November 9th, 2018

Chrome Concrete Architecture Report

Authors
Brynnon Picard (15bdrp@queensu.ca - #20005203)
Roy Griffiths (18rahg@queensu.ca - #20137434)
Alex Galbraith (18asrg@queensu.ca - #20135646)
Sam Song (15shs1@queensu.ca - #10211857)
Bradley Kurtz (15bdk2@queensu.ca - #20020794)
Dongho Han (16dhh@queensu.ca - #20027554)

1.0 Abstract

This report describes our concrete and revised conceptual architecture of Chrome, outlines our derivation process, and explains our rationale for each subsystem, module and connection in our architecture. In our previous report we obtained our initial conceptual architecture from the documentation of the open source analogue, Chromium. This allowed us to identify major subsystems and crucial design elements such as Chrome's multi-process architecture, use of implicit invocation, and Renderer/Browser decoupling. Through this process and running through use cases allowed us to identify the less important but nonetheless crucial subsystems, Network, Graphics, Extensions, and Plugins.

In producing this report we examined the source code supplied to us. We did this through the Understand tool, where we were able to see the dependencies between components and subsystems. This greatly aided us in creating our concrete architecture as well as updating our conceptual architecture. Understand also allowed us to reveal the components within the subsystems which we didn't know existed. The process of examining the source code and dependencies was repeated until we had a diagram exhibiting all the relationships with minor alterations. From this, we found that our architecture was remarkably more coupled than we had initially thought it was. Places where we had one component relied on another were often actually relying on each other to function.

Ultimately, our proposed architecture remained and Object-oriented, multi process architecture supported by implicit invocation for communicating between the Browser and Renderer. Despite all the additional coupling, our architecture was by and large supported by our analysis of Chrome's concrete architecture.

2.0 Introduction

The purpose of this report is to explore the concrete architecture of Google Chrome and explain the process our team used to derive our final concrete and revised conceptual architecture. The report has four main content sections. The first section is derivation where we will discuss how we derived our architecture and the rationale for many of the major decisions. The second discusses interesting unexpected dependencies. Discussion of unexpected dependencies will include files, components and source code involved, and the reason for why the dependencies exist. The third section section describes our final architecture and sequence diagram in detail, including a description of each subsystem and module, as well as the connections between them. The fourth section reviews limitations in our analysis and lessons we learned during the process of derivation. Finally, we will wrap everything up with a conclusion, followed by a list of references.

In our previous report, we determined that Chrome has both an object oriented architecture and a multi-process architecture, supported by implicit invocation. We constructed an initial architecture diagram (see Figure 3) using only documentation on Google Chrome without

looking into the code at all. For this investigation, we now look into the code of Google Chrome and reassess our original architecture. To do so, we derive a concrete architecture of Google Chrome which includes all dependencies within the code. We next strip away any hacks leaving behind Chrome's conceptual architecture.

Our derivation of Chrome's concrete architecture is driven by the use of Understand tool, Chromium's design document website, and the Chromium code search tool. Using Understand tool we organized code directories into components, which produced a dependency graph. We then divided dependencies into two parts, legal and illegal. Legal dependencies are a connection between components that exist for a proper reason (e.g. resource sharing and constant sharing). Illegal dependencies are connections that exist because of an error in Understand tool, incorrectly categorized file, or a hack by a developper. Illegal dependencies are analyzed and removed and all unexpected legal dependencies have been reflected on and explained in chapter 4.2 Reflection Analysis. Since there are too many dependencies to analyze, we have omitted some trivial dependencies which will be explained later.

Derivation of sequence diagram was done using Chromium's documentation and Chromium code search. We started the derivation by reading Chromium documents on specific use cases [7]. Then making guesses on which method should be responsible for a certain functionality. After finding the correct method, we used the inspector tab on the Chromium code search website to follow call hierarchy upwards and downwards to complete the sequence diagrams.

3.0 Derivation Process

After having a complete view of Chrome's code, we have gained a great deal of knowledge of Chrome's architecture, and confirmed many of the decisions we made on our initial conceptual architecture diagram (see Figure 3) and sequence diagrams. With the help of the source code analysis program Understand and Chromium code search website, we were able to derive a new subsystem and new dependencies between our previously existing subsystems.

We began by mapping source code to our original conceptual architecture. Our concrete architecture has many unexpected dependencies, most subsystems relied on most other subsystems. We figured this was likely incorrect, so we looked deeper into the source code folders, and pulled out bits of code that were in the wrong components, and moved them to the right ones, giving us the conceptual diagram in Figure 1. We initially believed that we could put the "base" folder into Browser, which resulted in a massive number of dependencies to and from Browser. We soon realized that this was essentially all common code, so we split it off into its own Library component. We then established which remaining dependencies were hacks, based on careful code analysis, and removed these hacks to produce our revised conceptual diagram seen in Figure 2. Later in this report we will discuss the dependencies and hacks found.

To justify the dependencies, we picked out some dependency files and figured out what they did. To do this, we looked into the subsystem's file to see what was being imported and what functions or variables were being used. To determine the purpose and functionality of the

dependencies, we utilized the comments in both code files, looked at what the functions specifically did (inputs, outputs), and the "readme" text files in each subsystem being examined.

