseline code.

## Browser Engine

The browser engine is the central part of our architecture and contains chrome's user interface and plugin subsystems. It creates a browser process which manages the UI, plugins and is in charge of the rendering processes being created for new tabs. Each render process has a global object that communicates with the browser process, while the browser has a render process host object which manages browser state and communicates with the renderer.[3] This is done through the browser's I/O thread.

Chrome's custom UI was built on a framework called views.[6] The UI was built with a tree of components responsible for rendering, layout and event handling. Some of the UI is not rendered in views but are native windows controls. Recently, Aura has been used for UI framework when building the ChromeOS.

Plugins are generally a source of browser instability seeing as they are run by third parties and their access to the operating system can't be controlled. Chrome solves this by running them as a separate process. The Plugin process host resides within the browser engine, then a separate process is created and then passed on to the rendering engine as needed.[9] This is advantageous as opposed to residing in the rendering engine because it means if two tabs use a plugin, only one instance of the plugin needs to be created.

## Networking

The networking component of Chrome is very autonomous and communicates only with the browser engine. Some of its goals include allowing coding to cross platform abstractions and providing greater control than with higher level system networking libraries.

The network stack is a generally single threaded library designed for resource fetching. It has two main interfaces, URLRequest and URLRequestContext.[8] URL request simply fetches the url and then url request context fetches additional details such as cookies or proxy resolver. It is mostly single threaded on the I/O thread with the browser.

## Display Backend

The display backend component of our conceptual architecture is composed of three main subsystems; Views, Skia, and GDI. These three subsystems are heavily dependent on each other for generating the widgets that make up a webpage's user interface, which is why we grouped them as the same component. These components work together with Blink to create the DOM tree that is then interpreted by the browser engine.

Views is Chrome's rendering system, similar to what WebKit uses. The user interface of a webpage is constructed from a tree of View components, analogous to how an HTML page is structured.

Painting and layout are done similarly; each view in the view tree needs to be painted from the upper left hand corner. The view constrains the size of the canvas that is to be painted, and then API calls are made to Skia and GDI to render objects onto the canvas.

Skia was developed in house and is used, "for nearly all graphics operations, including text rendering." GDI is a deprecated Windows graphics renderer, and was originally used by Chromium for text rendering. However, that operation has since been transferred to Skia, and GDI is now primarily used for native theme rendering.

