Assignment 2

Google Chrome's Concrete Architecture

November 9 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 gives an analysis and overview of the web browser, Google Chrome, and its concrete architecture. Through deriving file-folder relations found in the source code, and iteratively adding to Team Roblox's previous conceptual architecture, a thorough reflexive analysis can be given at a deeper level than previously doable. In doing so, our Object-Oriented conceptual architecture will transform into a more complex and thorough concrete architecture. From there, further analysis into the inner workings of Chrome's source code will provide a solid base to jump to in assignment three.

Introduction and Overview

After gaining a basic understanding of Google Chrome at a high level, it is now time to dive into the source code and gain a deeper understanding of Chrome as a piece of software. By working through Chrome's conceptual architecture, we learned how a system that big operates. While that is helpful, to fully understand its implementation we must learn about Chrome's unexpected dependencies, hacks and how they fit in to the broader picture.

To do this we started off with our base understanding – the conceptual architecture – then using SciTools' software *Understand* we started to map the Chromium directories to our conceptual architecture. This proved challenging as there were various unexpected dependencies that we had to investigate, which will be discussed further.

After doing this reflexion analysis we can do a deep dive into our browser engine subsystem at a lower level. From this level of understanding, we can go into more depth on the sequence diagrams we touched on last project – loading a webpage with JavaScript and saving a password.

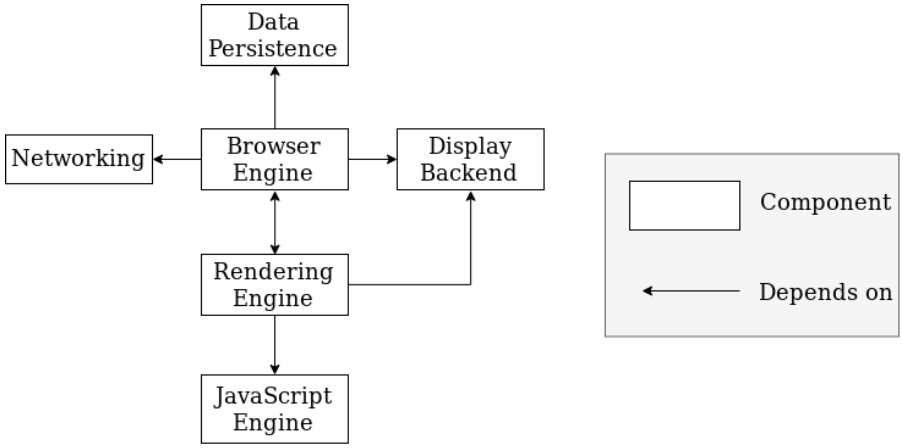
This deeper understanding of Chrome as a system provides insights to the concurrency of Chrome's multi-process architecture, which will prove valuable for assignment 3, where we discuss implementing a new feature into Chrome.

Derivation Process

The first step we took towards deriving a concrete architecture for Google Chrome was understanding the software and tools available to us. Specifically, *Understand* was a key component in this process. No one in our group had worked with this or a similar software program before, and we were unsure of how to use it. After figuring out the basics, we began to actually derive our concrete architecture. For us, this process was an iterative one with the steps as follows: The first step was generating the concrete architecture in *Understand* and comparing it to our conceptual architecture. Next, we combined and moved files and folders around to better represent our interpretation of Chrome's architecture. Finally, we updated the conceptual architecture to reflect new information learned. This process was repeated until we found an architecture, we were confident in.

