Chrome: Enhancement Proposal

Assignment 3 – Report

Dec 4th, 2018

Mackenzie Furlong – 15mwf1@queensu.ca
Alex Golinescu – 16ag16@queensu.ca
David Haddad – 16dh10@queensu.ca
William Melanson-O'Neill – 16wmon@queensu.ca
Michael Reinhart – 15mr34@queensu.ca
Lianne Zelsman – 13lkz1@queensu.ca

Abstract

The team was to propose a new feature to enhance the current Chrome browser experience. The team decided to maintain the same concrete architecture from the previous assignment—with the networking, rendering engine, user interface, data persistence and browser engine subsystems—and implement a facial recognition feature to improve efficiency and security within the browser. The feature's main two use cases are to protect the Chrome autofill functionality and allow users to bypass Chrome-prompted login requests with face confirmation.

Two different designs were initially suggested for the feature's implementation. The first design required no addition subcomponents or dependencies, but rather made substantial changes to the data persistence, browser engine and user interface subsystems to introduce the facial recognition functionality. The second design was built around preexisting functionality and added a new dependency between the data persistence and rendering engine subsystems, as well as creating a new subsystem to perform the computer vision portion of the feature.

A SAAM analysis was done on both designs to determine which one the team should implement. The analysis criteria were based around the non-functional requirements of the main stakeholders—users, Google and the developers. The maintainability, testability, security, evolvability and performance was evaluated for both designs. The large benefits to security, as well as the developmental bonuses from largely using preexisting functionality, led the team to decide on the second design.

The new feature would mainly operate through the rendering engine by using the Media Stream API and Autofill Manager framework which currently control webcam access and autofill functionality, respectively. The implementation is expected to have a low impact on the high-level system, as only evolvability is anticipated to be negatively impacted due to a slight increase in system coupling. On a low-level, the only impact to components will come from the data persistence and rendering engine subsystems requiring slight modifications to allow for their new communication channels. The main risks of the enhancement are security and performance, both of which should be tested thoroughly through various test cases.

The team had to make various assumptions about the current functionality of the webcam and autofill components, as the official Chromium documentation was outdated and difficult to piece together. This may have led to inaccuracies in the assumed architecture of the preexisting functionalities. The team also realized the large impact that even seemingly small changes can have on an entire system if not careful.

Introduction and Overview

This report discusses the new facial recognition feature proposed by the team as an enhancement for Chrome—a web browser built by Google based on the open-source Chromium project. The design of the concrete Chrome architecture was developed and designed in Assignment 2.

The report explores the process of deciding on a design for the implementation of facial recognition. Various non-functional requirements are taken into consideration based on the needs of stakeholders. The Architecture section provides details on the entire SAAM analysis that was performed between two design options, as well as an in-depth explanation of the implementation, impact, risks and testing for the chosen architecture.

The External Interfaces section provides details surrounding the data storage system, including how a user's facial recognition data is obtained and stored. Two use-cases—one for facial recognition protecting and auto-filling a form and another for facial recognition allowing a user to bypass the Chrome login—are walked through in the Use Cases section. Sequence diagrams are provided for clarification on how the major components interact with each other and the user. The Data Dictionary section has a glossary that briefly defines key terms, and the Naming Conventions section outlines the acronyms used throughout the report.

The Conclusions section discusses the difficulty the team had when researching the preexisting Chrome functionality and all the assumptions that had to be made regarding the design. The Lessons Learned section focuses on the realization of how small changes to a design can have large impacts on an entire system. References are provided at the end of the document. Readers looking for more details about the studied Chrome architecture may use the provided links to receive lower-level explanations from the Chromium Project documents.

Architecture

Concrete Architecture

The five main components in the concrete architecture, each organized as a separate object in the system, have remained the same from Assignment 2. They can be seen in
Figure 1: Concrete Architecture.

- **Networking:** interacts with the rendering engine to connect to the internet using FTP and HTTP. It translates different character sets and implements a cache for retrieved resources.
- **Rendering Engine:** interacts with the networking system and browser engine to accept HTML and CSS and parses it into information that can be read by the UI (it prepares the DOM). Blink is used as the rendering engine.
- **User Interface (UI):** presents the page and its features; it is how the user interacts with the browser.
- **Data Persistence:** works with the browser engine to collect and record continuous data and information from users.

- **Browser Engine:** runs the UI and manages the tab and plugin processes. Delegates HTML and CSS code to be sent to the rendering engine(s) for parsing.

