The Concrete Architecture of Google Chrome

Assignment 3
November 30, 2018 (Fall 2018)

Thick Glitches
Alastair Lewis (15ahl1@queensu.ca)
Andrea Perera-Ortega (15apo@queensu.ca)
Brendan Kolisnik (15bak2@queensu.ca)
Jessica Dassanayake (15jdd1@queensu.ca)
Liam Walsh (15lcw1@queensu.ca)
Tyler Mainguy (16tsm@queensu.ca)

Abstract

The lack of censorship in Google Chrome for young children and work environments was recognized as a gap within the web browser. We proposed a Safe Mode, which would allow certain words and images to be censored simply by activating the feature with a password. A significant reason for the easy implementation of this feature is Chrome's object-oriented architecture style. Despite the fact that all five subsystems are affected by this feature, the architecture style allows for abstraction which makes it simple to add a new feature. Multiple approaches were weighed out to determine the best way to implement the feature, and SAAM analysis was used to assist with this. By comparing methods and looking at how each approach plays out in a use case, the most suitable approach was selected.

Introduction and Overview

Web browsers have the ability to be used by a broad range of users in a variety of environments. With our proposed feature, Safe Mode, we hope to specialize the user experience for certain demographics. The Google Chrome experience is currently fairly standard and uniform amongst the different users. The password-controlled Safe Mode is intended to censor content for users, such as young children, or in certain environments, such as in a workplace.

The feature would be located in the Content Engine within Libraries. The implementation of this feature would affect all five subsystems of Chrome: Browser, Content Engine, Networking, UI, and Storage. The object-oriented architecture style of Chrome encourages modularity, which allows the feature to easily be implemented while limiting coupling. The effects of Safe Mode can be seen further through the system's non-functional requirements. Furthermore, a deeper look is provided in this report at the impacts this feature would have on the current Chrome system and what would need to be changed.

We developed two approaches to introduce this feature into Chrome, and we used SAAM analysis to determine which approach was best. Sequence diagrams were also used to demonstrate how each of these approaches works. After gaining a solid understanding of the conceptual and concrete architectures of Chrome in the past, this knowledge was put to use when determining a well-motivated feature for Chrome and its proposed implementation.

Proposed Feature: Safe Mode

The proposed feature for implementation in Google Chrome is “Safe Mode” which aims to provide a safer and more appropriate browsing experience online. The implementation of this feature will benefit children, as well as professionals in work environments to ensure that inappropriate words and content are not shown. By enabling this feature within Chrome’s settings, instances of pre-set blacklisted words on a website or images and videos that contain these words in their metadata, file path, URL, etc., will be censored. One can enable and disable the Safe Mode with a password. This is especially useful for the use case in which parents can enable Safe Mode on their family computer if they have children who use it. Safe Mode utilizes all of the subsystems within Chrome, and affects three of the subsystems. The affected subsystems are Content Engine, Browser, and UI. Safe Mode will be located within the Content Engine subsystem, inside the Libraries subsystem. Additionally, the Content Engine subsystem is affected as it is where the HTML and CSS parsing occurs. After the parsing stage, inappropriate words and content will be censored. Unsuitable words will be checked after the HTML has been parsed into the DOM tree and unsuitable images will be checked for by analyzing their metadata. Any metadata that contains inappropriate words or originating from an inappropriate source will trigger the Safe Mode to censor the content. The UI system is affected because a new button icon will need to be displayed and implemented into the UI for the browser to toggle on/off. Finally, the Networking subsystem’s SMTP request feature will be used to send out the email reset message if a user requires a password change.

Feature’s Interactions with Other Features

Our plugin will make use of many of Chrome’s already existing features to function to its fullest. The features included in this section refer to features of chrome that are used by, but not affected by, Safe Mode. The UI’s settings page will be used for enabling and disabling the safe filter and, to enter the password to do so, it will use the UI’s prompt box feature. It will also use chrome’s data storage feature in the Storage subsystem, to store the email associated with the Safe Mode, as well as the corresponding hashed password. The Storage subsystem will also store the Safe Mode’s blacklist itself. In the event the user needs to reset their password, the networking subsystem’s SMTP request feature will be used to send out the email reset message.