As for the sequence diagrams, we first searched for code relating to a password being submitted. Once we found this, we would continue tracing the method calls until we reach the point of a password being saved. Sometimes we worked in reverse though, and found a function that occurs a lot later in the sequence diagram. We were able to do this by using cs.chromium.org's feature that displays which files call the selected method. It was sometimes challenging to trace the method calls if IPC was used to communicate with other subsystems, as it wasn't always clear what files were used to handle the receipt of an IPC message.

We then updated our conceptual diagram from the previous assignment by adding the new Library subsystem and new dependencies found from the concrete diagram, but excluding the unnecessary hacks found.

4.0 Concrete vs Conceptual Architecture

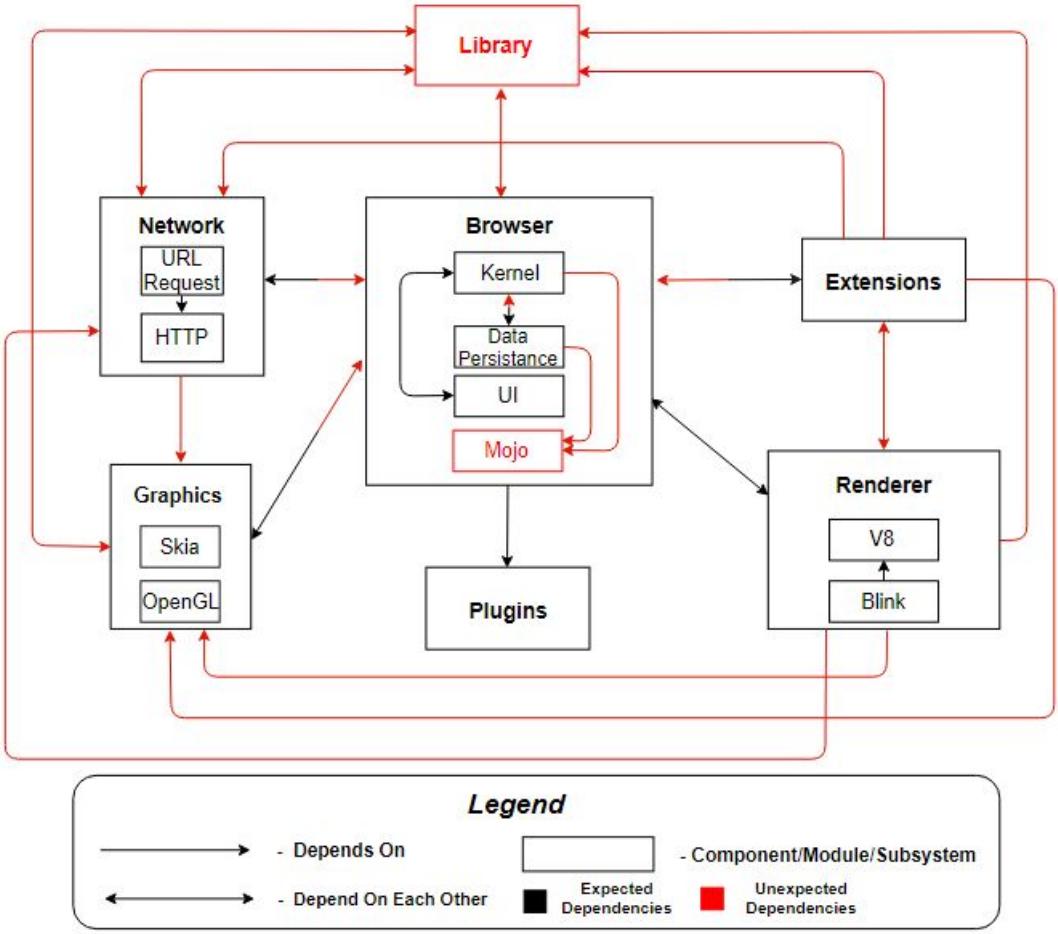


Figure 1: Concrete architecture diagram

4.1 Reflexion Analysis

Below are a collection of unexpected dependencies we identified and analysed from our concrete architecture. It is worth noting that we have not included dependencies on the Library subsystem as these are fairly obvious: all subsystems depend upon Library for shared code such as string manipulation.

Network -> Graphics
File *quic_server.cc* depends on *components/viz/common/features.h*

Description: "features.h" appears to be a file to hold constants related to the Graphics component. "quic_server.cc" is a file that handles communication over the QUIC transport-layer network protocol. There appears to be no reference to any of the constants "features.h" exports inside the "quic_server.cc" file, only the initial import statement. Since the import statement looks through the PATH environment variable path list and the LIB environment variable path list, it's possible that Understand has detected it as the wrong features.h file, as there are many in the source code, and obviously some code has been removed by TAs.