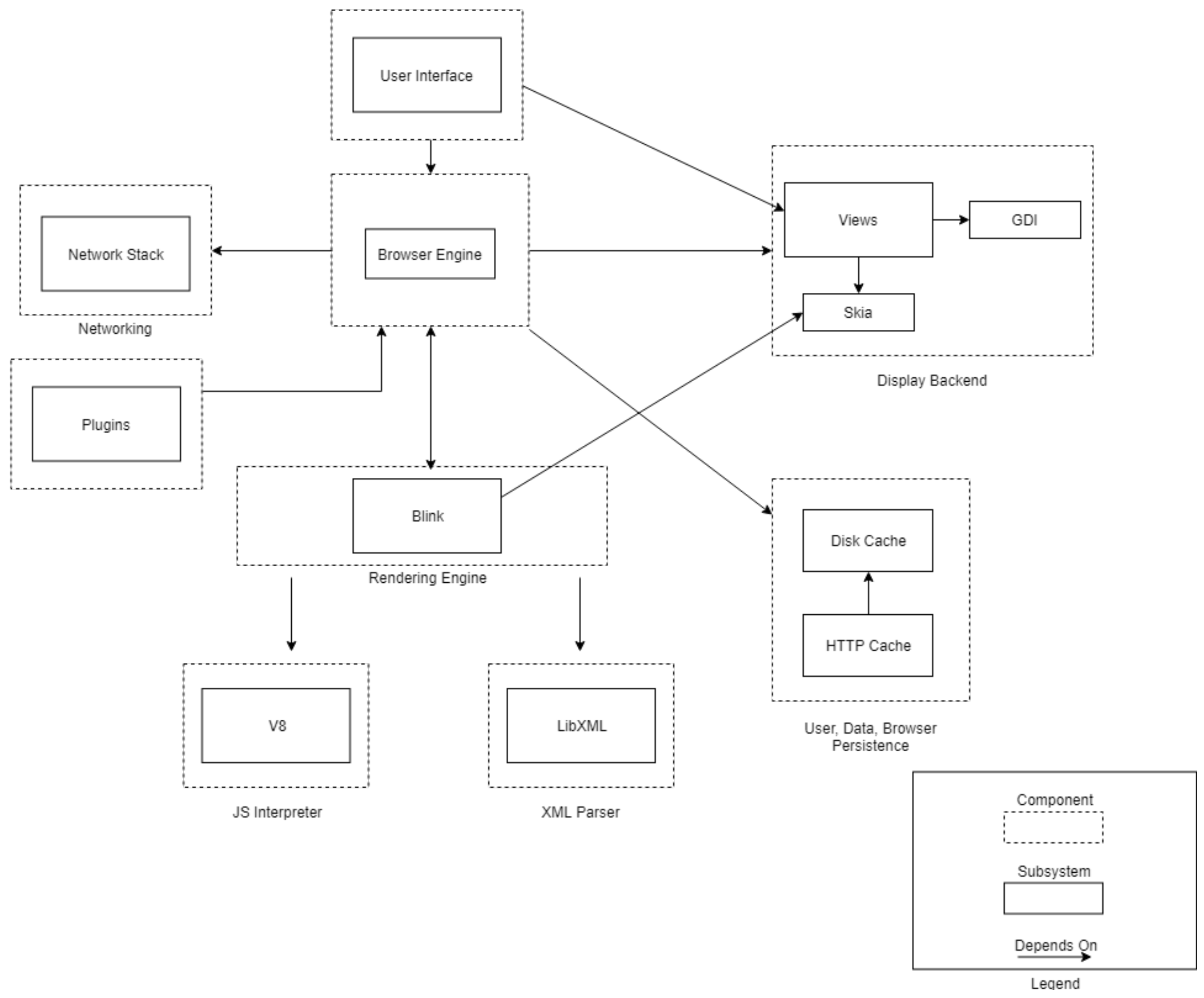
## Data Persistence

Data persistence involves data that is retained across session and browser tabs within a device. In our architecture, the data persistence system is composed of the disk cache and HTTP cache subsystems.

The disk cache holds resources from the web so they can be quickly accessed later. It uses two interfaces, the backend handles resources stored on cache and specific operations on resources are handled with the entry system.[12] An entry is identified by its key and is stored in separate chunks of data. All files stored in the disk cache are stored in a single folder, the cache. Every piece of data is given a cache address, a 32 bit number describing its location. Index information is stored in a hash table. Chrome stores user information in the cache such as passwords.

The HTTP cache receives requests and decides how to fetch data from the disk cache. It is responsible for the disk cache's backend and managing which transactions have access to it.[7]

# Alternative Architectures



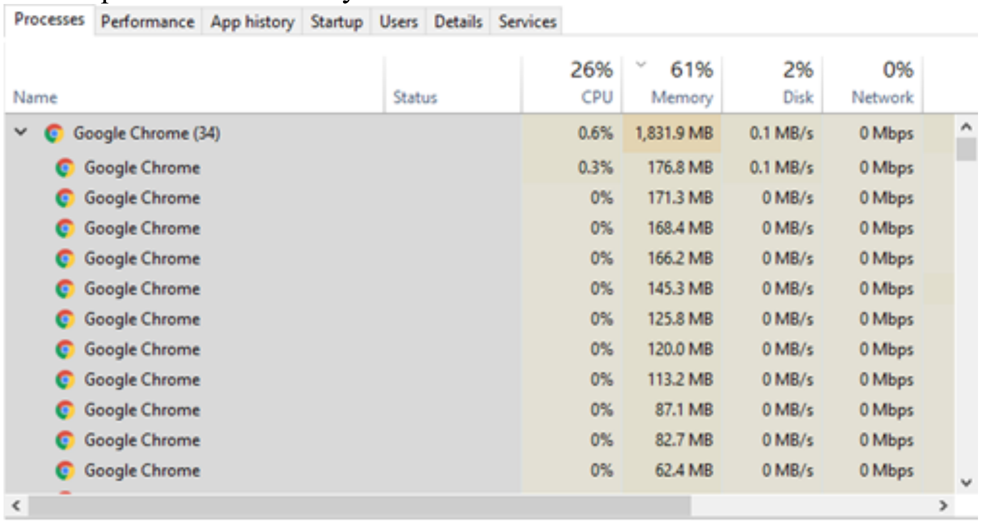
Above is one of the alternative architectures that our team came up with during the derivation process. The largest difference between our final decision and this alternative was the separation of the Plugins, UI, and Browser Engine components. Our reasoning for combining these three items into one subsystem is that this architecture contains too many boxes and complexity. We felt that it made sense to combine them as plugins are handled by the browser engine, as well as the displaying the user interface. As well, upon further research we decided that the XML Parser was too low level to be a main component of our architecture diagram. Blink has an XML Parser, so incorporating that into our rendering engine makes more sense conceptually. This architecture has higher cohesion than our final architecture, but it also has higher coupling. We felt as if trading off slightly lower cohesion for lower coupling in our final architecture was beneficial.