Required System Changes

The major changes made to our architecture are made in the Content Engine, as that is where the parsing of the document and creation of the DOM tree is done. These changes are minimal due to the way in which we have implemented our feature. By creating the DOM tree before traversing it, we are able to preserve the existing modularity of Chrome, making the interface and component changes minimal (while also keeping the complexity of searching the same). The majority of the interfacing of our component will be done with the *dom_storage* directory, which is responsible for constructing the DOM tree after receiving data from a network request. Our actual component will be implemented in the *libraries* subsystem of the Content Engine, which already interfaces with the renderer subsystem (where *dom_storage* directory is). With this interface already existing, the required change to our system is to introduce a function call between the *dom_storage* directory and the *safe_mode* directory (our feature). This function call will only be executed after checking that our feature was enabled by the user. The call from the *dom_storage* node will pass an alias of the DOM tree object to the *safe_mode* component, which in turn will do a breadth-first search of the tree (as each node could contain illicit text). During the search, the Boyer-Moore string search algorithm will be used on the content of each node, along with the filename of any media. This search will be done on each node for each word that exists in our blacklist of words, which leads to a complexity of O(x*y*z) (where x is the number of nodes, y is the number of characters in the string, and z is the number of characters in the pattern). Each occurrence of these words will be replaced by the * symbol. After the transformation is completed on the object, *safe_mode* will return the altered DOM tree to the *dom_storage*, which will pass it along to be rendered into a bitmap. No other changes are required in the Content Engine subsystem in order to implement it.

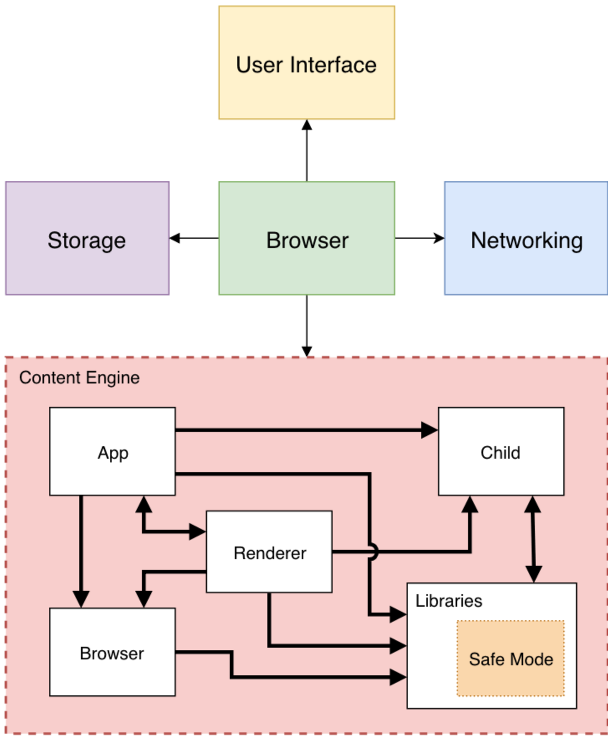

Figure 1. Required System Changes Within Content Engine's subsystem "Libraries"

Potential Impacts

Our feature is intended to be as modular as possible, and as a result, should have little impact on Chrome's high-level and low-level architectures. We managed to design our implementation of Safe Mode with only one architectural change, which takes place in the Content Engine subsystem. We have added a new subsystem, aptly named "Safe Mode", in the Libraries subsystem within the Content Engine. This new subsystem can be seen in figure 1. The new addition to our architecture should not produce any tangible impact to the existing functionality of the Content Engine subsystem, nor Chrome's architecture as a whole. Anything that interacts with the Safe Mode subsystem directly may be subject to unforeseen effects. However, we deem that this is not likely.