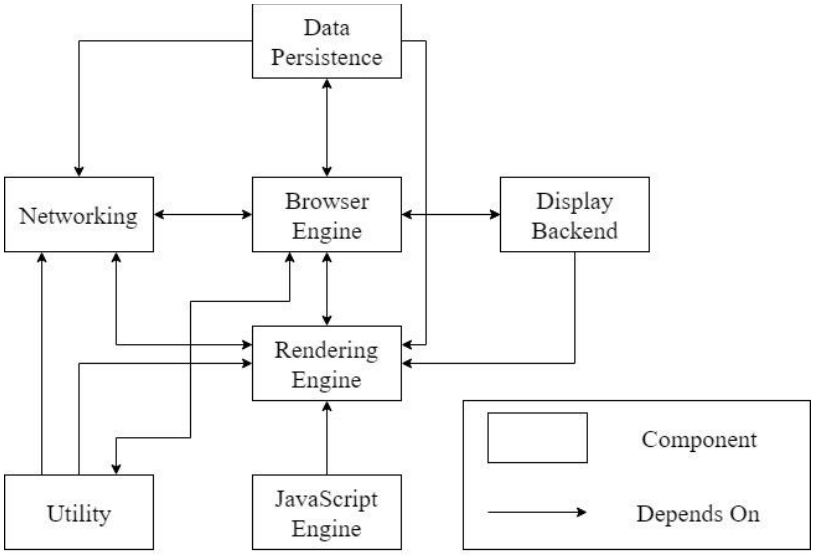
Concrete Architecture

Original Conceptual Architecture



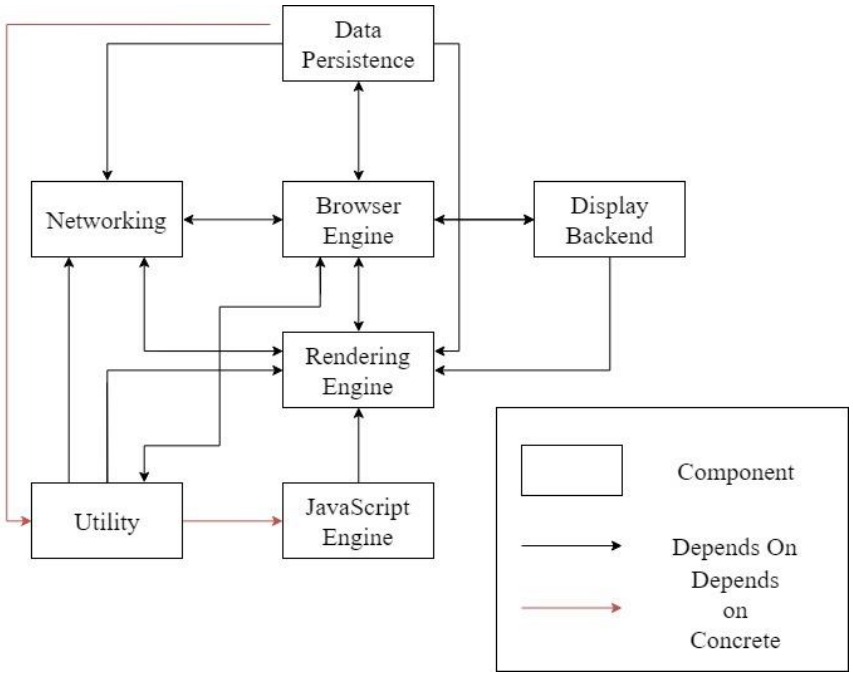
Our initial conceptual architecture was object-oriented with limited interactions between the systems. However, after looking into the source code we decided to make some changes to our conceptual architecture.

Revised Conceptual Architecture



We added a utility subsystem to hold the utility and services subfolders. These house a wide variety of libraries and systems including bookmarks, importer systems, and decoders. They depend on many other systems and we decided it would be easiest to house them in a new subsystem to ease the readability of our diagram. There are also numerous dependencies which we at first did not include. All of our existing dependencies went from being one-way dependencies to going both ways which is logical for all the communication between systems. The exceptions to that are that we initially had rendering depending on display backend and rendering depending on java, whereas we now have the reverse. This was a logical oversight as it's obvious the display backend and java require calls from the rendering engine in order to operate. New dependencies we added include: data persistence -> networking, data persistence -> rendering, and networking <-> rendering.