## Concurrency

The Concurrency of Google Chrome is the relationship between the Browser Engine and the Rendering Engine. When Chrome is running on a system the single browser engine will communicate with multiple renderer processes. These processes are standardly created for running each tab process on the system. This interaction mirrors an operating system, where each process is run separately, so if one of the renderers were to crash no other processes would be affected3. This is becoming more useful of a design every day, due to the increase in web pages complexity.

The Browser process acts as the manager to all other processes. In an instance of Chrome there will always be one browser process. Here all communication with the system takes place, any interaction from the user are interpreted here, along with any interaction with the disk, or network. Simultaneously with all this the browser administrates all other processes through its I/O thread. The Browser also handles all plug-in processes needed for the system, such as Flash. This may seem to be a lot of work, but the browser process does not generate any web content.

For each tab that is open on the system the Browser engine will create a renderer process for that webpage. The renderers use Blink to interpret all HTML, CSS, and JavaScript, and this is where the display of the webpage is created. All these renderers are kept in a sandbox, meaning that they are almost completely isolated from the system they are being run on. This forces all interaction between a webpage and the user to go through the Browser engine, where it can be monitored for questionable activity.



| Name | Status | 26% CPU | 61% Memory | 2% Disk | 0% Network |
|---|---|---|---|---|---|
| Google Chrome (34) | | 0.6% | 1,831.9 MB | 0.1 MB/s | 0 Mbps |
| Google Chrome | | 0.3% | 176.8 MB | 0.1 MB/s | 0 Mbps |
| Google Chrome | | 0% | 171.3 MB | 0 MB/s | 0 Mbps |
| Google Chrome | | 0% | 168.4 MB | 0 MB/s | 0 Mbps |
| Google Chrome | | 0% | 166.2 MB | 0 MB/s | 0 Mbps |
| Google Chrome | | 0% | 145.3 MB | 0 MB/s | 0 Mbps |
| Google Chrome | | 0% | 125.8 MB | 0 MB/s | 0 Mbps |
| Google Chrome | | 0% | 120.0 MB | 0 MB/s | 0 Mbps |
| Google Chrome | | 0% | 113.2 MB | 0 MB/s | 0 Mbps |
| Google Chrome | | 0% | 87.1 MB | 0 MB/s | 0 Mbps |
| Google Chrome | | 0% | 82.7 MB | 0 MB/s | 0 Mbps |
| Google Chrome | | 0% | 62.4 MB | 0 MB/s | 0 Mbps |

Here, Windows recognizes each of the webpages and the Browser as their own processes.