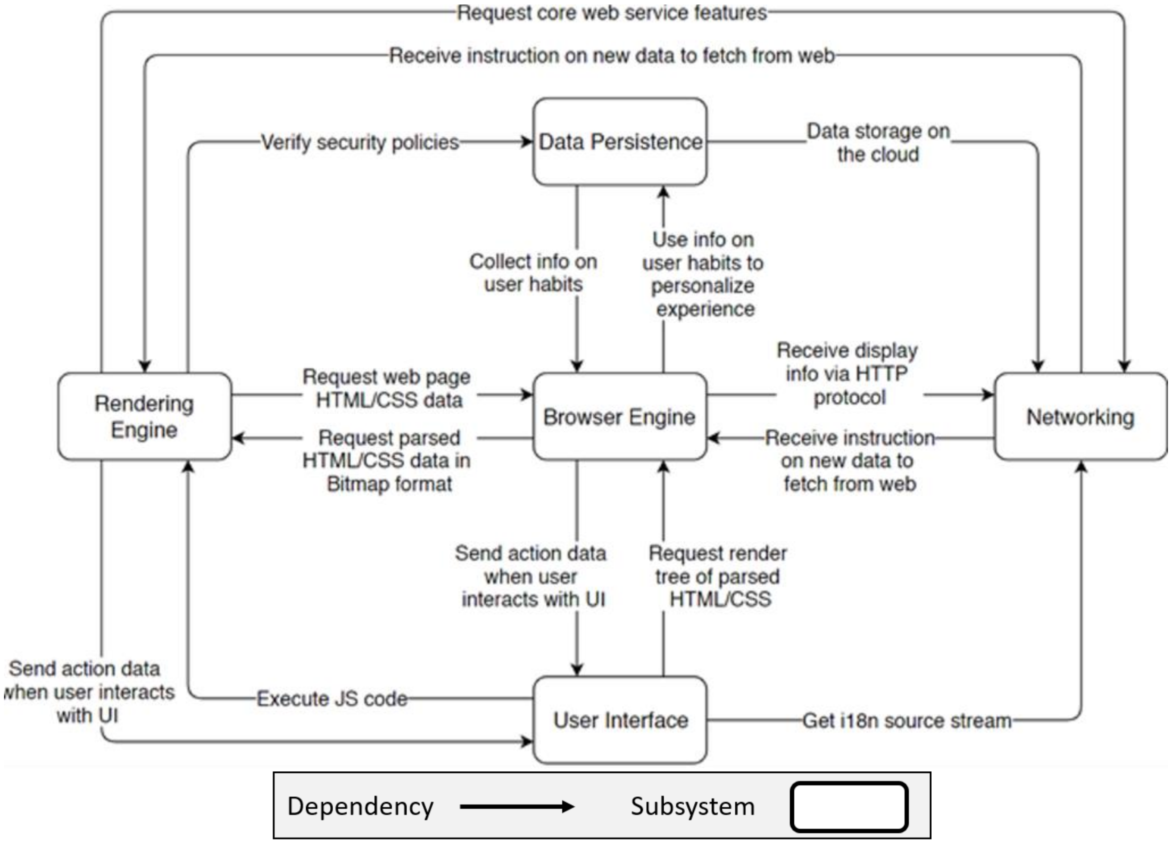
Request core web service features
Receive instruction on new data to fetch from web
Verify security policies
Data Persistence
Data storage on the cloud
Collect info on user habits
Use info on user habits to personalize experience
Rendering Engine
Request web page HTML/CSS data
Request parsed HTML/CSS data in Bitmap format
Browser Engine
Receive display info via HTTP protocol
Receive instruction on new data to fetch from web
Networking
Send action data when user interacts with UI
Request render tree of parsed HTML/CSS
Send action data when user interacts with UI
Execute JS code
User Interface
Get i18n source stream
Dependency
Subsystem

Figure 1: Concrete Architecture

Feature Proposal

The team proposed a facial recognition feature that can do the following two things:

1. Allow a user to bypass Chrome-prompted logins, such as when logging into their Chrome account or viewing saved passwords.
2. Allow a user to protect autofill data, so that online forms will only be automatically completed with pre-saved data if the user first verifies their identity with facial recognition.

The facial recognition enhancement will have two main values. By allowing a user to quickly bypass Chrome-prompted login requirements without needing to type in a password, it will improve the overall efficiency of the login process. By adding an extra layer of protection to the preexisting autofill functionality, it will maintain the user experience benefit of the autofill functionality while increasing browser security by preventing accidental or malicious use of the user's saved data.

SAAM Analysis

Stakeholders and Non-Functional Requirements

The team decided to focus the SAAM analysis on the key non-functional requirements that were determined to be the most important for the three main stakeholders:

1. The ***users*** are primarily concerned with system performance to ensure the feature does not negatively impact their user experience, as well as the security of their account data.
2. The ***Google company*** is primarily concerned with the legalities of the feature's security, ensuring that users' personal data is not misused or endangered.
3. The ***developers*** are primarily concerned with the feature's evolvability, maintainability and testability, each of which will directly impact their work.

Design One

The team's first design option does not involve the addition of any new subsystems or dependencies. Instead, it adds new functionality to the browser engine, UI and data persistence object, isolating most feature usage to those three subsystems. The user's preferences and facial data will be stored in the data persistence object. The webcam functionality is built into the UI object and controlled by the browser engine, which performs the required image processing algorithms. The design can be seen below in Figure 2: First Feature Implementation Option.