Network -> Browser
File *cert_verifier.h* uses type *CertificateList*, which is defined in *user_network_configuration_updater.h*

Description: "user_network_configuration_updater.h" defines the type CertificateList, which represents an array of X509 public key certificates. "cert_verifier.h" is used to verify the validity of certificates and uses the CertificateList type to store a list of trust anchors, which are private certificates typically used by software on your computer that intercepts network traffic (like security software). This looks like a hack to avoid reusing code by defining the CertificateList type twice.

Extensions -> Network
File *url_handlers_parser.cc* depends on *net/base/network_change_notifier.h*

Description: In the dependency between Extensions and Network, "network_change_notifier.h" is used for monitoring the network for changes, such as disconnecting from WiFi, Bluetooth, 3G, and as well as IP changes. It then notifies registered observers of those events, using implicit invocation style architecture. "url_handlers_parser.cc" seems to be the observer class for Extensions to receive these events from Network. This appears to be a legitimate dependency as some extension programs would need to know the status of the network connection.

Extensions -> Renderer
File *chrome_extensions_client.cc* depends on *url_constants.h*

Description: This was found to be a hack as it doesn't use any methods and only uses constant variables stored in Renderer. Almost all of the constants comprise of URLs for "Learn More" pages for various purposes such as an error page. This dependency likely exists purely for

code-reuse because it would be inefficient to copy and paste the same links into the Extensions subsystem.

Extensions -> Graphics
File *chrome_extension_messages.h* depends on *ui/gfx/transform.h*

Description: "chrome_extension_messages.h" handles inter-process communication related to Extensions, and is used to forward messages to extension processes when necessary. "transform.h" seems to be a file for performing transformation operations on matrices. More specifically, "chrome_extension_messages.h" appears to send data related to the accessibility for extensions, in the form of a data structure called an accessibility tree. "transform.h" provides methods and variables related to how something would be displayed/transformed, so that "chrome_extension_messages.h" can pass that on as part of the data needed to support accessibility in Chrome.

Renderer -> Extensions
File *chrome_extensions_renderer_client.cc* includes *extensions_constant.h*

Description: Renderer depends on Extensions in order to illustrate all extensions dialogue boxes. It require constants for render to render correct form of graphics for all extension. For example, extensions.h includes "extern const char kTextEditorAppId[]" to recognize given extension is text editor extension not an in-app payment support application.

Renderer -> Network
File *web_url_request_util.cc* relies on *net/base/load_flags.h*

Description: As it turns out, Renderer has a fair number of files related to fetching network resources, which we did not anticipate. One such file is "web_url_request_util.cc" which contains support for requesting files such as images, scripts, fonts, stylesheets, and more. "Load_flags.h" in Network provides flags used for general, generic information about load requests. This dependency seems to exist so that code would not be duplicated.

Renderer -> Graphics
Cached_bitmap_ Types CrossThreadSharedBitmap at *pepper_graphics_2d_host.h*

Description: Renderer uses the resource under graphics/cc/resources. Cached bitmap is a bitmap that was recently released by the compositor and maybe used to transfer byte to the compositor again. This dependency exists for resource reuse purposes. Renderer can now use bitmap stored as cache under graphics directory instead of calling new bitmap everytime

Graphics -> Browser
File *buffer_queue.cc* relies on *display_snapshot.h*

Description: There are a large number of dependencies between Graphics and Browser, many of which are tests or logging hacks, although there is also a fair bit of code related to the UI frame, which Graphics needs to know the scale and location of to properly draw graphics. One of the

more interesting dependencies is from “buffer_queue.cc”, which is a framework for reading and writing to GPU memory. “buffer_queue.cc” relies on “display_snapshot.h” which takes a “snapshot” of the current display state at any given time. The reason for this dependency is that UI is inside Browser, so to save the display state Graphics needs to communicate with UI.

Extensions -> Browser
File *notification_style.cc* includes *message_centre_constant.h*

Description: When Extensions tries to renderer a notification, it needs Browser to pass constants that determine size of all graphical components. When Extensions notifies the user using a dialogue box, Browser will tell Extension the size of the dialogue box, width and length of okay button, among other things. This is something that could be internalized as developer mentioned “// *TODO*(estade): many of these constants could be internalized.” at message_center_constants.h::14.

