UNIVERSITY OF MACEDONIA
POSTGRADUATE STUDIES PROGRAMME
DEPARTMENT OF APPLIED INFORMATICS

Experimental and Emerging Web APIs

Diploma Thesis

by

Kavvadias Spyridon

Thessaloniki, June 2023

EXPERIMENTAL AND EMERGING WEB APIS

Kavvadias Spyridon

Bachelor in Agricultural Technology, University of the Peloponnese, 2002

Diploma Thesis

submitted for the partial fulfillment for the requirements of

MSc in Applied Informatics

Supervising Professor
Kaskalis T.H.

Approved by the three member committee on DD/MM/YYYY

Kaskalis T.H	Georgiadis Christos	Xinogalos Stylianos
....	....	....

Kavvadias Spyridon

....

Περίληψη

Τα πειραματικά application programming interfaces του διαδικτύου (web APIs) είναι αναδυόμενες τεχνολογίες οι οποίες παρέχουν πρόσβαση σε λειτουργικότητα συσκευών και λογισμικού μέσα από διαδικτυακές εφαρμογές. Τα web APIs δίνουν τη δυνατότητα στους προγραμματιστές να δημιουργούν δυναμικές διαδικτυακές εφαρμογές οι οποίες έχουν πρόσβαση σε χαρακτηριστικά, παραδοσιακά διαθέσιμα μόνο σε τοπικές εφαρμογές. Ωστόσο, καθώς αυτά τα APIs βρίσκονται ακόμα σε φάση ανάπτυξης υπόκεινται σε περιορισμούς και τροποποιήσεις, επηρεάζοντας την ανάπτυξη των εφαρμογών. Στο πλαίσιο της παρούσας διπλωματικής εργασίας ερευνώνται δώδεκα πειραματικά web APIs, άμεσα σχετιζόμενα με το hardware. Αυτά τα APIs μπορούν να συμβάλουν στην ενίσχυση του browser με νέα χαρακτηριστικά και δυνατότητες. Ο browser θα αποτελέσει δυνητικά ένα αυτόνομο λειτουργικό σύστημα που θα παρέχει στις διαδικτυακές εφαρμογές τον ίδιο βαθμό πρόσβασης σε χαρακτηριστικά του hardware και του software όπως οι τοπικές εφαρμογές. Μέσα από τη συγκέντρωση και ανάλυση των πειραματικών web APIs αξιολογούνται όσα από αυτά είναι επιτυχημένα, όσα χρήζουν βελτίωσης και όσα πρέπει να αντικατασταθούν. Παράλληλα, παρουσιάζεται η καταγραφή του ρυθμού υιοθέτησης των web APIs από τους προγραμματιστές καθώς και η πραγματική τους χρήση, με σκοπό την ταυτοποίηση εκείνων των APIs που κερδίζουν ολοένα και περισσότερο έδαφος αλλά και όσων απαιτούν περαιτέρω ανάπτυξη ή αντικατάσταση. Επιπλέον, η παρούσα μελέτη παρουσιάζει δεδομένα για τον ρυθμό υιοθέτησης των πειραματικών web APIs που σχετίζονται άμεσα με το hardware και μπορούν να συμβάλουν στην ιεράρχηση των πόρων για τη βελτίωση και ανάπτυξη του οικοσυστήματος του διαδικτύου. Επιπρόσθετα, η έρευνα και ανάλυση των πειραματικών και αναδυόμενων web APIs παρέχει τη δυνατότητα να προβλεφθούν πιθανοί κίνδυνοι για την ασφάλεια και την ιδιωτικότητα, ζητήματα που πρέπει να αντιμετωπιστούν πριν ακόμα τα συγκεκριμένα APIs ενσωματωθούν πλήρως στους browsers. Απώτερος στόχος της παρούσας έρευνας είναι να παράγει δεδομένα που συντείνουν στη μελλοντική ανάπτυξη του διαδικτύου και στην ενδυνάμωση του web browser ως αυτόνομου λειτουργικού συστήματος.