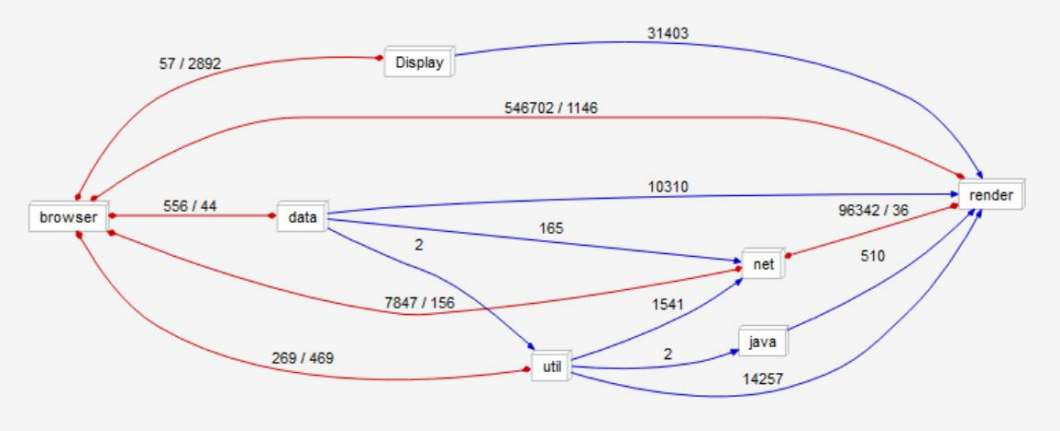
Concrete Architecture



Our concrete architecture differs from our conceptual because of two additional hacks. Hacks are interactions between systems that should not exist, but that developers put in to help system interaction. Data persistence depends on utility for the special storage policy interface which grants special rights to extensions and applications. Some types of storage are protected from browser intervention. For example, the durable storage is not subject to storage eviction, while other processes are only allowed to store session-only data which is erased when the session ends.

Utility depends on the JavaScript Engine for two image decoder unit tests which test that decoded images are not too large and can be properly processed by the IPC.

Reflexion Analysis



As you could see in our concrete architecture, there was high coupling with many interdependencies between subsystems. This was somewhat shocking at first, but we determined that it was largely due to some “hacky” solutions, and some dependencies that we wouldn’t have thought of until we saw them. We also saw a decent amount of testing code scattered throughout, which we just chose to ignore as it did not add anything of value to our concrete architecture.

As you could see in our concrete architecture, probably the biggest change was the utilities subsystem. When generating our concrete architecture with Understand, we were finding a lot of dependencies that just weren’t making sense. We concluded that we needed a utilities subsystem to account for some of the folders such as services, and some utility folders found throughout the provided code. This helped clear up some confusion and make our architecture more readable.

Another big difference was the dependencies on the networking subsystem. We originally did not have many dependencies to or from the networking subsystem, but it made sense as networking is needed for many crucial things such as caching, rendering, and data persistence.

Interesting Dependencies

A large difference between the initial conceptual architecture of Chrome and the revised version was the addition of multiple dependencies that were initially understood to be one way. This can be seen from the Data persistence, Networking, and Display backend relying on the Browser Engine. The first conceptual architecture had the Browser Engine and Rendering Engine rely on these subsystems on their own, but the Understand software showed that these dependencies are reciprocated by their respective subsystems. The Browser Engine and Rendering Engine both do a lot of work, and it was initially understood that the other subsystems could run their processes on their own, acting as providers to the engines and not clients.

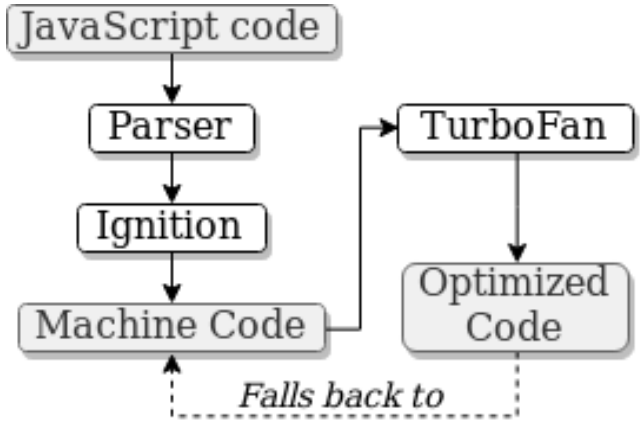
Three dependencies that were completely missed in the first conceptual architecture are the dependencies from Data Persistence to Networking, and Rendering Engine, and the two-way dependency between the Networking subsystem, and the Rendering Engine. The Data Persistence subsystem relies in the Networking subsystem for database related trackers, along with local file reader, and writer methods. Data Persistence also relies on the Rendering Engine for confirming SQL commands, along with error callback support. The relationship between the Networking subsystem and the Rendering Engine was the most significant dependency that was missed in the first conceptual architecture. These two subsystems rely on each other for methods like network traffic annotations. A network traffic annotation is a way of recording what the intent a webpage has behind a network request, and what user data is being requested (Threading and Tasks in Chrome).

Subsystem Analysis: JavaScript Engine

A JavaScript engine is a program that compiles and executes JavaScript source code (V8 Project Authors). JavaScript is specified by the EcmaScript Standard created by Ecma Technical Committee 39. In 2008, Google created Chrome V8 as a modern JavaScript engine that is meant to run performance-intensive web applications like Google Maps which had been recently released (Kennedy, 2008).