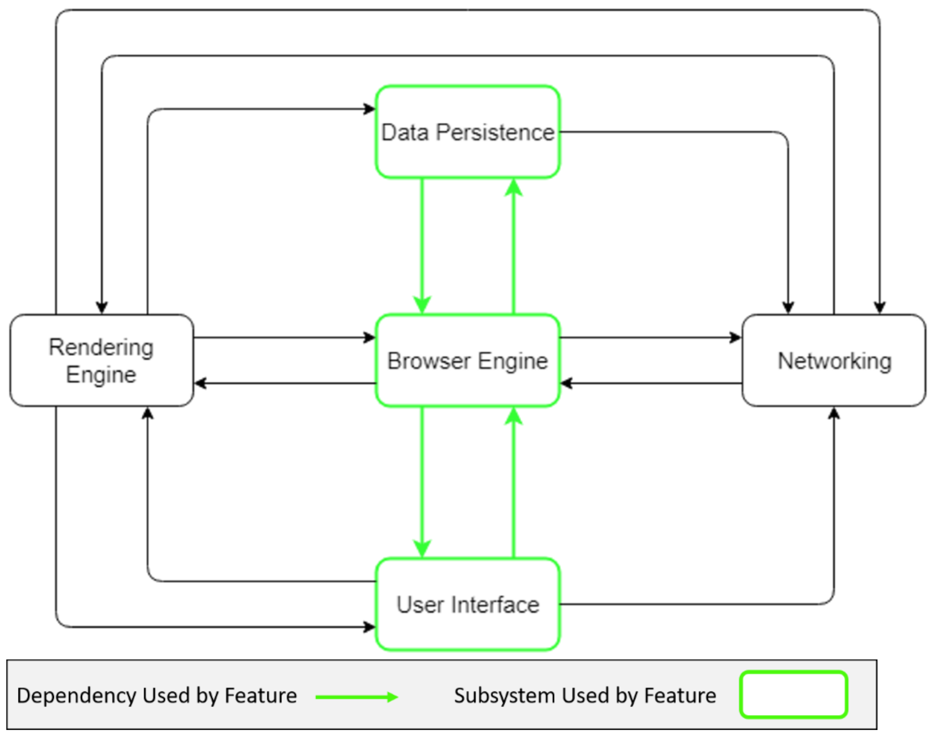

Figure 2: First Feature Implementation Option

Design Two

The team's second design option builds off the preexisting autofill and webcam functionality within Chromium. Most of it is controlled by the rendering engine through the Media Stream API [1], which communicates with both the data persistence object and UI. This design would require adding a new dependency between the rendering engine and storage to allow for the API to function correctly. It also adds in a new "Computer Vision" subsystem, which performs the computer vision algorithms for the facial recognition. This design can be seen below in

Figure 3: Second Feature Implementation Option.

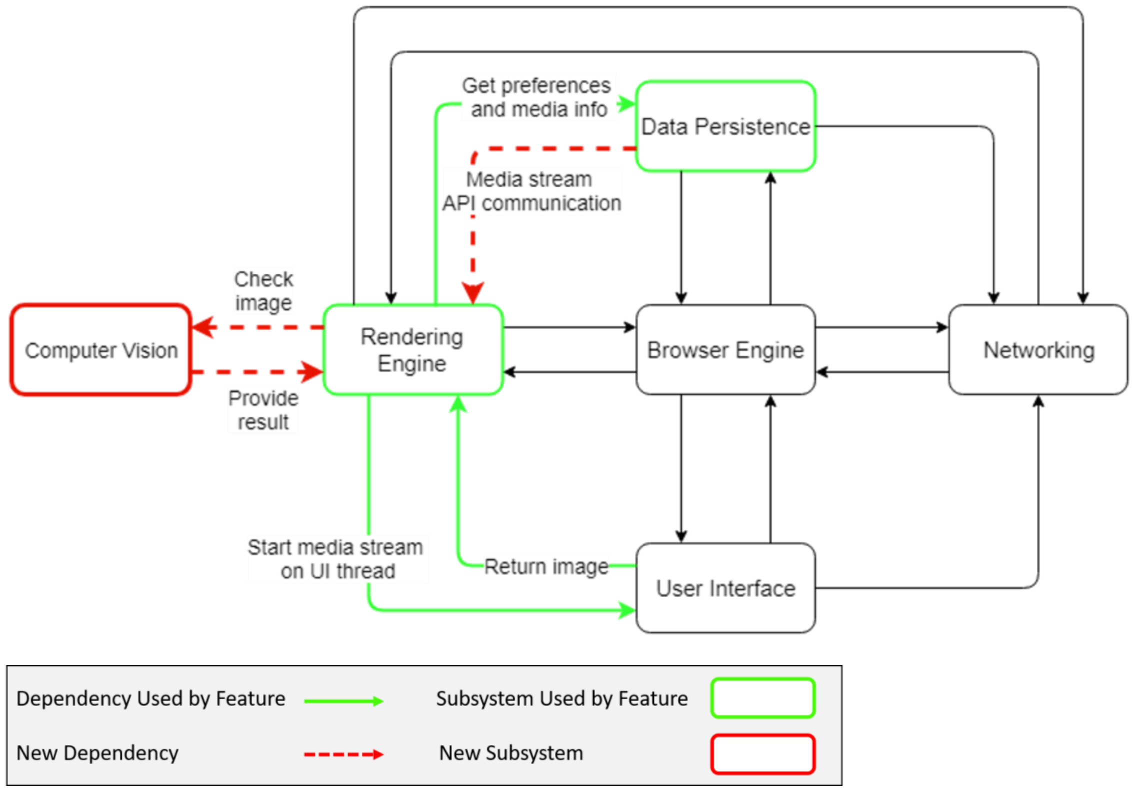

Figure 3: Second Feature Implementation Option

Advantages/Disadvantages

The team analyzed the advantages and disadvantages of the two design options based on the main non-functional requirements of the stakeholders. In Table 1: Design Option Advantages and Disadvantages below, advantages can be seen in green, with disadvantages in red.