Λέξεις Κλειδιά: Πειραματικά Web APIs, Web browser, Λειτουργικό Σύστημα, Web development, Αναδυόμενες τεχνολογίες, Standardization, Adoption, Hardware interfaces, Javascript, Web technologies, Web applications.

Abstract

Experimental web application programming interfaces (web APIs) are emerging technologies that provide access to device hardware and software functionality through web applications. These APIs allow web developers to create more sophisticated and powerful web applications that can access features traditionally only available to native applications. However, as these APIs are still in development, they are prone to limitations and changes that could potentially affect web application development. For the purposes of this master's thesis, twelve experimental and emerging web APIs have been identified, collected and analyzed with a specific focus on hardware APIs. These APIs are essential in demonstrating how web browsers can be enhanced with new features and capabilities. The ultimate goal of using the browser as an operating system is to provide web applications with the same level of access to hardware and software features as native applications, thus using the browser as an operating system. Through collecting and analyzing experimental web APIs, we can help achieve this goal by identifying which APIs are successful, which ones need improvement, and which ones need to be replaced. Furthermore, this analysis aims to demonstrate the adoption rate of experimental, hardware-related web APIs, which can help prioritize resources and development efforts for improving the web ecosystem. By measuring their adoption rate by developers, as well as actual usage, we can determine which APIs are currently gaining popularity and which ones may need further development. Additionally, the analysis of experimental web APIs can reveal potential security and privacy concerns that need to be addressed before APIs are fully integrated into web browsers. Overall, this thesis aims to provide valuable insights into the future of web application development and the potential of the web browser as a full-fledged operating system.

Keywords: Experimental Web APIs, Web browser, Operating System, Web development, Emerging technologies, Standardization, Adoption, Hardware interfaces, Javascript, Web technologies, Web applications.

Prologue - Thanks

I would like to thank my supervisor, Professor Theodoros Kaskalis for his guidance and valuable feedback in this diploma thesis as well as his inspiring class in the MSc programme. I would also like to thank my fellow student in the University of Macedonia, Angelos Martidis, for his help during research and study. Finally, I owe gratitude to my life partner, Dora, for her support and love in everything I try to accomplish.

Contents

Index of Figures 7
Index of Tables 8
1. Introduction 10
1.1 Description 10
1.2 Purpose - Goals 11
1.3 Research Questions 12
1.4 Contribution 13
1.5 Terminology 13
1.6 Description of Chapters 14
2. Overview 15
2.1 Introduction 15
2.2 Experimental Web APIs 17
2.2.1 Definition and Classification 17
2.2.2 Importance of the Research 19
2.2.3 Challenges 22
2.2.4 Use cases and examples 23
2.2.5 The Future 24
2.3 The Web Browser 25
2.3.1 Evolution and Use Cases 25
2.3.2 Categorization 27
2.3.1 Rendering Engines 28
2.3.4 Market Share 29
2.4 The Web Browser as an Operating System 30
2.4.1 Advantages 31
2.4.2 Challenges 32
2.4.3 Essential Features 33
2.4.4 Where we are now 34
2.5 Experimental Web API Implementation 35
2.5.1 Implementation in Chromium 35
2.5.2 Implementation in Chrome 36
2.5.3 Implementation in Firefox 37
2.5.4 Implementation in Safari 38
3. Methodology 40
3.1. Experimental and Emerging Web APIs 43
3.1.1 Audio Output Devices API 43
3.1.2 Background Synchronization API 51
3.1.3 Barcode Detection API 58
3.1.4 Compute Pressure API 64
3.1.5 Web Bluetooth API 71

3.1.6 Keyboard API 79
3.1.7 Presentation API 86
3.1.8 Web NFC API 92
3.1.9 WebGPU API 97
3.1.10 WebHID API 106
3.1.11 WebUSB API 112
3.1.12 WebXR Device API 118
4. Epilogue **128**
4.1 Conclusions 128
4.2 Limits of the research 129
4.3 Future Expansions 129
5. Bibliography **137**