4.2 Comparison of Original and Revised Conceptual Architecture

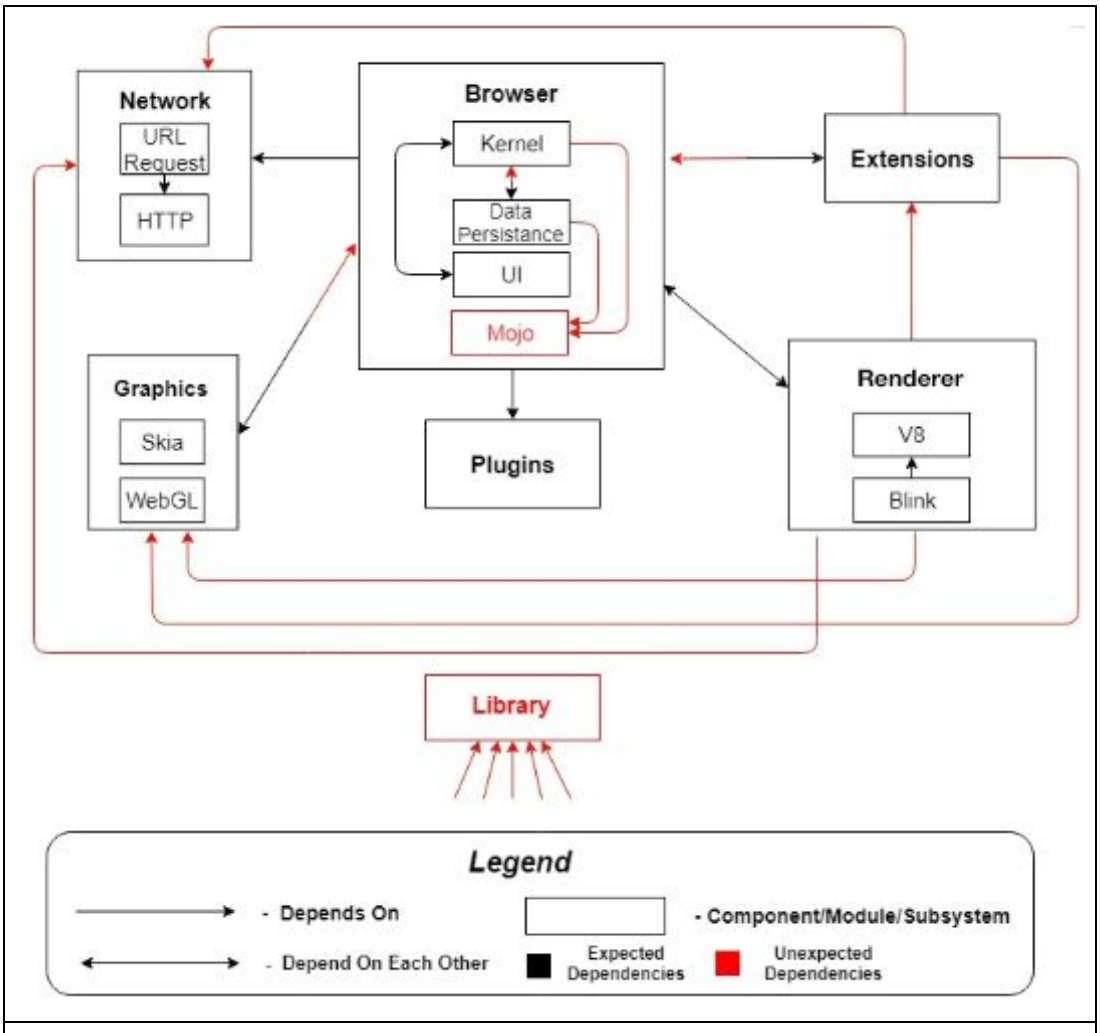


Figure 2: Our revised conceptual architecture diagram

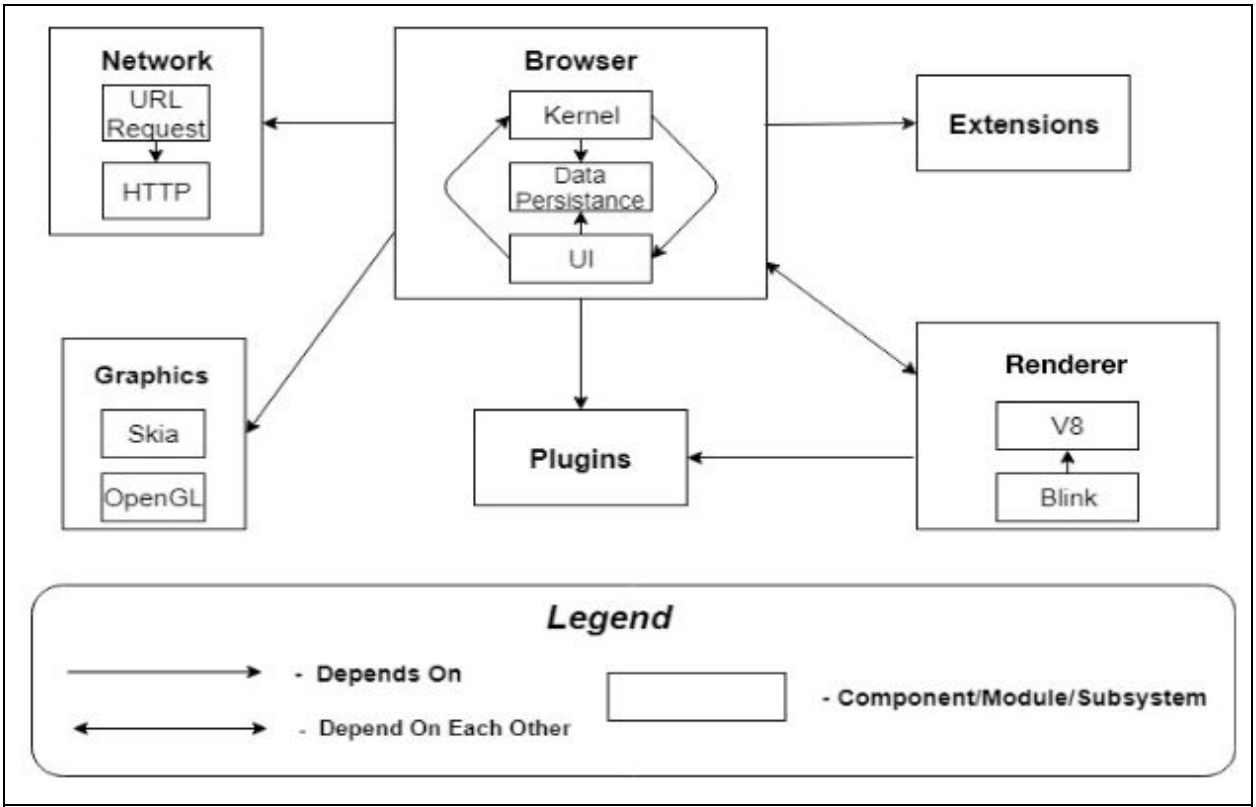


Figure 3: Initial Conceptual Architecture

Our revised architecture includes newly added component library and a subsystem mojo and many new dependencies. Inside the Browser subsystem, new dependencies from Data Persistence to Kernel, Kernel and Data Persistence to mojo. Chapter 4.1 explained the rationale of these newly created dependencies, and 4.4 will cover the purpose of each subsystem. The dependency between Rendering to Plugins is removed because plugin related code was removed from the Chromium file, and Understand tool could not make actual dependencies.