Testing Impact of Interactions with Other Features

Although the research for our proposal indicates that there will not be any interactions on the already existing features of Google Chrome[18], it is still our responsibility to verify that this is true. For every existing feature of Chrome used by Safe Mode, we will test every case of its usage with Safe Mode enabled, as well as one use case with Safe Mode disabled. For example, since Safe Mode makes use of Chrome's SMTP request capability, we will include a test case where safe mode utilizes it, and one where chrome utilizes it without relation to Safe Mode. For the single instance of code being changed, which is the function call between the *dom_storage* directory and the *safe_mode*, we will include a test case where chrome passes through this section with safe mode enabled, and a case where chrome passes through this section with safe mode disabled. Regardless of whether Safe Mode is enabled or disabled. With Safe Mode enabled, any text bodies or media file names containing a word that is part of the blacklist, should be censored. If Safe Mode is disabled, Chrome should behave as if Safe Mode is not installed, the only difference being dom_storage containing a boolean expression that evaluates to false if Safe Mode is enabled.

Effects of Enhancement

One of the advantages of the object-oriented architecture style the system is using is that the maintainability and testability of the system remain high after the feature implementation. The new proposed feature will have objects in the Browser, User Interface and Content Engine which can all be tested independently and maintained separately. For example, our UI object can be included in the UI tests, our Bowser object can be unit tested in the Browser tests and our Content Engine object can be included in the Browser tests. The filtering option is for each page specifically and the coupling is loose between the Browser and Content Engine. The design philosophy is to be consistent with the rest of Chrome and for each object to perform clear cohesive tasks in their respective subsystem. As for the overall performance of the Chrome system, after the feature implementation the performance should be high as each performance intensive task (the page filtering) is encapsulated in the Content Engine processes and thus if one page takes a long time to filter it is not a blocking operation for the rest of the application as the other Content Engine processes for other pages will not be affected. The object-oriented style is well regarded for evolvability as the knowledge of object implementations is not necessary for an object's use. The implementation of the feature may change over time (different algorithm in the Content Engine, different looking UI button, different settings etc.) but this will not change how the objects interact with each other. If completely new abilities are added to the objects of our proposed feature that also will fine as objects can inherit, override and new functions can be added but new code interacting with the objects may need to be introduced.

Potential Risks and Limitations

There are security risks as per the password storage. Passwords must be salted and hashed and not stored in cleartext like other Google Chrome passwords. There are also potential security risks for the account recovery system which enables a parent to get an email starting account recovery. Communication should be encrypted over https to protect the user's email from man in the middle attacks. This password recovery system is a maintainability risk as it must be maintained forever to support the proposed feature. The password management/recovery system will have to be hosted on a dedicated server. Also, to recover the password, the user will have to have an internet connection (required for browsing anyway). There is a slight performance risk to the initial rendering of the page due to the actual filtering algorithm we have designed but in practice, it should not be too dissimilar. The Breadth-first search is O(x) for the DOM tree where x is the number of nodes. The Boyer-Moore component is worst case O(y*z*1) for the text it is searching where y is the number of characters in the search pattern, z is the number of characters in the string and the number of blacklisted words it must check for is constant (relatively low). The replacement operation for each word is in-place in the string and thus constant. This gives the total worst case complexity of O(x*y*z). A limitation is that users will need an internet connection to reset their password but they will already require one to browse. The potential risks of the architectural changes are that in the Content Engine if our proposed feature does not work then the content displayed may be incorrect and damage the user experience. Some sites could even become non-functional. A performance limitation is that carefully conceived content that is illicit such as videos with no reference in their title or metadata to inappropriate content will not be blocked. Thus it cannot be totally 100% accurate at blocking out all bad content. This is also highly subjective as to what content should be illicit. Another technical limitation is that users cannot add words or phrases to be blacklisted. This is also for security reasons in case a Content Engine sandbox becomes compromised. The architectural changes also are cause for concern with security as it must be ensured that only Browser is accessing the Storage system. Content Engine must use IPC to go through Browser to access or store info into the Storage system.

SEI SAAM Analysis

Alternative Approach Analysis

Our alternative implementation is activated through the Chrome settings dropdown menu where a password hash is checked against a value contained in the storage module. This version of the implementation concurrently parses the DOM tree and filters out blacklisted content from a list stored locally in the Content Engine.