Index of Figures

Figure 2-1: Web browser market share 2009-2023 - Statcounter, p.30
Figure 3-1: HTMLMediaElementSetSinkId percentage of page loads over time in Chrome, p.50
figure 3-2: BackgroundSync percentage of page loads over time in Chrome, p.56
Figure 3-3: PeriodicBackgroundSync percentage of page loads over time in Chrome, p.57
Figure 3-4: BarcodeDetectorDetect percentage of page loads over time in Chrome, p.62
Figure 3-5: ShapeDetection_BarcodeDetectorConstructor percentage of page loads over time in Chrome, p.63
Figure 3-6: WebBluetoothGetAvailability percentage of page loads over time in Chrome, p.77
Figure 3-7: WebBluetoothGetDevices percentage of page loads over time in Chrome, p.78
Figure 3-8: WebBluetoothRequestDevice percentage of page loads over time in Chrome, p.78
Figure 3-9: KeyboardApiGetLayoutMap percentage of page loads over time in Chrome, p.83
Figure 3-10: KeyboardApiLock percentage of page loads over time in Chrome, p.84
Figure 3-11: KeyboardApiUnlock percentage of page loads over time in Chrome, p.85
Figure 3-12: PresentationDefaultRequest percentage of page loads over time in Chrome, p.90
Figure 3-13: PresentationAvailabilityChangeEventListener percentage of page loads over time in Chrome, p.91
Figure 3-14: WebNfcAPI percentage of page loads over time in Chrome, p.96
Figure 3-15: WebNfcNdefReaderScan percentage of page loads over time in Chrome, p.96
Figure 3-16: Web GPU API in the GPU-Web Application Stack, p.98
Figure 3-17: WebGPU percentage of page loads over time in Chrome, p.105
Figure 3-18: HTMLCanvasElement_WebGPU: Percentage of page loads over time in Chrome, p.105
Figure 3-19: HidGetDevices percentage of page loads over time in Chrome, p.110
Figure 3-20: HidRequestDevice percentage of page loads over time in Chrome, p.111

Figure 3-21: UsbGetDevices percentage of page loads over time in Chrome, p.116
Figure 3-22: UsbRequestDevice percentage of page loads over time in Chrome, p.117
Figure 3-23: NavigatorXR percentage of page loads over time in Chrome, p.123

Index of Tables

Table 3-1: MediaDevices.selectAudioOutput() browser support, p.48-49
Table 3-2: HTMLMediaElement.setSinkId() browser support, p.49
Table 3-3: HTMLMediaElement.sinkId browser support, p.49
Table 3-4: HTMLMediaElementSetSinkId top Five sample URLs, p.50
Table 3-5: SyncManager, getTags, register browser support, p.55
Table 3-6: Background Sync in service workers browser support, p.55
Table 3-7: BackgroundSync top five sample URLs, p.56
Table 3-8: PeriodicBackgroundSync top five sample URLs, p.57
Table 3-9: BarcodeDetector, BarcodeDetector() constructor, detect, getSupportedFormats browser support, p.62
Table 3-10: BarcodeDetectorDetect top five sample URLs, p.63
Table 3-11: Bluetooth browser support, p.76
Table 3-12: availabilitychanged browser support, p.76
Table 3-13: getAvailability browser support, p.76
Table 3-14: getDevices browser support, p.77
Table 3-15: requestDevice browser support, p.77
Table 3-16: WebBluetoothGetAvailability, WebBluetoothGetDevices, WebBluetoothRequestDevice top five sample URLs, p.79
Table 3-17: api.Keyboard browser support, p.82
Table 3-18: api.KeyboardLayoutMap browser support, p.83
Table 3-19: KeyboardApiGetLayoutMap top five sample URLs, p.84
Table 3-20: KeyboardApiLock top five sample URLs, p.84
Table 3-21: KeyboardApiUnlock top five sample URLs, p.85
Table 3-22: Presentation, defaultRequest browser support, p.89
Table 3-23: receiver browser support, p.89
Table 3-24: PresentationDefaultRequest top five sample URLs, p.90
Table 3-25: PresentationAvailabilityChangeEventListener top five sample URLs, p.91
Table 3-26: NDEFReader, NDEFReader(), makeReadOnly, reading, readingerror, scan, write browser support, p.95