Ultimately, comparing Figures 2 and 3, our architecture has not changed all that much. While many extra couplings have been introduced, our fundamental subsystems have, with the exception of Library, remained untouched. Most importantly, our investigation into the code of Chrome confirmed what we had learned from the documentation: Google Chrome has an Object oriented architecture, and a multiprocess architecture supported by implicit invocation for inter-process communication. The presence of Mojo, entire folders dedicated to sandboxing, and excessive amounts of implicit invocation throughout the codebase confirm our previous findings.

4.3 Sequence Diagrams

Below are our updated sequence diagrams.

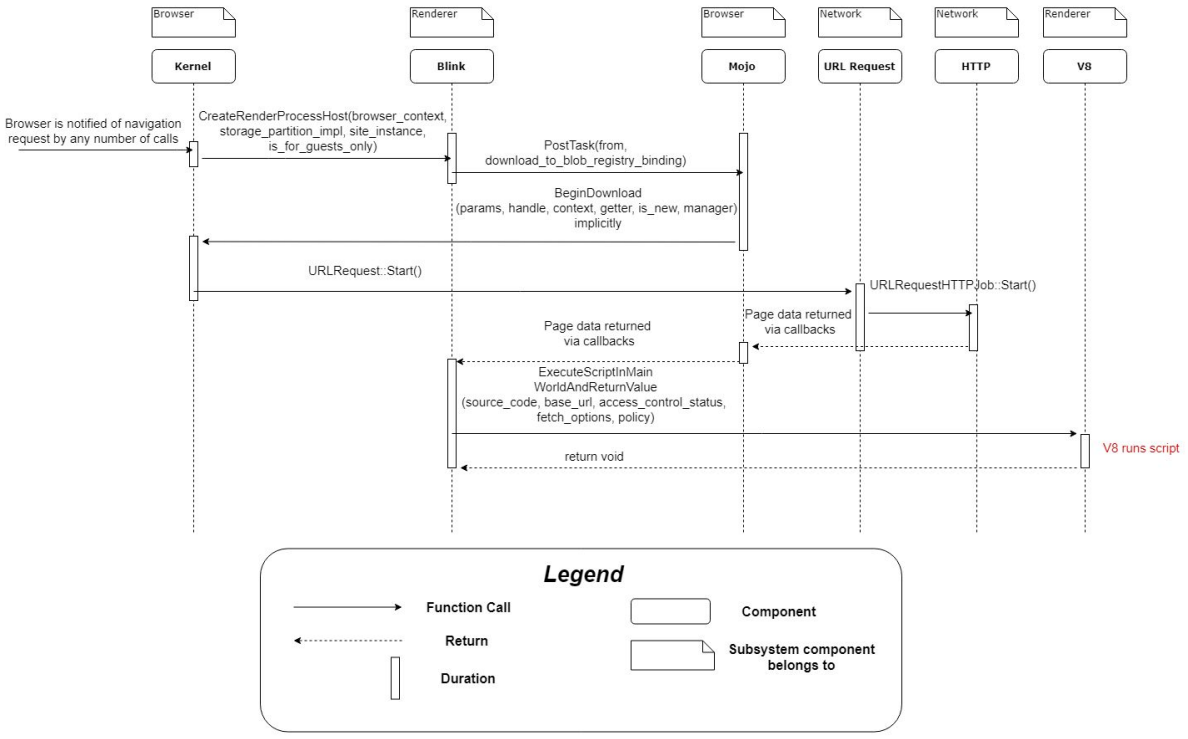


Figure 4: Sequence diagram illustrating Chrome rendering page with Javascript.