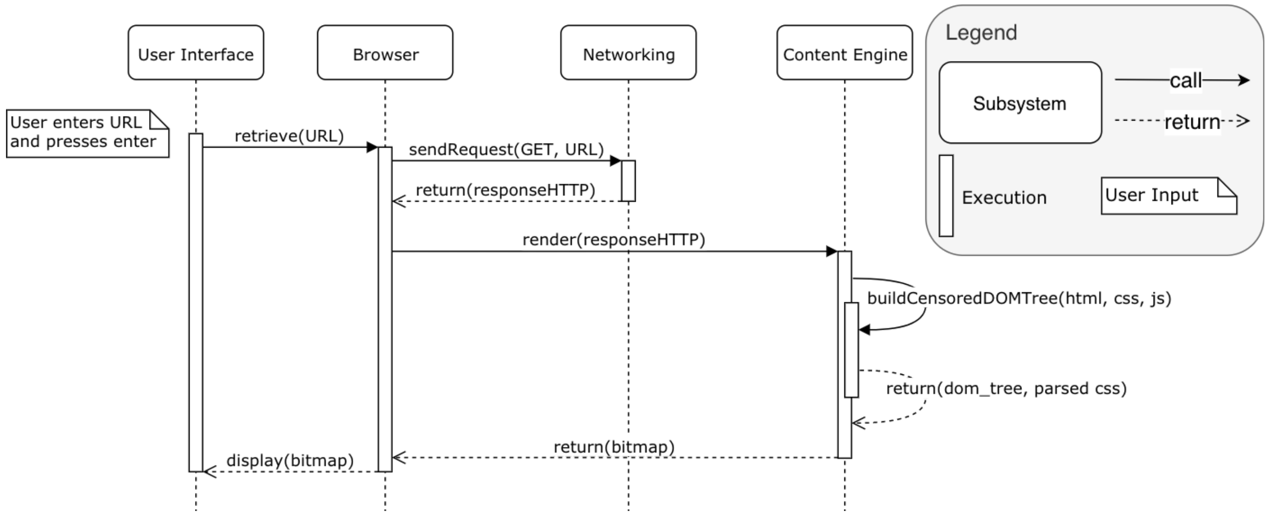

Figure 2. Loading a Webpage and Applying Filters (Alternative Approach)

The alternative approach to our proposed feature is depicted on the above sequence diagram. The diagram shows the flow of control during the loading of a webpage and filtering out blacklisted content. The sequence begins with the user submitting a URL in the UI. The URL is passed to the Browser which generates a GET request. The Browser sends this request to the Networking subsystem which resolves it and returns the HTTP response back to the Browser. That data is then passed to the Content Engine which begins the process of parsing the data and rendering it into a bitmap. During the parsing process, while each DOM tree node is being built, if the parser comes across a word contained in our blacklist, a list which is stored in the Content Engine, it will be censored by replacing each letter with the `*` character. The Content Engine will then forward the censored bitmap to the UI to be displayed to the user.

Proposed Approach Analysis

Our proposed approach differs from the alternative in several ways. The feature is activated with a button on the UI next to the URL bar. The user is prompted for a username and password which is checked against the values stored in the storage module. The username is necessary for password recovery functionality in the feature. This approach stores the blacklist in the storage module so that it is easier to modify the blacklist and add custom filters as per the user's needs.

The Google Development Team and the users are stakeholders. There were two key non-functional requirements from our SAAM analysis that affected the development team. For maintainability, the proposed approach is easier to maintain in terms of the actual filtering process because it is separate from other Chrome functionalities. In terms of reusability, the retrieval and filtering of the blacklisted content are in two distinct sections of the code that can be extracted and reused in another environment, if desired. While the users are stakeholders, they do not directly benefit from maintainability and reusability. That being said, they do benefit from other non-functional requirements, such as performance.