Table 3-27: WebNfcAPI, WebNfcNdefReaderScan top five sample URLs, p.96
Table 3-28: GPU, getPreferredCanvasFormat, requestAdapter browser support, p.104
Table 3-29: WebGPU, HTMLCanvasElement_WebGPU top five sample URLs, p.106
Table 3-30: HID, connect, disconnect, getDevices, requestDevice browser support, p.109
Table 3-31: HidGetDevices top five sample URLs, p.110
Table 3-32: HidRequestDevice top five sample URLs, p.111
Table 3-33: getDevices, requestDevice browser support, p.116
Table 3-34: UsbGetDevices top five sample URLs, p.117
Table 3-35: UsbRequestDevice top five sample URLs, p.117
Table 3-36: xr browser support, p.122
Table 3-37: NavigatorXR top five sample URLs, p.123
Table 3-38: Reference support table for experimental and emerging web APIs in major browsers, p.124-125
Table 3-39: Adoption Table of Experimental and Emerging Web APIs, p.126

1. Introduction

1.1 Description

Experimental web browser APIs refer to a set of browser APIs that are currently in the experimental stage and not yet standardized by web standards bodies. They are provided by web browser vendors in order to allow developers to experiment with new features, functionalities, and capabilities of web browsers **[1]**. These APIs are subject to change or removal without notice, and their behavior and performance may not be consistent across different web browsers. As such, they are typically not recommended for production use but rather for testing and development purposes **[1, 2]**. Experimental web browser APIs are designed to help web developers stay ahead of the curve and to provide early feedback on new web technologies to browser vendors.

Experimental web browser APIs play a crucial role in the development of the web by allowing developers to create innovative applications that can make the most out of new technologies and features. These APIs provide a way for developers to access and use new functionality that is not yet standardized or widely supported across browsers. By experimenting with these APIs, developers can explore new ideas and push the boundaries of what is possible on the web.

Moreover, experimental web browser APIs can help drive the standardization process by providing feedback to the standards bodies and browser vendors **[3]**. Developers can provide valuable feedback on the usability, performance, and security of these APIs, which can help refine the API specification and ensure that they meet the needs of the web development community. Experimental web browser APIs are essential for pushing the web forward and enabling developers to create innovative, cutting-edge applications. By providing early access to new functionality and driving the standardization process, these APIs help shape the future of the web.

Experimental web APIs that are hardware-related are very important because they enable web developers to access hardware functionality and features that were previously only available to native applications **[4]**. This allows for the development of more powerful and immersive web-based applications that can take advantage of hardware

capabilities such as sensors, cameras, microphones, and other peripherals **[5]**. With the increasing use of web technologies in different domains, including education, gaming, e-commerce, and healthcare, the ability to access hardware through the web browser opens up new possibilities for web application development. Additionally, as the world becomes more connected through the Internet of Things (IoT), web-based applications that can interact with physical devices become increasingly important **[6, 7]**. Therefore, experimental hardware-related web APIs play a vital role in shaping the future of the web and its ability to support innovative applications that leverage hardware functionality.