Our first use case entails Chrome displaying a page with Javascript. Figure 4 illustrates the sequence of events that occur in order to render the page. First, Chrome is notified of a navigation request. This request could come from any number of sources including a Javascript navigation, a user interacting with the UI, or a redirect request. Next, the Browser constructs a new Rendering instance by calling CreateRenderProcessHost which creates both the Browser and Renderer side of the IPC communication pipes. The new Renderer is constructed with knowledge of its site instance, i.e., the page it should be loading. Once constructed the Renderer's Blink component sends a resource request over IPC to Mojo in the Browser using PostTask to implicitly pass a download-to-blob-registry binding. This binding requests that a piece of data be downloaded and placed inside the Renderer's local data store, or blob registry. Mojo then forwards this request to the Kernel using BeginDownload, which causes the Kernel to create a URLRequest and run it. The URLRequest recognizes the request as an HTTP request, and creates a URLRequestHTTPJob which when started, retrieves the page data. At each step the objects were provided with callback functions, and these are used to pass the data back down the chain to Mojo, which then passes the data back to Blink over IPC. A callback then triggers Blink to begin processing the received data. Blink then recognises there is Javascript to run, and

invokes ExecuteScriptInMainWorld to run the Javascript, which then returns void. It is worth noting that this is only one possible path. At almost every stage the route changes depending on the origin of the request and the data returned. There are also many hundreds of inter component calls for minor requests we have not included as these would only confuse the diagram.

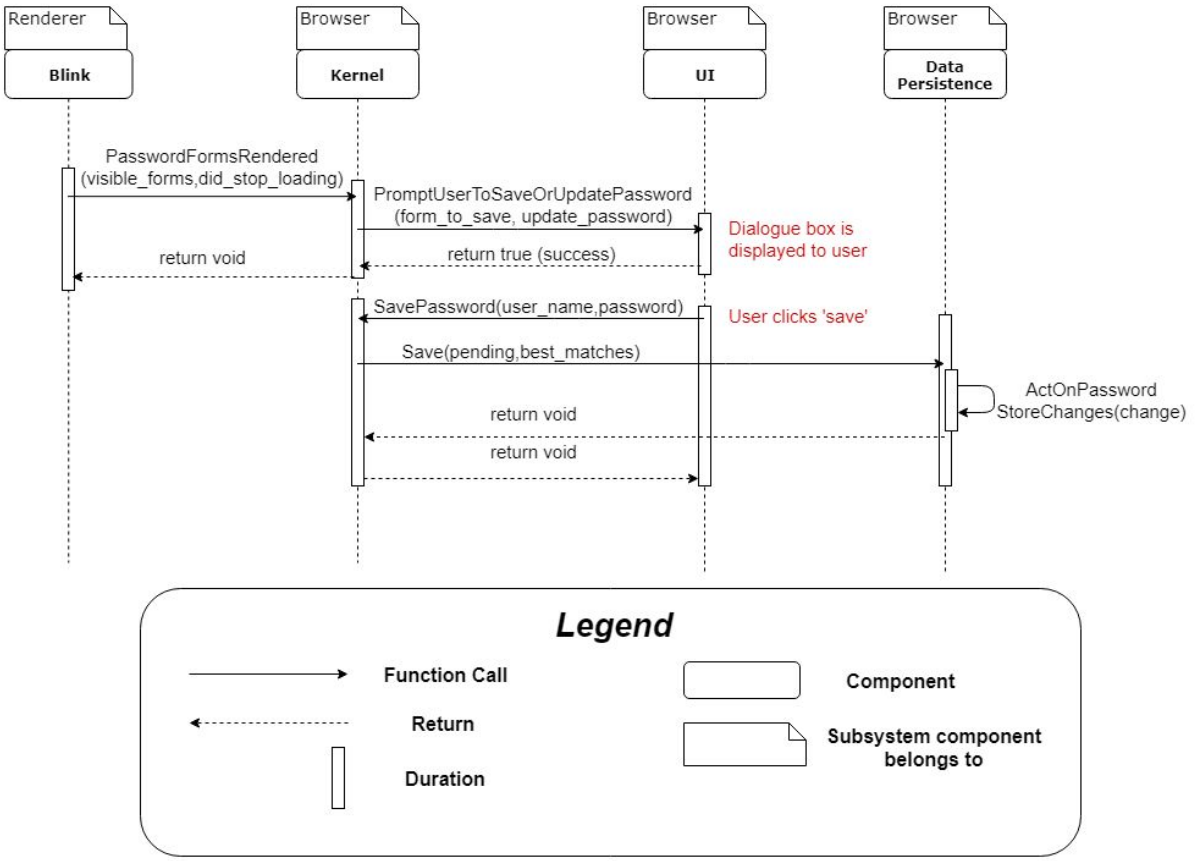


Figure 5: Sequence diagram illustrating the use case of user sign in in and credential store by Chrome.