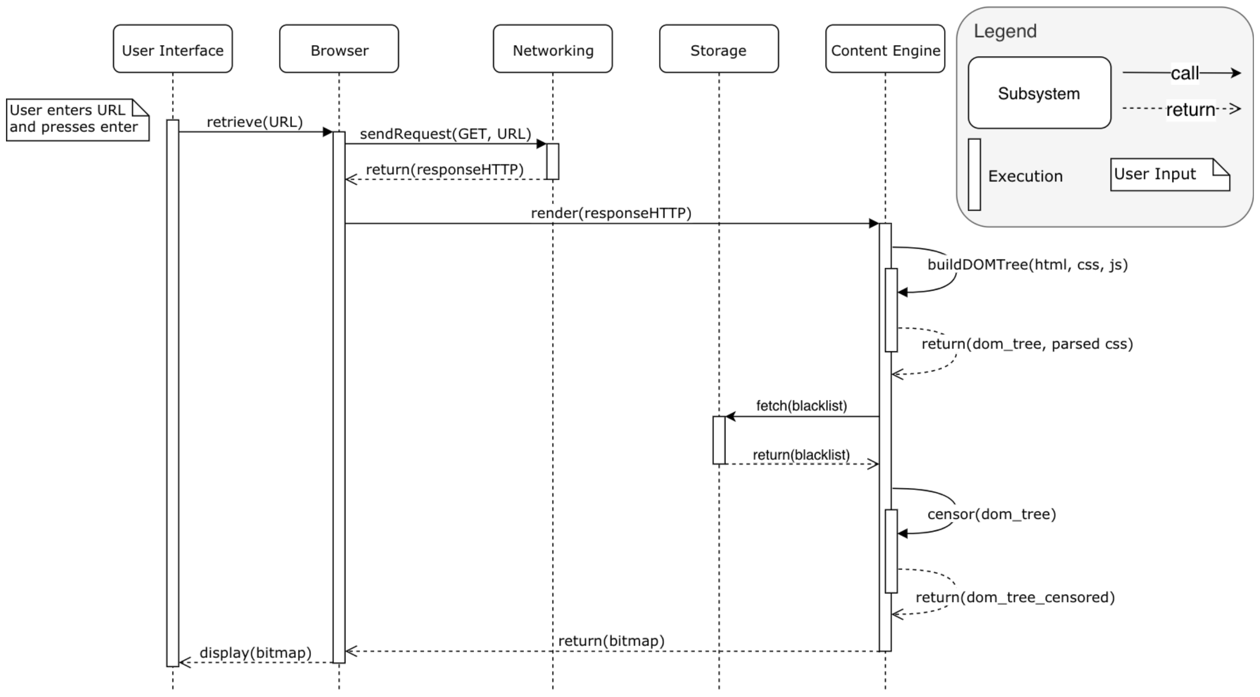

Figure 3. Loading a Webpage and Applying Filters (Proposed Approach)

The final approach to our proposed feature is depicted on the above sequence diagram. The diagram shows the flow of control during the loading of a webpage and filtering out blacklisted content. The sequence begins with the user submitting a URL in the UI. The URL is passed to the Browser which generates a request. The Browser sends the request to the Networking subsystem which resolves it and returns the HTTP data back to the Browser. That data is passed to the Content Engine which begins the process of parsing it and rendering it into a bitmap. First, the data is parsed into a complete DOM tree with no filtering applied to it. The Content Engine then queries the Storage subsystem for the current version of the blacklist containing all words to be filtered and their substitutions. The Storage subsystem returns the list to the Content Engine and iterates through the DOM tree with the Boyer-Moore algorithm to find indexes that need to be filtered. The values at these indexes are then censored by replacing them with the `*` character. After this process is complete, the Content Engine sends the rendered bitmap to the UI to be displayed to the user.

Decision

We chose to advance with the second approach (the proposed approach) over the original (alternative approach) for several reasons. The first is that the chosen implementation is more modular. It doesn’t modify any large sections of code already inside of Chrome, it only adds extra functionality. The second reason we chose this implementation is because the blacklist can be more easily modified as it is contained in the Storage subsystem which is easier for the user to access and edit than if they were stored within the Content Engine as originally planned. Lastly, the second approach is far easier to implement as it wouldn't involve modifying significant portions of Chrome’s source code, only adding components to it to facilitate the filtering process in the feature.

External Interfaces

There are multiple external interfaces that Chrome communicates with: GUI, file system, and network. The GUI allows the browser to process user input events and send visual output to the user by utilizing the UI. The file system allows Chrome to interact with the local files stored on the host machine. Chrome is limited to accessing the files that the operating system has given it permission to read and write to. The browser is able to interface with the network through the networking subsystem, which allows it to send and receive content that is not stored locally. Networking also allows the new proposed feature to communicate when a password reset has been initiated by the user.