1.2 Purpose - Goals

The purpose of this thesis is to provide a point of reference in 2023 in regards to recent progress in web development by identifying, cataloging and analyzing experimental and emerging web APIs in regards to advancing the concept of the web browser as an Operating System (OS) **[8]**. The thesis will focus on those APIs specifically that enable, or are directly connected to hardware, be it device hardware or external. This is an important area of research, because one of the fundamental barriers for the use of the web browser as an Operating System is hardware enablement, control and usage. Collecting and analyzing experimental web APIs is important in the context of using the web browser as an Operating System for several reasons:

1. Enhance functionality: For the web browser to evolve into a full-fledged operating system, it is essential that new APIs should enable it to perform functions that were previously only possible with native desktop applications. This can enhance the functionality of web applications and make them more competitive than their native counterparts.
2. Improve user experience: New APIs should improve the user experience of web applications, making them more responsive, efficient, and user-friendly. This is particularly important for real-time applications like video conferencing and gaming, where even small delays or poor quality can significantly impact the user experience.
3. Drive Standardization: Experimental web APIs should help standardization efforts, where developers from different companies and organizations work to establish consistent, cross-platform ways of implementing features. This can

potentially reduce fragmentation and improve compatibility across different devices and browsers.

4. Protect and enhance security and privacy: New APIs may introduce new security and privacy concerns that need to be addressed. By analyzing experimental web APIs, developers can identify potential vulnerabilities and address them before they become widespread.

Experimental and emerging web APIs are essential for the continued growth and evolution of the world wide web as a platform and the web browser as an operating system.

1.3 Research Questions

The present study aims to examine experimental and emerging Web APIs in relation to the browser and the hardware of a computing system. Emphasis is put specifically on hardware related APIs that in time will materialize the concept of the web browser as an OS. The thesis collects and analyzes data during the Spring of 2023 based on the following research questions:

1. Are web browsers capable of handling resource management for running applications within the browser, and do experimental or emerging APIs exist that aim to improve resource allocation and usage?
2. What new hardware capabilities can experimental APIs enable for web applications, and how can these capabilities be leveraged to create novel and more powerful web applications?
3. What are the ways in which an Experimental Web API can be enabled and used in different browsers and platforms?
4. What methods and features in Javascript can be used to run and test experimental APIs?
5. What is the current level of support provided by browsers in regards to experimental hardware web APIs?
6. What is the estimated current adoption of Experimental Web APIs by developers and what is an anticipated projection for the immediate future?
7. What conclusions can be drawn from the data in regards to the progress of using the browser as an operating system?

1.4 Contribution

This study provides a thorough examination of the most recent experimental and emerging technologies in the web browser ecosystem during the Spring of 2023. The main focus of this project is the identification of experimental web APIs that are directly or closely related to hardware and the subsequent analysis of their capabilities through a well defined structure. The analysis is performed through the perspective of the concept of using the web browser as an operating system. Research and data collection focus on sources of information on the web that identify current usage and adoption for those APIs. These usage and adoption patterns provide valuable insights into the development of the web platform, and can recognize potential trends for the future of web development and the future of web browsers in general.

1.5 Terminology

API: An API stands for Application Programming Interface. In the context of web browsers, it is a set of instructions that allow developers to interact with the browser and access its features and capabilities.

Web API: A Web API is a type of API that specifically interacts with web technologies such as HTML, CSS, and JavaScript. Web APIs provide a standardized way for web applications to interact with a browser's features and capabilities.

Experimental Web APIs: They are features that are not yet standardized or fully implemented in web browsers but are made available for developers to test and provide feedback. They are often released as prototypes or beta versions, and their functionality, behavior, and API design may change before being officially adopted by web standards organizations. Experimental web APIs allow developers to experiment with new capabilities and technologies, provide feedback to browser vendors and standardization bodies, and ultimately help shape the future of the web.

JavaScript: It is a high-level programming language primarily used for creating interactive and dynamic web content. It is a popular scripting language that is executed on the client-side (i.e in a web browser) and on the server-side (i.e using technologies such as Node.js). JavaScript is designed to be lightweight, flexible, and easy to learn,

making it a popular choice for web development. It is used to add interactivity to websites, build web and mobile applications, and create server-side applications. JavaScript code can be embedded directly into HTML pages or stored in separate files and included in HTML documents.