Table 1: Design Option Advantages and Disadvantages

	Design One	Design Two
Security	• Need to build new security measures to ensure data security and proper authentication	• Preexisting security built into API • Use of rendering engine ensures individual threads used for each tab, so facial recognition must be confirmed for each new site
Maintainability	• Added redundancy due to recreation of similar features that already exist elsewhere can clutter code	• Easy to adjust computer vision algorithms separately • No difficulty to maintain existing functionality
Testability	• Decreased cohesion of subcomponents can make	• Components can be tested individually

	testing individual parts more difficult	• Most of the required unit tests already exist for the API and autofill component
Evolvability	• Adding more functionality to existing subsystems decreases cohesion. This decreases evolvability as changes to existing subsystems can risk impacting the feature	• Increased coupling can make evolution difficult • However, having functionality in increased number of subsystems allows parts to be changed individually
Performance	• Less subsystems and required connections, so less chance of bottlenecking or chocking	• Media Stream API is efficient • More subsystems and required connections can increase chance of bottlenecking
Other	• More work to develop	• Less work to develop

As can be seen from the table, Design Two is anticipated to be superior in most categories. Building off existing functionality provides numerous benefits to developers, including easy testability, evolvability and maintainability. The added security benefit of running the feature through the rendering engine is also a significant advantage. Thus, the team decided to go with Design Two.

Implementation

The facial recognition feature will mainly be centered through the rendering engine, which currently enacts the Media Stream API used for webcam access, and the autofill functionality. When a user comes across a webpage form that could potentially be auto-filled, the UI gesture listener notifies the rendering engine to instruct it to start the facial recognition process. The rendering engine uses the API to tell the data persistence subsystem to check the user's preferences to ensure that the facial recognition options are enabled and that the autofill functionality and webcam access are allowed. It also retrieves the user's stored facial data, which will be used to verify the user's identity.

The rendering engine tells the UI to start a new UI thread for the webcam process, and then passes the user's image captured by the webcam to the new computer vision subsystem. The computer vision component contains the intelligence and algorithms required to confirm or deny the user's identity based on their stored facial data and the facial data captured by the webcam. The computer vision algorithms should be developed by the Chrome team to optimize accuracy and performance, unless a suitable third-party computer vision tool is tested and confirmed as sufficient. If the user's face is confirmed a match, the system will then proceed to either auto-fill a form (using the AutofillManager feature) or bypass a Chrome login screen, depending on the scenario [2].

Impact

High-Level

As most of the feature is implemented using preexisting functionality, the enhancement should not have a major impact on the non-functional requirements of the overall system. Aside from a potentially reduced evolvability due to higher coupling caused by the additional dependency, the maintainability, testability and performance of the system are expected to remain the same. The higher-level architecture will still use the object-oriented design discussed in Assignment 2.

Low-Level

There should be minimal impact to the system's internal components, as the main functionality of each subsystem remains the same. The subsystems will still mainly have an object-oriented design internally, with the rendering-browser engine system using multiple application layers and the UI using an implicit style for user events. The only required additions to the existing subsystems are to allow the data persistence object the ability to communicate back to the rendering engine, as well as the two-way communication between the rendering engine and the new computer vision subsystem. Since this new communication will build off the efficient Media Stream API system, there should be no noticeable impact to system performance. Some directories that will require changes are *content/browser/renderer_host/media/*, *content/renderer/media/* and *content/public/renderer*.

Concurrency

The Main/UI thread is responsible for rendering web pages on the screen, while the I/O thread takes care of IPC communication between the browser process and render process, as well as any network communication. Resource requests—e.g. for web pages—can be handled entirely on the I/O thread and do not block the user interface. This process can be seen below in Figure 4: Browser-Render Process.

.