Limitations and Lessons Learned

While carrying out the assignment, our team encountered and confronted a small number of limitations. Unlike assignment 1 and 2, we felt that the limitations we experienced were less inconvenient, as we already had the experience from completing the first two assignments to guide us through our troubles. As our semesters are coming to an end, everyone on the team was doing their best to juggle approaching deadlines and meetings from our other classes along with this assignment. Additionally, since this assignment slightly differed from the first two, we had to come up with new approaches for beginning this project and couldn't rely entirely on our experiences with the previous two assignments.

Despite the limitations, we were provided with valuable lessons. By working through the last two assignments as a team, we were able to recognize the strengths and weaknesses of each member and delegate components of the assignment accordingly. By learning this, we were able to establish a solid and smooth workflow from the start. Finally, this assignment gave us the opportunity to apply the information we learned during the course. Through our implementation of a new feature in Chrome, we gained practical and tangible experience.

Conclusion

Safe Mode enhances the current Chrome system by adjusting viewable content. This feature would be beneficial to parents since it gives them the ability to filter the content accessible to their children. It would also be helpful in work environments if managers want to limit the explicit content their workers can access on the job. Safe Mode utilizes all of the subsystems, but the majority of the low-level effects exist mainly within the content engine because this is where the feature resides. The beauty of implementing new features into Chrome is that the architecture is object-oriented, so abstraction makes it easy to enhance the system without increasing the coupling too much. Through SAAM analysis, we were able to determine the best approach. We compared two possible approaches and decided on our final one because it worked better than the alternative in terms of maintainability and reusability. The modular nature of the approach makes it easy to implement the feature into Chrome without affecting the existing code too much. For these reasons, we believe that Safe Mode is a useful feature to add to Chrome that would not negatively impact the current system.

Data Dictionary/Naming Conventions

Bitmap: A mapped output generated by the rendering engine to be displayed in the user interface graphically.
Blacklist: A list of terms or values that have been deemed as undesirable and are flagged in our feature as inappropriate.
Breadth-First Search: Tree searching algorithm that iterates through the tree by exploring one depth level at a time before proceeding to the next.
Browser: The main module in the program that controls data exchange across modules in the application.
Content Engine: The subsystem within our architecture that handles everything that can be displayed in the content pane of Google Chrome.
Cohesion: The level of separation of functionality between modules in the software. A highly cohesive system will have modules that perform very specific tasks tailored to their attributes.
Concurrency: Multiple software processes running at the same time by utilizing multiple processing cores.
Coupling: The number of dependencies your program has in between its various modules. It is optimal to have low coupling in your system so that if one module malfunctions there is minimal effect on the other modules in the architecture.
CSS: (Cascading Style Sheets) Lightweight styling language that directly interacts with HTML to allow web pages to have custom appearances and layouts.
DOM Tree: (Document Object Model) The programming interface used to convert HTML into a tree structure where each node represent a component of the document. It is used to parse and render web pages.
GET Request: A request that is sent out into the internet with the intention of fetching a response containing data.
Hash: A unique string of characters that represents a particular piece of data that has been run through a Hashing Function. It is used for encryption purposes as well as sorting and searching algorithms.
HTML: (Hypertext Markup Language) The format in which web pages are written in. Uses text with 'tags' surrounding them to describe how they should look and appear on the page.
HTTP: (Hypertext Transfer Protocol) The protocol used by the internet to send text data across the web from servers to clients.
IPC: (Interprocess communication) Set of programming interfaces that allow for communication and coordination between software processes running concurrently.
JavaScript: An event-driven programming language that allows web pages to be interactive and have features that would otherwise be impossible with only HTML.
Metadata: A set of data that provides information about another body of data.
Modularity: The process of subdividing software into individual components. It results in more understandable code and allows for changes to individual modules without having it affect all others.