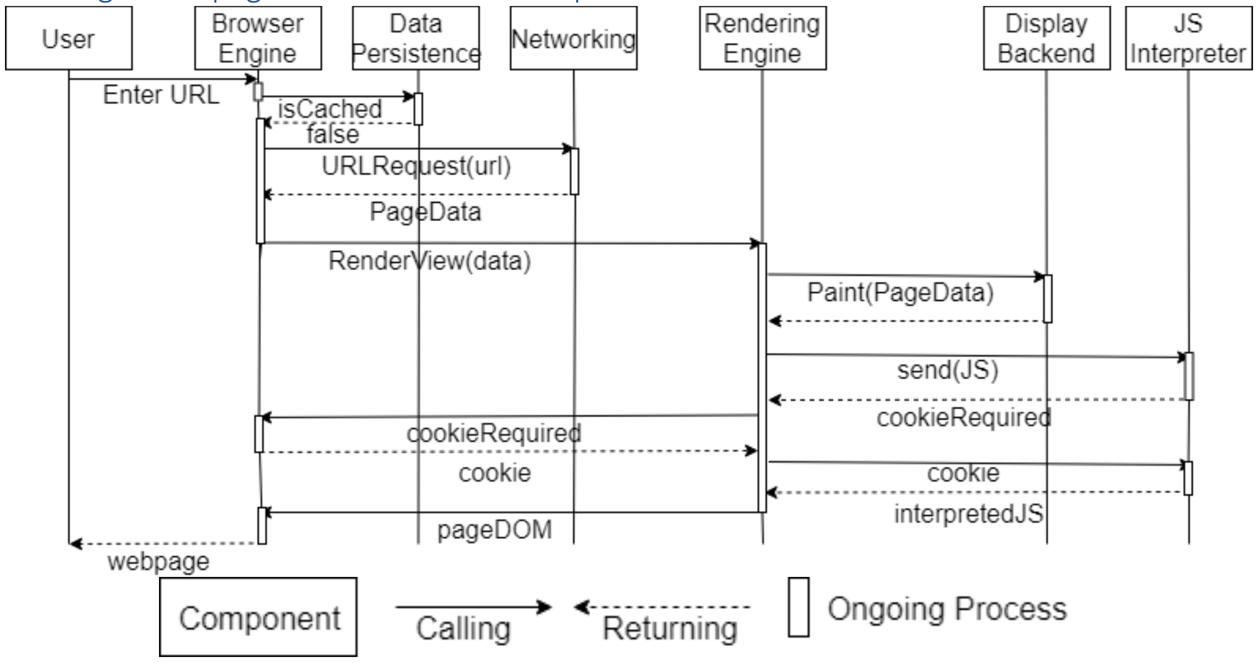
## External Interfaces

Chrome requests all information on web pages through its network stack. To initially load a webpage, the stack will request a URL, and the web page's content will be returned, such as the HTML, CSS, and JavaScript. The network stack also handles all requests from the web page, such as requesting the information on a file to download[7]. When communicating with databases, the stack will send a URL request along with all information it wants changed, such as changing a password, or information it wants to receive, like the image to a profile picture.

Furthermore, the network stack will send out all needed information to a webpage like cookies or cached information. The information on the cache is run by the HTTP cache, which decides what data needs to be retrieved from the disk cache or network8. The cache is an integral part of information conversation between the system and network. For example, managing the disk cache on initialization, creating the HTTP cache transactions, and managing the active entries in the HTTP cache.