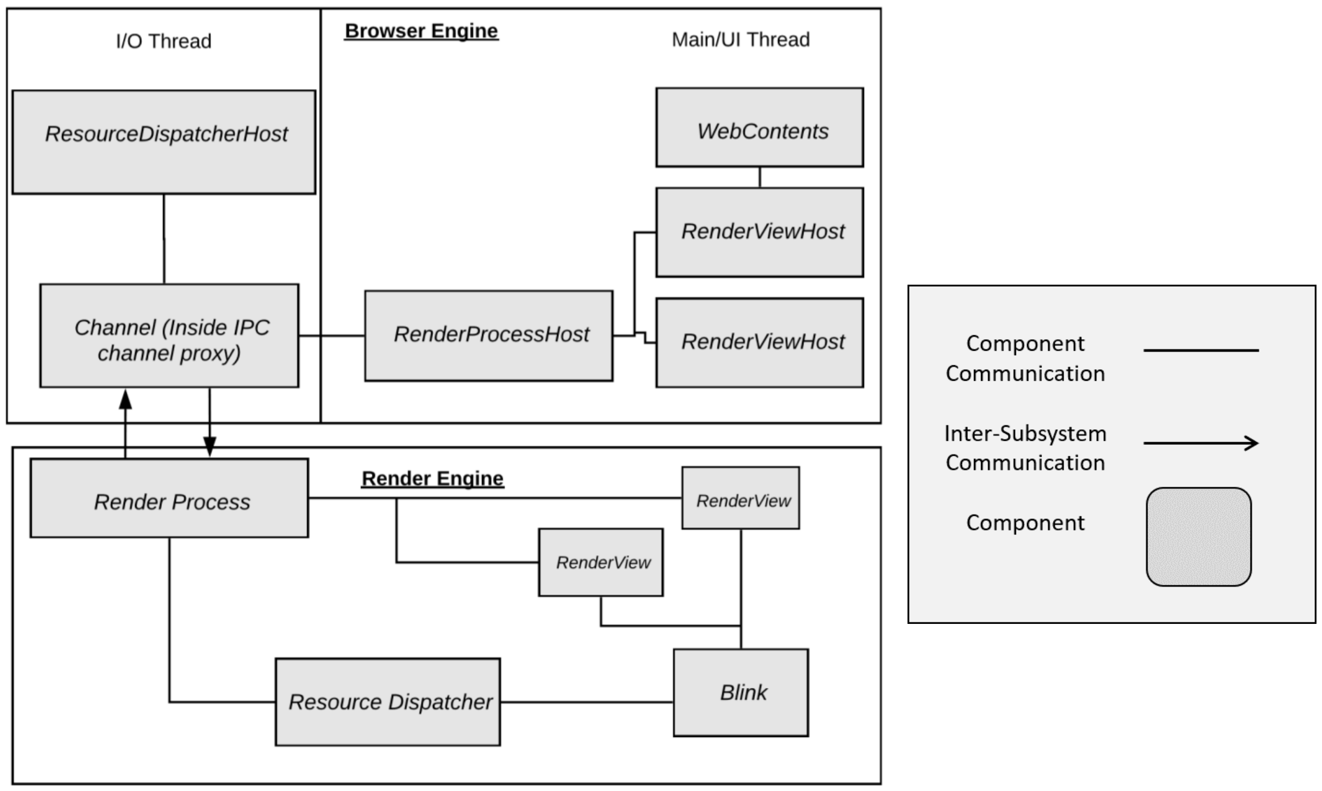

Figure 4: Browser-Render Process

With each browser tab running on a separate renderer thread, all webpages operate concurrently. The concurrency provided by the system's architecture introduces multiple benefits to the feature's implementation. It ensures that there are no performance bottlenecks between different tab processes and that errors in one tab will not impact the login or auto-fill success for another. The separate instances of the rendering engine also improve the overall security of the feature, as each webpage requires the user to complete the facial authentication process (rather than only requiring a single authentication for the entire browser session).

Each webcam instance is run on its own UI thread, so the facial recognition functionality can operate concurrently from the rest of the browser. The process will not disrupt currently open webpages or prevent the user from performing separate actions within Chrome.

Risks

The inclusion of the new feature has a few risks, mainly to the system's performance and security. If the facial recognition algorithm is not efficient, it could end up taking much longer than entering a password, which greatly reduces usage speed and renders the enhancement useless. An inaccurate facial recognition algorithm that results in false positives would allow strangers to maliciously log in and access a user's personal data, creating a large security hole.

Testing

Multiple tests need to be run to ensure the risks discussed above are mitigated and that the enhancement functions correctly when interacting with pre-existing features. The following are five test cases that should be used by the development team:

1. Test that the completion of a facial recognition authentication correctly automatically fills a webpage form.
   - This test will confirm the overall functionality of the new changes interacting with the existing autofill and webcam features.
2. Test the performance of the facial recognition algorithm. This test can be done by using 100 different faces, at various angles and lighting conditions.
   - The recognition feature should have a response time of less than two seconds 95% of the time. This will ensure the face-matching technology performs well enough to provide an efficiency benefit to the user experience.
3. Test the accuracy of the facial recognition algorithm. This test can be done by using 100 different faces, at various angles and lighting conditions.
   - The recognition feature should have > 95% success rate at identifying individuals that match the saved facial image (< 5% false negative rate is important for usability)
   - The recognition feature should have > 99% success rate at denying individuals that do not match the saved facial image (< 1% false positive rate is important for security)
4. Test that the facial recognition feature is properly enabled or disabled based on a user's preferences in their settings.
   - This checks that the UI and rendering enhancement is correctly interacting with the existing functionality of the data persistence object, which stores user preferences.
5. Test that the facial recognition process responds correctly to errors and system failures.
   - For example, the system should not allow a user to bypass the recognition stage if internet connectivity is lost during the process.
   - This test verifies that the new feature is correctly interacting with the existing failure procedures and security checks.

Team Issues

The fact that the new feature's implementation involves multiple subsystems makes it difficult for a single development team to implement it alone. The development would require cross-collaboration between multiple different teams. However, as the majority of the feature's functionality is centered within the rendering engine subsystem, it is recommended that the rendering team take on the development of the enhancement. They would need to consult with members of the UI and data persistence teams to ensure that each part of the new feature functions properly within those subsystems.