Multi-Processing: Software that is designed and optimized to run on computers with multiple processing cores.
Network: Our architecture's subsystem that handles the browser interfacing with the Internet to retrieve content not stored locally on the host machine.
Object-Oriented Architecture: Architecture style with multiple, single-purpose 'Objects' that have interfaces to interact with other objects in the system. Objects are not concerned with the implementation of the other objects in the system, only with the data getting passed to them.
POST Request: A request that is sent out into the internet with the intention of sending data to a remote server.
Sandbox: The programming practice that segregates a certain set of functionality to a restricted amount of resources, file access, and operating system interactions on the computer in order to increase security. This way, it the sandboxed portion of the application crashes or is compromised, it will have no serious effect on the rest of the system.
SAAM Analysis: (Software Architecture Analysis Method) A process used for making architecture modifications or additions that involve comparing multiple versions of the same implementation and deciding which is the optimal version to continue with.
Storage: The subsystem in our architecture that handles file I/O with the host machine. It provides functionality to handle data persistency.
UI: (User Interface) The graphical representation of the application that interacts with the user to show/get input and output from them.
URL: (Uniform Resource Locator) A string of text that identifies a specific web page on the world wide web. These are used by the web browser to retrieve specific pages requested by the user.

References

[1] "Blink." *The Chromium Projects*, www.chromium.org/blink.
[2] "Desktop Browser Market Share Worldwide." *StatCounter Global Stats*, gs.statcounter.com/browser-market-share/desktop/worldwide/#monthly-201801-201810-bar.
[3] "Google Chrome." Wikipedia, Wikimedia Foundation, 16 Oct. 2018, en.wikipedia.org/wiki/Google_Chrome.
[4] "Inter-Process Communication (IPC)." *The Chromium Projects*, www.chromium.org/developers/design-documents/inter-process-communication.
[5] "Multi-Process Architecture." *The Chromium Projects*, www.chromium.org/developers/design-documents/multi-process-architecture.
[6] "The Security Architecture of the Chromium Browser". Barth, A., Jackson, C., Reis, C., & Google Chrome Team, 16 Oct, 2018. http://seclab.stanford.edu/websec/chromium/chromium-security-architecture.pdf
[7] "Multi-Process Architecture" *Chromium Blog*, https://blog.chromium.org/2008/09/multi-process-architecture.html
[8] "Network Stack", *The Chromium Projects,* https://dev.chromium.org/developers/design-documents/network-stack
[9] "Disk Cache", *The Chromium Projects* https://dev.chromium.org/developers/design-documents/network-stack/disk-cache
[10] "HTTP Cache", *The Chromium Projects* https://dev.chromium.org/developers/design-documents/network-stack/http-cache
[11] "CookieMonster", *The Chromium Projects* https://dev.chromium.org/developers/design-documents/network-stack/cookiemonster

[12] “Why is Chrome Using So Much RAM? (And How to Fix it Right Now). Albright, Dann, 17 June, 2017 https://lifehacker.com/why-chrome-uses-so-much-freaking-ram-1702537477

[13] “Explore the Magic Behind Google Chrome”. Zeng, Dico, 21 Mar, 2018 https://medium.com/@zicodeng/explore-the-magic-behind-google-chrome-c3563dbd2739

[14] “Which WebKit Revision Is Blink Forking from?” Google Groups, Google, 18 Apr. 2013, https://groups.google.com/a/chromium.org/forum/#!msg/blink-dev/J41PSKuMan0/gD5xcqicqP8J.

[15] “Understanding How the Chrome V8 Engine Translates JavaScript into Machine Code.” FreeCodeCamp.org, FreeCodeCamp.org, 20 Dec. 2017, https://medium.freecodecamp.org/understanding-the-core-of-nodejs-the-powerful-chrome-v8-engine-79e7eb8af964.

[16] Kosaka, Mariko. “Inside Look at Modern Web Browser (Part 1) | Web | Google Developers.” Google, Google, Sept. 2018, https://developers.google.com/web/updates/2018/09/inside-browser-part1.

[17] “Life of a URLRequest”
https://chromium.googlesource.com/chromium/src/+/master/net/docs/life-of-a-url-request.md?fbclid=IwAR0k_r8KKCXzXAaRRV_C7xZASVR3cX-IHmh0MK4wyHLvxwea5wj25uTOIlI

[18] “Code Search”, Google Chrome. https://cs.chromium.org/