The JavaScript Engine is divided into three subsystems: the JavaScript Parser, the Baseline Compiler (Ignition), and the Optimising Compiler (TurboFan) (V8 Project Authors). The parser takes JavaScript source code as input and outputs an abstract syntax tree (AST), a structure describing the program that is easy for a compiler to work with. Ignition takes the AST and generates unoptimised machine code which is executed from the filesystem. The machine code Ignition generates includes inline caches to log the argument types of function calls. TurboFan uses the data collected by Ignition to optimise the machine code (V8 Project Authors).

TurboFan can use the collected data to optimise the machine code because JavaScript is dynamically typed. For instance, one can define the variable x with *let x = 17* instead of *int x = 17*. This poses a challenge to JavaScript compilers because having type information would allow the compiler to generate optimised machine code. The JavaScript Engine uses just-in-time compilation (JIT) to handle this problem. As the baseline machine code is run, the optimising Compiler uses Ignition's inline caches to optimise functions that that are called frequently with the same argument types (hot functions). When an optimised function gets passed a type that has not been optimised for, the JavaScript Engine falls back to the baseline machine code (Nickolov).



The JavaScript Engine's only dependency is to the Rendering Engine, due to the need for JavaScript programs to access and modify the Document Object Model (DOM) of the webpage. In summary, Chrome V8 takes JavaScript source code and parses it into an AST. 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 optimised machine code which is used in favour of the baseline code.

Concurrency

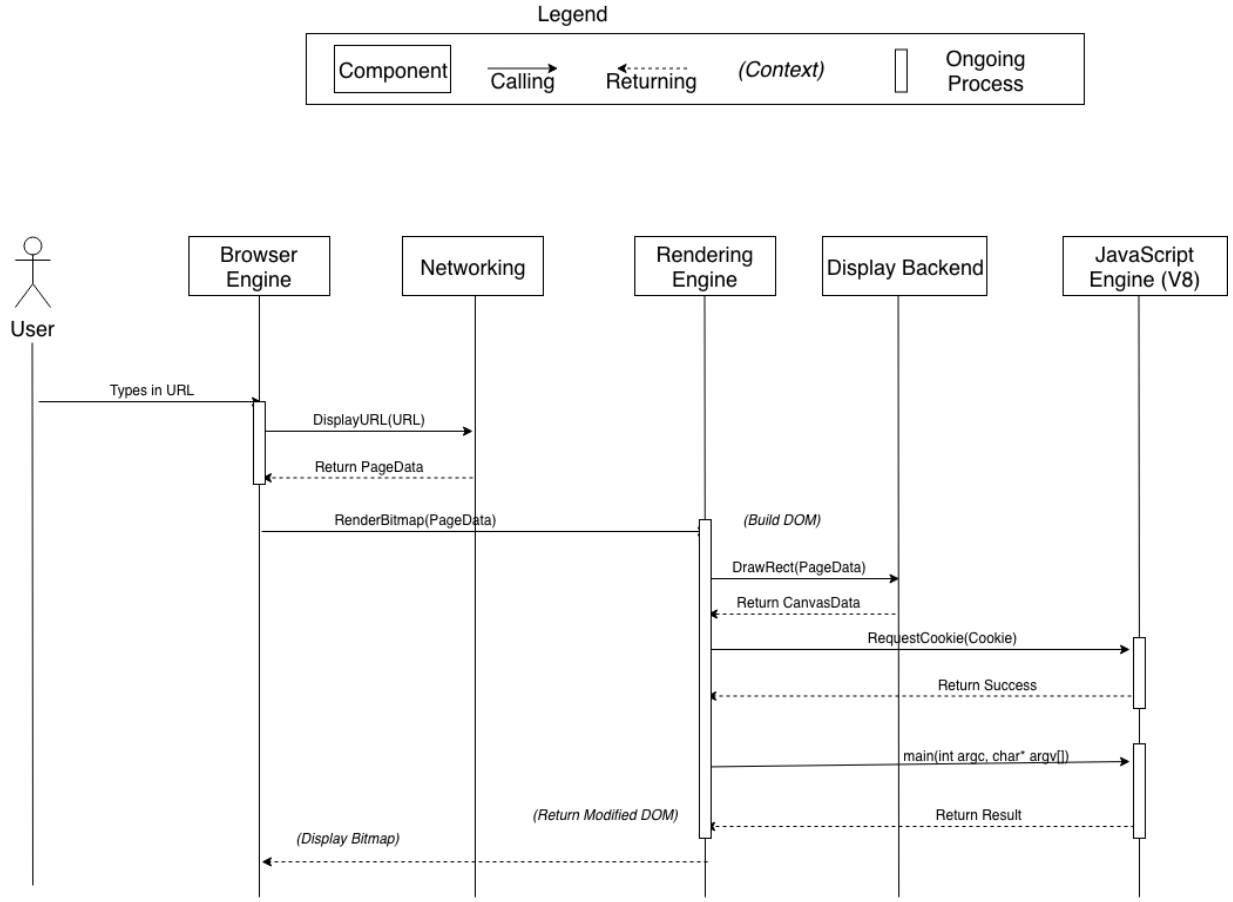
The concurrency of Google Chrome is the relationship between the Browser Engine and the Rendering Engine. When an instance of Chrome is running on a system the single Browser engine will manage the multiple rendering processes inside the Rendering Engine. The Rendering Engine will create a render process for each tab open in Chrome, acting more like an operating system than a normal program (The Chromium Projects). As websites grow more complex, they become more likely to crash. Google is combating this by building Chrome to act more like an operating system, keeping all the processes separate from one another. Having each of these processes independent from one another stops a problem on one webpage affecting the activity of another.