Event: An event is an action or occurrence that happens within a web page or application, such as a user clicking a button or submitting a form. Web APIs allow developers to listen for and respond to events, such as by executing a JavaScript function when a specific event occurs.

Web application (or web app): It is a software application that is accessed through a web browser over a network, such as the internet or an intranet. Web apps are designed to run on different devices with internet connectivity and do not require installation on a specific device or operating system. They can be accessed from any device with a web browser, including desktop computers, laptops, tablets, and smartphones. Web apps can provide a wide range of functionalities, including online shopping, social media, email, and productivity tools, among others. They are built using web technologies such as HTML, CSS, and JavaScript, and can communicate with servers using protocols like HTTP and WebSockets.

Hardware Web APIs: They are a set of interfaces exposed by web browsers that allow web applications to interact with hardware devices connected to the user's device or computer, such as cameras, microphones, sensors, and other input/output devices. These APIs provide a bridge between web applications and hardware devices, enabling web developers to create rich, interactive experiences that can access and utilize hardware resources. Hardware Web APIs are a key component of the Web of Things (WoT) vision, which aims to connect everyday devices to the web, making them accessible and controllable through web applications.

1.6 Description of Chapters

The thesis is structured in the following way:

The first chapter is an introduction to the subject of the thesis which specifies the goals and purpose of the research, presents the research questions and cites useful terms for the subject and the analysis that will follow in the rest of the thesis.

The second chapter is an overview of the theoretical background through available literature on the subject. It introduces key concepts that are essential and useful in the scope of the essay and the research conducted.

In the third chapter, I introduce the research methodology. I provide qualitative and quantitative data that help to create the framework for the results of the study. Experimental and emerging web APIs are analyzed with a consistent format while results are presented for their support and usage.

In the fourth chapter, conclusions derived from the research are presented, followed by the limits encountered during the study for this thesis and finally some suggestions as to future expansions on the subject. I also provide tables with data on all of the APIs that are analyzed in this thesis. These summary tables show browser support as well as adoption ranking. What follows is a brief analysis of the data presented in the summary tables.

In the final chapter, an extended bibliography of Journal articles and web pages is cited for reference.

2. Overview

2.1 Introduction

The study of Web APIs in general and experimental Web APIs in particular can only be examined through the retrospective of the Web and its evolution.

The history of web browsers can be traced back to the early 1990s, when the first web browser, WorldWideWeb, was created by Tim Berners-Lee **[9]**. Over the years, various web browsers have been developed, including Netscape Navigator, Internet Explorer, Firefox, Chrome, Safari, and Opera, among others. Each new browser introduced new features and improvements, leading to increased functionality and ease of use for users **[10]**.

Today, the Web has become an increasingly important part of modern society, with billions of users around the world relying on it for everything from communication and entertainment to education and commerce **[11]**. It has evolved significantly since its

creation in the late 1980s. A brief overview of the evolution of the web since its creation, can be summarized in the following milestones:

- Web 1.0: This era was characterized by static HTML pages and the use of the first web browsers like Netscape Navigator and Internet Explorer. It was primarily a read-only web, where users could only passively consume content in the form of simple pages that included text and static images only.
- Web 2.0: This era was characterized by the emergence of dynamic and interactive web pages that allowed users to interact with the web and contribute content. This era saw the rise of social media platforms, online marketplaces, and the increasing use of multimedia content.
- Mobile Web: As smartphones and mobile devices became more prevalent, the web had to adapt to these smaller screens and new interaction models. Responsive design and mobile-first development became standard practices.
- Web 3.0: Also known as the Semantic Web, this era is characterized by the use of machine-readable data and the increasing use of Artificial Intelligence and Machine Learning. The focus is on creating a more intelligent web that can better understand and serve users needs.

With the increasing importance of the web in people's lives, web browsers have evolved to become more than just tools for accessing websites; web browsers are now powerful platforms that allow developers to create and run applications within the browser itself, without the need for a separate operating system **[12, 13, 14]**. This concept of the web browser as an operating system (OS) has led to the emergence of experimental web browser APIs, which provide developers with access to hardware resources and other functionalities that were previously only available to native applications.