Our second use case entails a user signing into a web page and having their credentials stored. There are two separate sequences shown in Figure 5, one for submission and one for saving the password. When the user clicks enter on the sign in form, Blink will send a function call to the kernel through PasswordFormsRendered(visible_forms, did_stop_loading). Then Kernel will call UI through PromptUserToSaveOrUpdatePassword() so UI can display a dialogue box to the user.

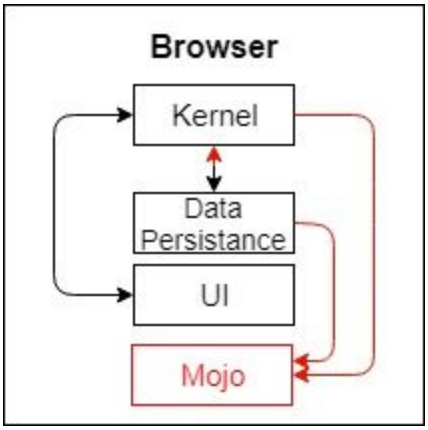
Once the user clicks “save password” the UI starts a new process by first calling SavePassword function to Kernel in order to begin the password saving stage. After Kernel receives the username and password, then it sends all data to Data Persistence using the Save function. Data

Persistence then modifies its database and enters the new credentials by calling ActOnPasswordStoreChange.

4.4 Subsystem Breakdown

4.4.1 Browser

The Browser subsystem is at the core of our whole architecture, linking between all other subsystems and processing any requests which have been made. This is achieved with the aid of the four main modules: Kernel, Data Persistence, UI and our new component, Mojo. As we saw in our previous diagram, the objects in red are our unexpected components and dependencies whereas our objects in black are our expected ones.



Kernel: This is considered the "brain" of the browser. All data marshalling between other subsystems is handled here, thus effectively sandboxing the whole environment. Kernel depends on data persistence, UI and our new component Mojo. Kernel must depend on Data Persistence so it is able to load content such as extensions. Kernel must also depend on UI so that it is able to return the result of a user input if it has an effect on the interface. An example of this could be if Javascript (running in a renderer) opens a new tab, this command is sent to Kernel, which must then notify UI. Kernel must rely on Mojo so that it is able to communicate with other subsystems such as the Renderer with IPC.

Data Persistence: All data such as sign in details, bookmarks and cookies are stored internally here. Data Persistence can also use the network subsystem to sync data through the cloud. Data persistence must rely on Kernel as we found out in the event where data within the component has changed and thus must notify Kernel about it. An example of this may be when a password has been updated. Data Persistence must also rely on Mojo to communicate to the Renderer with IPC, as was with the case with Kernel.

User Interface (UI): As the name suggests this is where all user input is handled. The UI depends on Kernel. This is so user input such as the pressing of a key or the movement of the mouse can be sent to Kernel for processing and distribution.

Mojo: We have added this new component to our browser subsystem after having examined the source code. It handles and processes all interprocess communication (IPC).

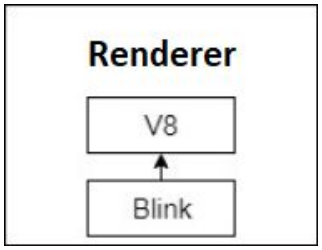
4.4.2 Rendering

The Rendering subsystem's function is to parse and render web pages. It contains 2 modules which support this functionality. These are: Blink and the JavaScript V8 engine. Blink is a

rendering engine, which is responsible for rendering HTML and CSS web pages. It, in turn, relies on the V8 engine to run any JavaScript that a given page uses.

Blink: Blink is a rendering and layout engine, responsible for rendering web pages to display to the user. It was created as a fork of the WebCore component of the Webkit engine, which is used in browsers like Safari. Blink is developed separately from Webkit and contains support for the Document Object Model (DOM) which enables JavaScript manipulation of HTML, and Scalable Vector Graphics (SVG) [2]. Blink relies on the V8 engine for executing JavaScript code, which is an important part of most modern websites.

V8 Engine: The V8 engine is a high-performance JavaScript engine initially developed for Chrome. It improves performance over traditional JavaScript interpreters by compiling JavaScript code directly into native machine code before execution. This contrasts with traditional approaches such as interpreting bytecode. Additional optimization is done dynamically at runtime, through a variety of methods including inline expansion and inline caching [3].



4.4.3 Graphics

Graphics is a subsystem for drawing graphics to the user's device display. This is different from Rendering, which is at a higher level and is more abstracted. Graphics contains the Skia graphics library, which contains APIs for drawing to numerous target devices.

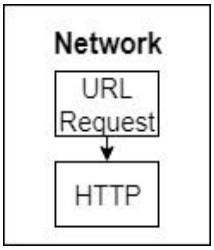
Skia: Skia is an open source 2D graphics library used in Chrome, the Android OS, and Firefox, among others. Skia enables Chrome to draw shapes, text, lines, and images. Skia's extensive API enables effective graphics drawing across many platforms and devices [4].

WebGL: WebGL, or Web Graphics Library, is the primary library used by Chrome to render 3D graphics. This was formerly OpenGL but after some code inspection we found out that this was in fact what was used.

4.4.4 Network

The network subsystem handles all networking functions of Chrome, such as communicating with servers and transferring files. It employs two modules for these tasks: URL Request, and HTTP.