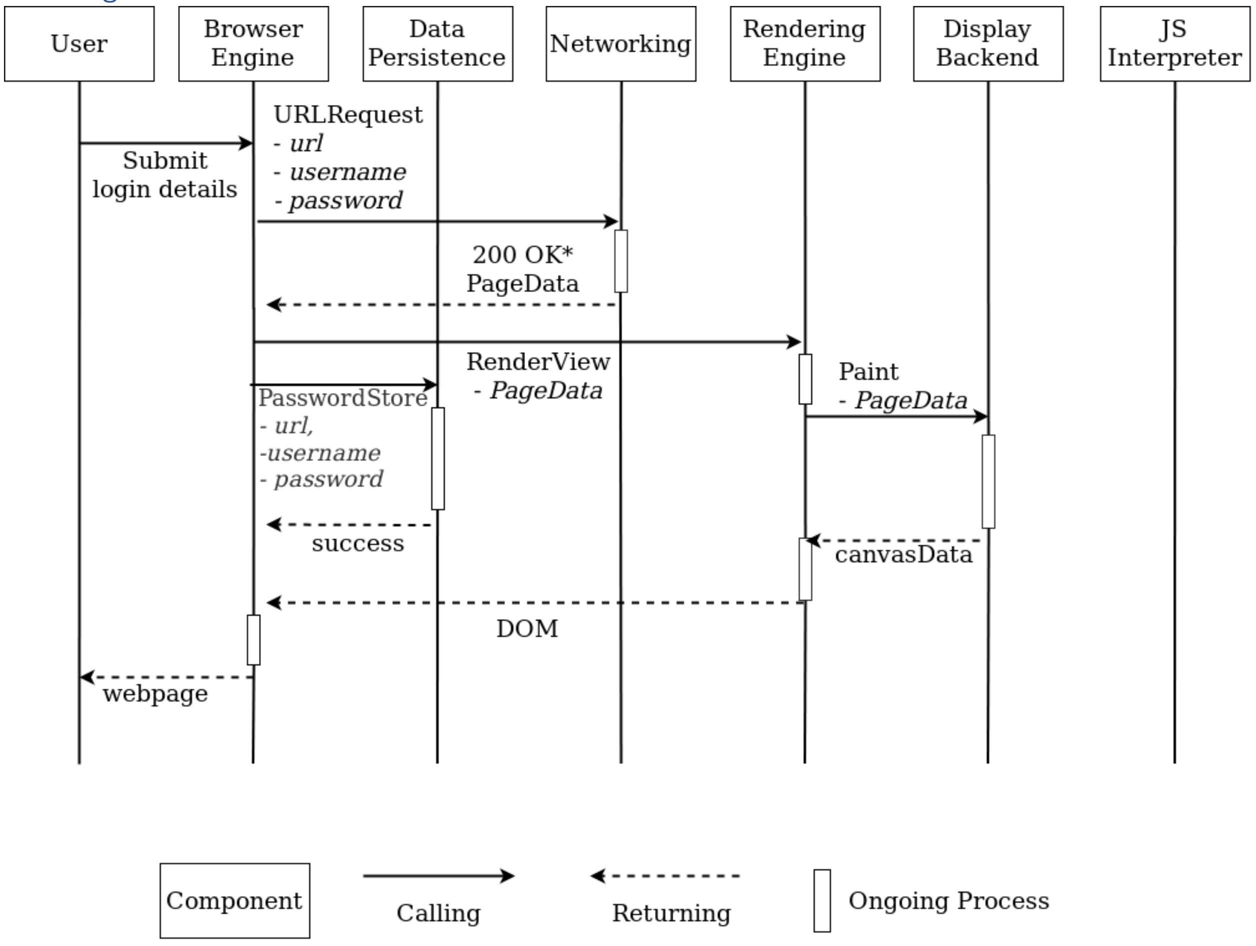
## Sequence Diagrams

### Loading a Webpage with Cached Javascript



The webpage request begins with the user entering or clicking on a URL. The Browser engine checks if this URL has been cached. In this case it has not, so page data is requested from the networking component. After page data is received by the browser engine, it is sent to the rendering engine. The page data is rendered in the display backend, while Javascript is sent to V8. Since a cookie is required, the browser engine retrieves the cookie and forwards it through the rendering engine to V8. The Interpreted JS is returned. Once rendering is complete, the DOM is sent to the browser engine and displayed through the UI to the user.

## User logs into a website and Chrome saves the Password



*200 is an HTTP status code that indicates that the request has succeeded

The user submits their username and password through a website's login form. First, the Browser Engine sends the login HTTP request to networking, which communicates with the third-party web server. A 200 OK HTTP response with the webpage data is received from the server which Networking then returns to the Browser Engine. Seeing that the login details were authenticated, the Browser Engine calls Data Persistence to store them. Meanwhile, the Browser Engine calls Blink which constructs and renders the DOM using the display backend. The rendered webpage is then returned to the Browser Engine and displayed to the user.