The Browser Engine acts as the manager to all the renderer processes. When Chrome is running, the single Browser Engine communicates to the Renderer processes, and the system it is being run on. Along with this, the Browser Engine manages all the renderer processes through its I/O thread. The Browser Engine will also govern all plug-in processes needed for the system, such as Flash. Although this appears to be a large amount of work on the Browser, it does not generate any webpage content.

All visuals for the web pages are rendered and interpreted in the Rendering Engine. The Rendering Engine will produce a renderer process for each tab open on the system, called renderers. As of 2013 the renderers have used Blink to interpret HTML, CSS, and JavaScript. These renderer processes are kept in a sandbox, meaning all communication requested by a webpage to the system is run through the Browser Engine, this is an extra security measure to protect the user from malicious websites.

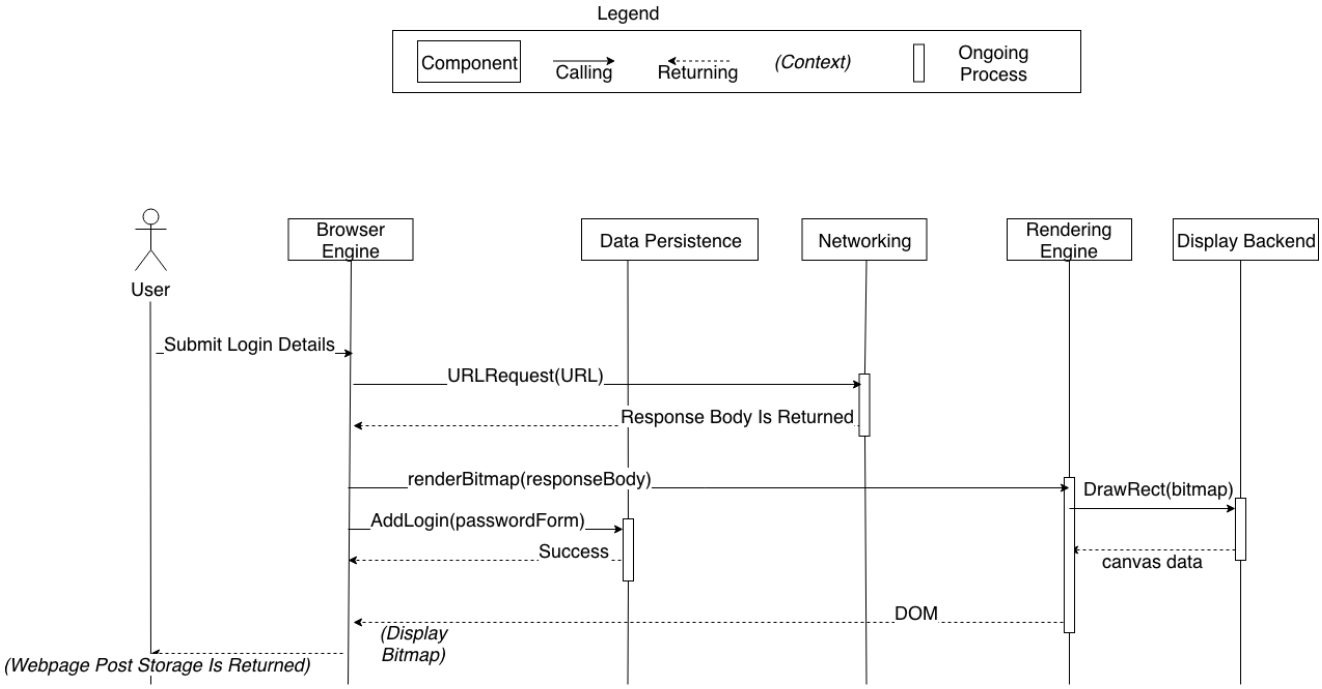
Sequence Diagrams

Loading a Webpage with Cached JavaScript



The webpage request begins with the user entering or clicking on a URL. The browser engine handles this request and sends the URL to the networking component to request the page data. 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 (cs.chromium.org). Since a cookie is required, Blink forwards a cookie to the JS engine which interprets the JavaScript and modifies the DOM (V8 Project Authors). Once rendering is complete, the modified 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



The user starts off by typing in their username and password, and clicks "Enter" on the login form. From there, the browser sends the post storage URL to the networking subsystem through a URLRequest. The response body is sent to Blink, the rendering engine, which interacts with the display backend subsystem to construct the DOM. While this happens, the browser engine constructs a PasswordForm. This is a struct that encapsulates information about a login form, which can be an HTML form or a dialog with username/password text fields (cs.Chromium.org). That data-structure is then sent to the data persistence subsystem, where the 'Web Data' SQLite database stores the username/password and the form metadata that is associated with the PasswordForm (Boyd, 2013). On success, the rendered webpage post storage is displayed to the user.

Limitations

By far the biggest limitation for this project was the size of Chrome and the time available to us. Google Chrome is a massive project with over 5 MLOC. Given only two weeks to develop an architecture diagram proved challenging. We had to try and focus on the bigger picture and avoid getting dragged into the details of individual files and subsystems. To do this, we had to make assumptions about details for which there was not sufficient time to investigate. Another limitation we found was inconsistencies between the Chromium documentation and source code. Often the source code would differ from the documentation on crucial components. Given these differences, we were forced to make assumptions about which information was correct, which resulted in conflicts within the concrete architecture and especially the sequence diagrams.