URL Request: URL Request is responsible for handling requests to and from external servers. It keeps track of the various requirements that might be needed to fulfill a given request, such as cookies, host resolver, proxy resolver, or cached data [5]. It relies on the HTTP module to fulfill requests that require communication over HTTP.



HTTP: Hypertext Transfer Protocol is the primary application protocol of the internet, and is at the core of any web browser's network communications. The HTTP module is used by the URL Request module to handle all HTTP traffic.

4.4.5 Extensions

Extensions contain the application programming interface (API) for third-party developers to use for writing browser extensions, small programs that affect the user's browsing experience. Browser extensions such as ad blockers and password managers have become widely used today, and extensions are a significant part of nearly every major browser. Since extensions typically require data to function, the Extensions subsystem also provides the ability for an extension program to interact with and modify received HTML, CSS and JavaScript code.

4.4.6 Plugins

Plugins are similar to "Extensions" but it is more integrated into the browser's core functionality and is far more complex than browser add-ons. The main difference between the two is that plug-ins provide extra functionality which does not modify the core functionality. It is a third-party program that is plugged-in to a web page and affects only the web page that is using the plugin. Examples of plugins would be Flash or a PDF reader. Essentially, the biggest difference between Plugins and Extensions is that plugins are standalone programs that integrate into Chrome whereas extensions are programs that can only be used in Chrome.

4.4.7 Library

Library: This is an entirely new subsystem we added to our architecture after having examined the source code using Understand. Along with this we have attached the dependency arrows to it to signify the fact that every other subsystem relies on Library to function (apart from Plugin as we did not have the code.) Library is essentially, as the name suggests, a collection of all the code which is used and shared by all other subsystems.

5.0 Limitations and Lesson Learned

Throughout the duration of this project, our group encountered multiple limitations that brought on many challenges that needed to be overcome.

- Understand is not a very stable program, and is very resource intensive. Many members of the group were unable to run the software on their PCs, which made it difficult to distribute work among us. Being a resource intensive program also meant it took a long time to save our work, sometimes upwards to 10 minutes, only for the software to crash and delete it. This cost us a lot of valuable time while working on our project and was a major problem for our team.

- The Understand software would often pick the wrong file as a dependency if there were multiple files with the same name and often crash when trying to inspect specific files. Thus we lost valuable time trying to navigate through the dependencies without causing the software to crash.
- Another limitation we encountered is that none of us had much experience with the C++ language which sometimes made understanding the source code difficult.
- Navigation was also a difficulty due to the large size of the program as it made it hard to trace dependencies and sequences of function calls through the code. Finally, since some source code had been purposely removed, it was impossible for us to verify Plugins dependencies.

That being said, we also made several mistakes and learned valuable lessons while working on this project.

- Despite initial difficulties, we figured out how to navigate the cs.chromium website by following where a method call goes, and where methods are called from.
- We realised that in an actual workplace environment, the software we would work with will be most likely be large and difficult to navigate. There may be poor documentation and bad coding practices which could hamper our work, so the skills we have developed to analyze source code will help us handle these situations.
- Code is not always as clean and perfect in reality as it is conceptually, so sometimes hacks are necessary to get the functionality working within the bounds of time or management constraints in the team.

6.0 Conclusion

In summary, Chrome's architecture is not a neat as we expected, but nonetheless met most of our fundamental expectations. Google Chrome is designed around an Object-oriented architecture, supported by implicit invocation for inter-process communication. The Browser component acts as the central subsystem, linking between all other subsystems and processing any requests which have been made. As discussed earlier, this is achieved with the aid of the four main modules: Kernel, Data Persistence, UI plus our newest component, Mojo. Chrome's sandbox architecture enforces strict security rules upon the browser ecosystem while promoting stability. The source code for Chrome is massive, which makes it impossible to review every piece of code as it would simply take too much time and effort. Instead, we decided that we needed to focus our efforts on smaller specific sections. There were many dependencies we did not expect, and even an entire component, which we called Library. Reviewing the actual source code of Chrome gave us a glimpse into the messy reality of software development, where not everything is as clear cut as in theory.

7.0 References

1. Chromium search (n.d.). Retrieved November 9, 2018, from https://cs.chromium.org
2. Core Principles. (n.d.). Retrieved October 18, 2018, from https://www.chromium.org/developers/core-principles
3. WebKit. (2018, September 17). Retrieved October 19, 2018, from https://en.wikipedia.org/wiki/WebKit
4. Chrome V8. (2018, September 24). Retrieved October 19, 2018, from https://en.wikipedia.org/wiki/Chrome_V8
5. Graphics and Skia. (n.d.). Retrieved October 18, 2018, from https://www.chromium.org/developers/design-documents/graphics-and-skia
6. Network Stack. (n.d.). Retrieved October 18, 2018, from https://www.chromium.org/developers/design-documents/network-stack
7. Chrome resource loading documentation. (n.d.). Retrieved November 8, 2018, from https://www.chromium.org/developers/design-documents/multi-process-resource-loading