The development of experimental web browser APIs has been driven by the need for more powerful and versatile web applications that satisfy user's needs, as well as advancements in technology. These APIs provide developers with new ways to interact with hardware devices and other system-level features, such as the file system and network interfaces, within the browser environment. They have also enabled the development of new types of web applications, such as games and multimedia-rich applications, that were previously only possible using native applications **[15]**.

Despite the benefits of experimental web browser APIs, their development and implementation poses challenges, including lack of standardization, incomplete or unstable implementations, and potential security risks **[16]**. Nevertheless, the ongoing evolution of web browsers and the continued development of experimental web browser APIs are likely to drive further innovation in web application development and usage.

2.2 Experimental Web APIs

2.2.1 Definition and Classification

Experimental web browser APIs are web APIs that are not yet fully standardized and are subject to change or removal **[1, 2]**. They are typically made available for developers to test and experiment with in order to provide feedback and influence their development. The testing nature of these APIs means that they are not yet implemented in the official releases of web browsers. They are experimental in the sense that they are not yet considered stable or reliable for use in production environments **[1]**. Experimental web browser APIs are often created to address specific use cases or to enable new functionality on the web, and are typically released in a beta or developer preview version of a web browser.

Experimental web browser APIs are typically released as part of a browser's developer tools, and are often accompanied by documentation and examples **[5, 17]**. As developers experiment with these APIs and provide feedback, they may be refined and eventually adopted as standard APIs. This emergence of experimental web browser APIs has played an important role in the development of the web, providing developers with new tools and capabilities to create more sophisticated and interactive applications. While the Web continues to evolve, it is likely that new experimental web browser APIs will continue to emerge, providing developers with even more powerful tools for web development. The main characteristics of experimental web browser APIs are:

1. Limited implementation: Experimental web browser APIs are often only partially implemented in web browsers and may not be available on all platforms or operating systems.
2. Limited documentation: Since these APIs are still in the testing phase, the documentation may be incomplete or not yet available.

3. May change or be removed: Experimental web browser APIs are subject to change or removal as they are refined or deprecated in favor of more stable APIs.

Experimental web browser APIs differ from standard web browser APIs in that they are not yet fully supported or standardized, and may not be compatible with all web browsers or operating systems **[18]**. Standard web browser APIs, on the other hand, are well-established, documented, and stable, and are considered safe to use in production environments. Experimental web browser APIs can be classified into several categories based on their functionality or purpose. These categories may include:

1. Multimedia APIs - APIs that allow access to multimedia resources such as audio, video, and images.
2. Geolocation APIs - APIs that allow access to location data of the device or user.
3. Input/output APIs - APIs that allow access to user input devices and output devices such as printers.
4. Web storage APIs - APIs that allow access to client-side storage solutions like Local Storage and IndexedDB.
5. Web worker APIs - APIs that allow the creation of background threads to execute JavaScript code outside of the main execution thread.
6. Real-time communication APIs - APIs that allow real-time communication between clients, such as WebRTC.
7. Hardware access APIs - APIs that allow access to hardware devices such as cameras, microphones, and sensors.
8. Security APIs - APIs that allow access to security features such as encryption and authentication.

These categories are not exhaustive and new categories can emerge as new experimental web browser APIs are developed. From the above list, this thesis is going to focus on Hardware access APIs because of their importance in the role of the web browser as an operating system concept. Experimental web browser APIs can also be classified based on their functionality and potential impact on the web as follows:

1. Accessibility APIs: These APIs allow developers to create web applications that are accessible to users with disabilities. They provide access to assistive technologies such as screen readers, magnifiers, and braille displays.