Lessons Learned

One important lesson we learned was using software like *Understand* to analyze code. Before starting this project noone in our group had any idea where to begin with a project like this. However, using software tools gave us a great basis and starting point to look at Chrome's architecture. This information might be very useful in the future if any of us need to do something similar. Another lesson learned from this project was how extraordinarily complex a massive project like Chromium actually is. Generating a top-level architecture similar to assignment one makes it seem simple. However, while diving deeper we learned how truly extensive all the subtle interactions between files and folders actually are.

Conclusion

To conclude, using Understand in our derivation process was a challenging yet interesting experience. It helped us uncover some unexpected dependencies that we had not originally thought of. Generating our concrete architecture led to some changes from our conceptual, such as a utilities sub system and some new dependencies. The new dependencies were interesting to analyze and gave us a better sense of how certain things work in Chrome. In generating our object-oriented concrete architecture, we now have a strong understanding of at least some of the functionality of Chrome.

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 language 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
SQL: Structured Query Language - Standard language for database management and communication
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 Optimising 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
Hot Function: A function that is often called with the same argument types

References

1. Boyd, Ian. “How Does Google Chrome Store Passwords?” Super User, 2013, superuser.com/questions/146742/how-does-google-chrome-store-passwords.
2. Cs.chromium.org, 2013, cs.chromium.org/chromium/src/out/android-Debug/gen/components/autofill/content/common/mojo_types_java/generated_java/org/chromium/autofill/mojom/PasswordForm.java?type=cs&q=PasswordForm&sq=package%3Achromium&g=0&l=17.
3. V8 Project Authors. “V8 Documentation” V8 JavaScript Engine, https://www.v8.dev/docs.
4. 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
5. Kennedy, Niall. (2008). The Story Behind Google Chrome. https://www.niallkennedy.com/blog/2008/09/google-chrome.html.
6. “Multi-process Architecture.” The Chromium Projects, https://www.chromium.org/developers/design-documents/multi-process-architecture.
7. “Network Traffic Annotations.” Threading and Tasks in Chrome, chromium.googlesource.com/chromium/src/ /lkgr/docs/network_traffic_annotations.md.