External Interfaces

Storage and Databases

The data persistence subsystem has a dependency on the networking subsystem as Chrome stores some of the user's data in the Google cloud storage system, as well as locally on the user's device in */Users/<user>/AppData/Local/Google/Chrome/UserData/* as an SQLite

Database. The user's facial image is stored locally on the user's device for privacy reasons, which means the user will need to replicate the facial recognition setup process on each device they wish to use the feature on. The user's preferences and autofill data are stored both locally and securely within the cloud so that the data can be shared between devices for convenience. This means that the user can fill in their username and password once on a device and have this password stored and available on all their associated devices. Unlike the user's facial image which only needs to be set up once on each device, a user's autofill data needs to be constantly updated to remain accurate and useful.

The data persistence subsystem is made up of various components, which are described below:

- **Plugins**: information on what plugins are used by the user.
- **Chrome Generated User Model:** builds a model of the user through the collection and manipulation of their usage data. This model is stored in the cloud to be used by Chrome to tailor the user's experience and sold to third-parties for commercial purposes. Data analytic techniques are done to predict future behavior of the user.
- **Browser History:** keeps a record of previous search history of the user. This information is stored both locally, as well as in Google's cloud storage.
- **Cookies:** records personal information, such as preferences for specific webpages. This information is stored locally in an SQLite Database.
- **Cache:** files, images, and sites are saved in cache memory for faster retrieval (stored locally).
- **Autofill Data:** user information (such as names, addresses, phone numbers) that are stored locally, as well as in Google's cloud storage.
- **Facial Data:** a user's facial image data used for the facial recognition feature. This is stored locally in an SQLite Database.
- **User Preferences:** the user's settings, security policies and feature preferences, stored locally, as well as in Google's cloud storage.
- **SQLite Database:** used for local storage on a user's device.

The interactions between these components and other subsystems can be seen in Figure 5: Data Persistence Subsystem Diagram. The new communication channel from the data persistence system to the rendering engine is present.

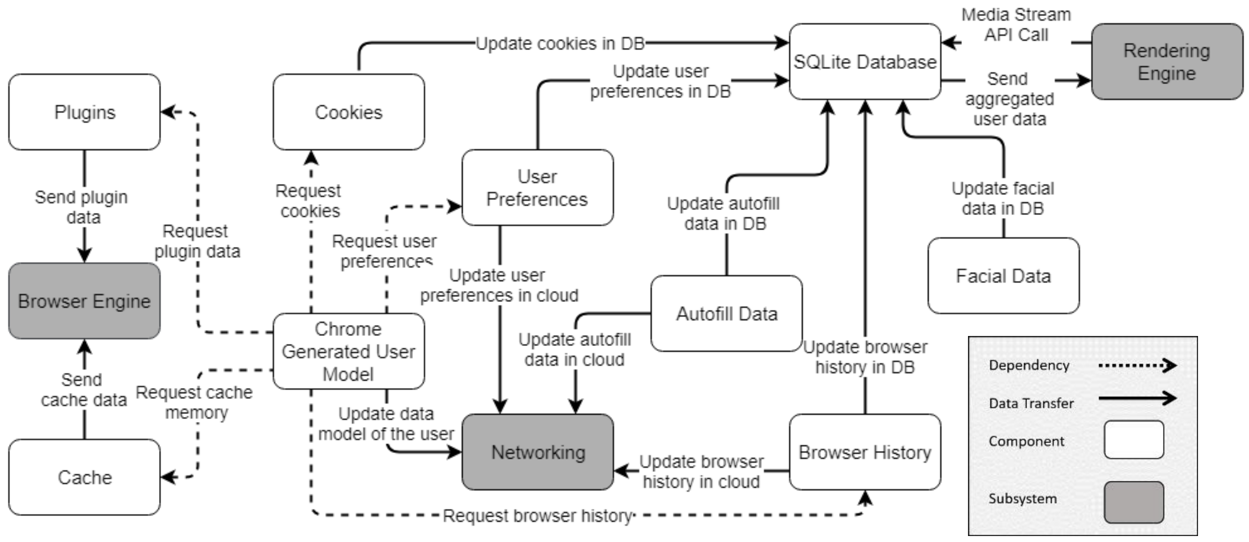

Figure 5: Data Persistence Subsystem Diagram

Use Cases

The sequence diagram in Figure 6: Autofill Sequence Diagram describes how the team's proposed architecture would autofill a form on a webpage after authenticating with facial recognition. The diagram assumes the user has already enabled the facial recognition autofill protection in their user preference settings.