## Limitations and Lessons Learned

Many lessons were learned and limitations were faced. Firstly, Chrome documentation and general information is readily available and there is lots of it. We had difficulty in sorting through what was relevant to our project. In the future, we will aim to first understand what we are actually looking for before attempting to dive into the documentation or code. Chrome has also been around for a while now, meaning people are less concerned with its architecture and design. This made finding up to date and relevant information challenging. We also had difficulty in even starting our conceptual architecture. The research paper provided was helpful, but it was hard to apply it to something newer and different like Chrome. Chrome is also quite complex with many interacting parts. It was challenging to hash out all of the components and their interactions.

## Conclusion

As previously mentioned, deriving our conceptual architecture was a challenging and interesting experience. However, once we had everything figured out, we gained a strong high level understanding of Chrome's architecture and behind the scenes interactions that will prepare us well for upcoming projects. Our most important finding was Chrome's differentiation from other browsers with things such as the multiprocess architecture.

## Data Dictionary & Naming Conventions

**AOT:** Ahead-of-time (compilation)  
**AST:** Abstract Syntax Tree - A tree structure denoting the semantic routes possible in source code.  
**CSS:** Cascading Style Sheets - A tool for formatting HTML components  
**Cache:** A small amount of fast, expensive memory  
**DOM:** Document Object Model - A universal specification for laying out and giving access to HTML components.  
**HTML:** HyperText Markup Language - An encoding language used to format a static web page layout.  
**HTTP:** HyperText Transfer Protocol - defines how messages are formatted and transmitted on the World Wide Web  
**IPC:** Inter-Process Communication - the communications channel created to deliver messages between processes spawned by Chrome  
**Ignition:** Chrome V8's baseline compiler  
**TurboFan:** Chrome V8's optimizing compiler  
**JIT:** Just-in-time (compilation)  
**URL:** Uniform Resource Locator- reference to a web resource on a network and how to retrieve it  
**MLOC:** Millions of Lines of Code

# References

1. Elson, Sarah. "Top 5 Most Popular Desktop Browsers in 2018." Medium, 10 May 2018, https://www.medium.com/@sarahelson81/top-5-most-popular-desktop-browsers-in-2018-17011afb6dc
2. "Blink." *The Chromium Projects*, www.chromium.org/blink.
3. "Multi-process Architecture." *The Chromium Projects*, https://www.chromium.org/developers/design-documents/multi-process-architecture.
4. Barth, Adam. "Blink: A Rendering Engine for the Chromium Project." *Chromium Blog*, 3 Apr. 2013, blog.chromium.org/2013/04/blink-rendering-engine-for-chromium.html.
5. "Graphics and Skia." *The Chromium Projects*, https://www.chromium.org/developers/design-documents/graphics-and-skia.
6. "Views." *The Chromium Projects*, https://www.chromium.org/developers/design-documents/chromeviews
7. "Network Stack." *The Chromium Projects*, https://www.chromium.org/developers/design-documents/network-stack
8. "HTTP Cache." *The Chromium Projects*, https://www.chromium.org/developers/design-documents/network-stack/http-cache
9. "Plugin Architecture." *The Chromium Projects*, https://www.chromium.org/developers/design-documents/plugin-architecture
10. Hassan, A. E. (2018). Lecture 03: Architecture Styles [PDF slides]. Retrieved from Queen's University, http://cs.queensu.ca/~ahmed/home/teaching/CISC322/F18/slides/CISC322_03_ArchitectureStyles.pdf
11. Kennedy, Niall. (2008). The Story Behind Google Chrome. https://www.niallkennedy.com/blog/2008/09/google-chrome.html.
12. "Disk Cache." *The Chromium Projects*, https://www.chromium.org/developers/design-documents/network-stack/disk-cache
13. Grosskurth, Alan, and Michael W Godfrey. A Case Study in Architectural Analysis: The Evolution of the Modern Web Browser. A Case Study in Architectural Analysis: The Evolution of the Modern Web Browser.
14. Nickolov, Lachezar. How JavaScript works: Parsing, Abstract Syntax Trees (ASTs) + 5 tips on how to minimize parse time. https://blog.sessionstack.com/how-javascript-works-parsing-abstract-syntax-trees-asts-5-tips-on-how-to-minimize-parse-time-abfcf7e8a0c8
15. Google. V8 Documentation. https://v8.dev/docs