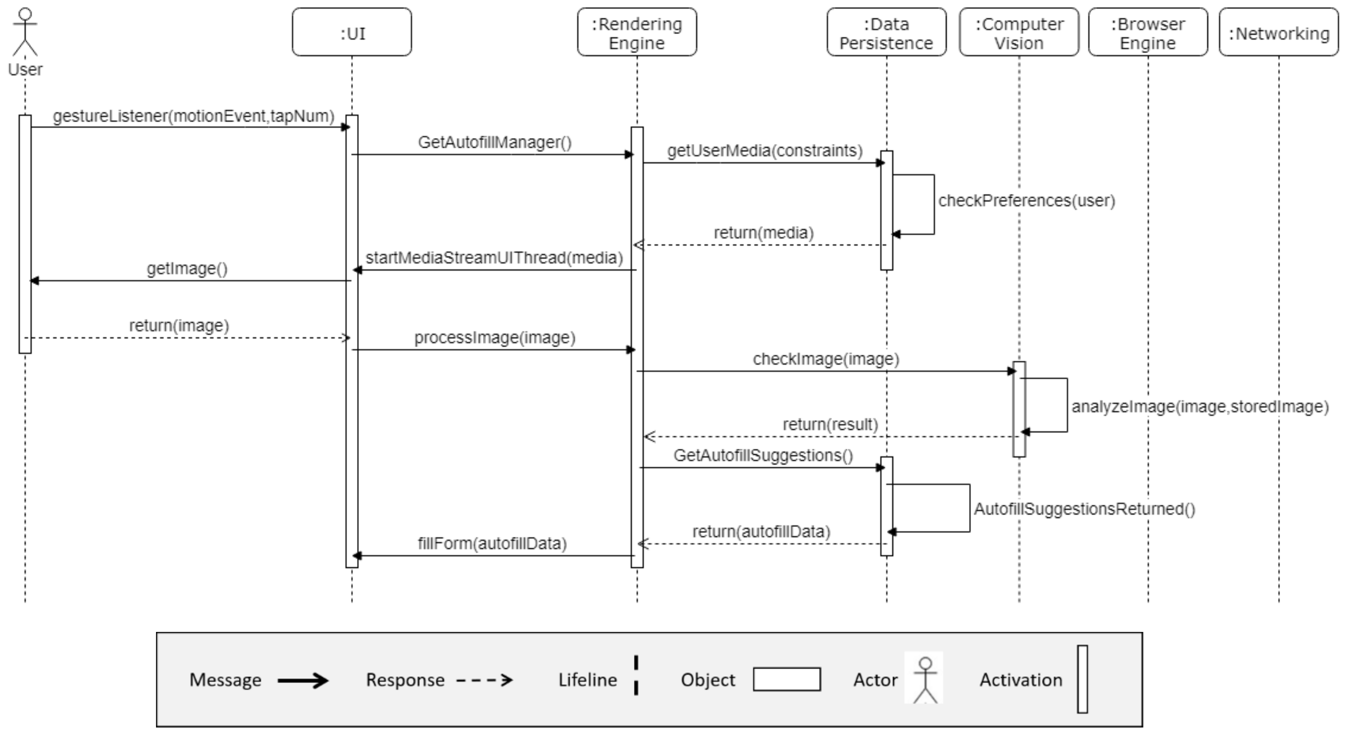

Figure 6: Autofill Sequence Diagram

In the above diagram, please note the placement of the “AutofillSuggestionsReturned()” function call which happens within the data persistence subsystem. This is different from how the sequence diagram was displayed during the team’s presentation, as it was realized that the specific function call does not go to the rendering engine. Instead, it happens internally within the storage system and the results are returned to the rendering engine as a response. No dependency originally existed from the data persistence subsystem to the rendering engine.

The sequence diagram in Figure 7: Chrome Login Sequence Diagram describes how the team’s proposed architecture would allow a user to bypass the Chrome login page with facial recognition. The diagram assumes the user has already enabled the facial recognition login option in their user preference settings.

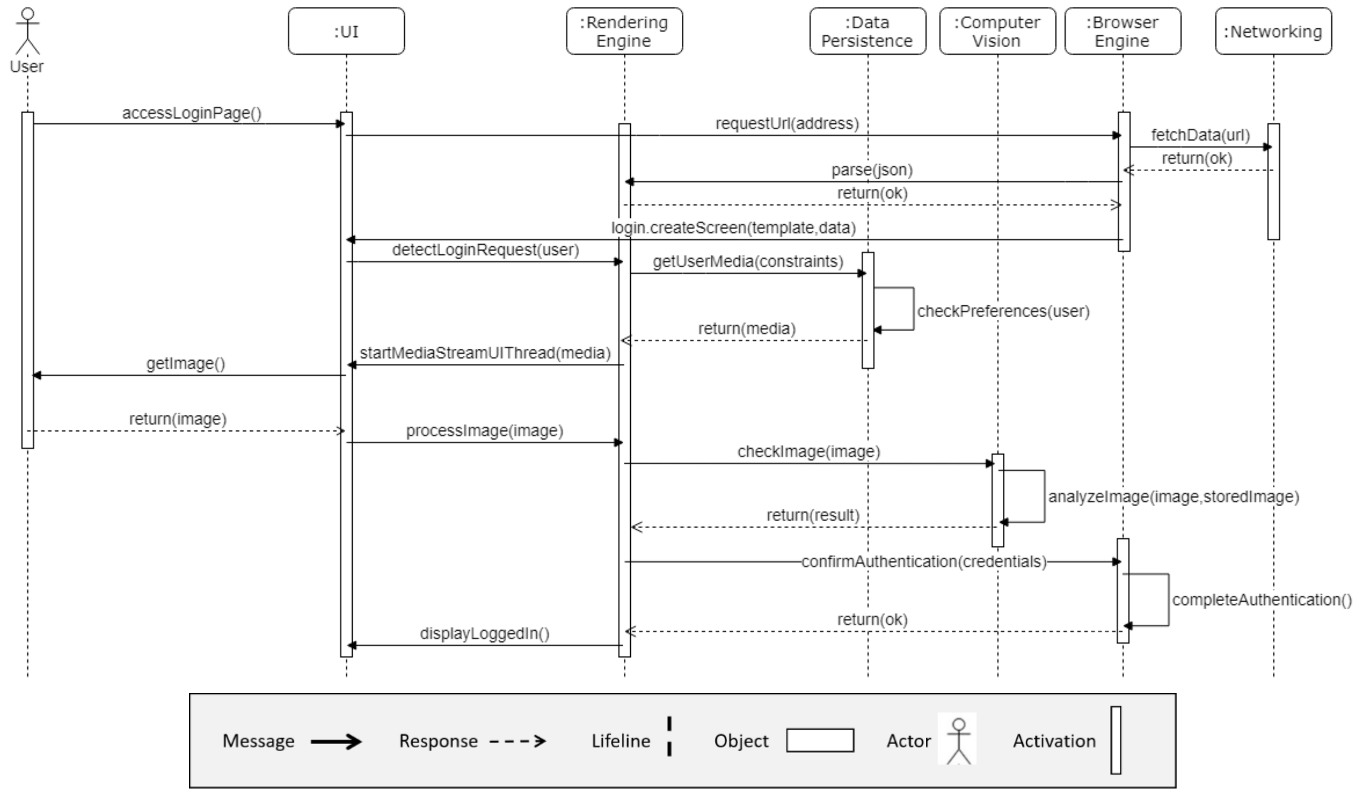

Figure 7: Chrome Login Sequence Diagram

Data Dictionary

Blink: WebKit-based rendering engine
Browser: represents the browser window, contains multiple instances of WebContents
Cache: space in a computer's hard drive and in RAM memory where the browser saves copies of previously visited web pages. The browser uses the cache like a short-term memory
I/O Thread: a thread in the browser process that handles IPC’s and network requests, and in the renderer process that just handle IPC’s

IPC Channel Proxy: Within the browser, communication with the renderers is done in a separate I/O thread. Messages to and from the views need to be proxied over to the main thread using a Channel Proxy
Main/UI Thread: a thread in the browser process that updates the UI and renderer process that runs most of Blink; responsible for rendering web pages on screen
Parser: compiler or interpreter component that breaks data into smaller elements for easy translation into another language
Renderer/Render host: multi-process embedding layer
RenderProcessHost: initialized on the main thread that creates a new render process for each website
RenderView: responsible for navigational commands, receiving input events and painting web pages
RenderViewHost: receives view-specific messages from RenderProcessHost
RenderWidget: input event handling and painting
RenderWidgetHost: handles the input and painting for RenderWidget
ResourceDispatcherHost: responsible for sending network requests to the internet
Tab Helpers: separate objects that attach to WebContents
WebContents: embedded to allow multi-process rendering of HTML
WebKit: rendering engine
WebKit Glue: converts WebKit types to Chromium types.
WebKit Port: integrates with platform dependent system services such as resource loading and graphics

Naming Conventions

API: Application Program Interface
CSS: Cascading Style Sheets
DOM: Document Object Model
I/O: Input/Output
IPC: Inter-Process Communication
NFR: Non-Functional Requirements
OO: Object Oriented
OS: Operating System
UI: User Interface

Conclusions

It was difficult to piece together the existing autofill and webcam functionality without spending weeks investigating thousands of individual code files. The existing documentation did not provide a “bigger picture” explanation, so many assumptions had to be made about the connections between systems. It is likely that multiple of the assumptions made by the team resulted in inaccuracies in the feature design.

Regardless of the difficulties faced during the design stage, the team decided to implement the design option that built off the preexisting functionality in Chrome in order to ease the development process for the engineers and take advantage of its embedded security benefits. Although the feature’s implementation could cause risks to performance and security if designed poorly, it was determined that the efficiency and security benefits provided by a successful system outweighed the risks.

Lessons Learned

The team realized that even seemingly small changes to a subcomponent can have a large impact on the entire system in unexpected ways. This really reinforced the importance of low coupling and high cohesion within software architectures. It is important for developers to understand how their actions can affect other parts of a system. Regression testing needs to be completed for full systems to confirm that any architectural changes do not cause negative effects in unforeseen areas. The team also learned the difficulty of selecting one design option, as there are always many advantages and disadvantages to each approach. It is important to be able to determine which non-functional requirements are the most important to stakeholders, and how each design alternative meets the criteria.

References

[1] E. Bidelman, "Capturing Audio and Video in HTML5," 2016. [Online]. Available: https://www.html5rocks.com/en/tutorials/getusermedia/intro/#toc-gettingstarted. [Accessed 2018].

[2] Chromium Project, "Form Autofill," 2014. [Online]. Available: https://www.chromium.org/developers/design-documents/form-autofill. [Accessed 2018].

[3] Chromium Project, "How Chromium Displays Web Pages," 2012. [Online]. Available: https://www.chromium.org/developers/design-documents/displaying-a-web-page-in-chrome. [Accessed 2018].

[4] The Chromium Project, "Multi-process Architecture," 2008. [Online]. Available: https://www.chromium.org/developers/design-documents/multi-process-architecture. [Accessed 2018].

[5] Chromium Blog, "Multi-process Architecture," 2008. [Online]. Available: https://blog.chromium.org/2008/09/multi-process-architecture.html. [Accessed 2018].

[6] M. Mahemoff, "Extensions and Apps in the Chrome Web Store," Google, 2010. [Online]. Available: https://developer.chrome.com/webstore/apps_vs_extensions. [Accessed 2018].