2. Device APIs: These APIs allow web applications to access and use device hardware features such as cameras, microphones, accelerometers, and geolocation sensors.
3. Media APIs: These APIs provide access to media streaming and playback capabilities, including audio and video capture and playback.
4. Network APIs: These APIs allow web applications to communicate with servers and other devices on the internet, including HTTP and WebSocket protocols.
5. Security APIs: These APIs provide security features such as secure storage, authentication, and encryption.
6. User interface APIs: These APIs provide developers with tools for creating custom user interfaces, including 2D and 3D graphics, animations, and layout engines.

The potential impact of experimental web browser APIs on the web can be significant, as they can introduce new features and capabilities that were previously unavailable. However, their impact can also be limited by their incomplete or unstable implementation, lack of standardization, and potential security risks. It is important for developers to carefully evaluate the risks and benefits of using experimental APIs in their applications **[19]**.

2.2.2 Importance of the Research

Experimental web browser APIs are important for several reasons in the development of the web. First, they provide developers with early access to new features and technologies, which allows them to create innovative applications and improve user experience. This is particularly important in a fast-changing technology landscape, where the ability to stay ahead of the curve can give developers a significant competitive advantage.

Experimental web browser APIs can also help improve resource management within the browser **[20]**. This is important because web applications are becoming more complex and resource-intensive, and without proper management, they can slow down or crash the browser. By providing developers with tools and APIs for better managing resources, experimental web browser APIs can help ensure that web applications run smoothly and efficiently.

In addition, experimental web browser APIs can advance the capabilities of web applications **[21]**. For example, new APIs for access to hardware resources like cameras and microphones can enable the creation of new types of applications, such as video chat or augmented reality experiences. Other experimental APIs, such as those for Machine Learning and Artificial Intelligence, can enable developers to create smarter and more personalized applications.

Overall, experimental web browser APIs play a crucial role in driving innovation and progress in the web development community **[22]**. By providing developers with access to new technologies and capabilities, they enable the creation of more powerful and engaging web applications that can improve the user experience and drive business success.

Web browser APIs in general provide developers with access to a wide range of powerful capabilities and functionalities that are not available through standard web development technologies like HTML, CSS, and JavaScript. APIs enable developers to build more sophisticated and feature-rich web applications that can offer better user experiences, increased security, and improved performance. In this context, experimental web browser APIs are essential to the web ecosystem for several reasons:

- They allow developers to create more powerful and feature-rich applications. Web Browser APIs provide access to a wide range of functionalities, such as multimedia, geolocation, and device sensors, which can be used to enhance the user experience and create more interactive applications.
- They improve cross-browser compatibility. Web Browser APIs are standardized, meaning that they are implemented in the same way across different browsers. This makes it easier for developers to create applications that work consistently across multiple browsers and platforms.
- They enable integration with third-party services. Web Browser APIs allow web applications to interact with external services and APIs, such as Social Media platforms and payment gateways, making it easier to incorporate external data and functionality into the application.
- They promote innovation and experimentation. Web Browser APIs are constantly evolving, with new features and functionalities that are constantly being added.

This way developers are provided with opportunities to experiment with new technologies and create innovative applications.

- They enable the development of web-based operating systems. Web Browser APIs provide the infrastructure for running applications within the web browser, enabling the development of web-based operating systems and application platforms that can function similarly to traditional desktop applications.

Early examples of experimental web browser APIs and their impact on the web:

One of the earliest examples of experimental web browser APIs was the XMLHttpRequest API, which was first introduced in Internet Explorer 5 in 1999 **[23, 24]**. This API allowed web developers to create dynamic, interactive web applications that could make requests to a web server and update the page without requiring a full page reload. This technology paved the way for the development of modern single-page applications and has had a significant impact on the web.

Another example of an experimental web browser API that had a significant impact on the web is the WebGL API, which was first introduced in 2009 **[25]**. This API allows web developers to create 3D graphics and animations directly in the web browser without requiring any plugins or additional software. Thus a new realm of possibilities for web-based games and visualizations has opened up.

The Web Audio API is another example of an experimental web browser API that has had a significant impact on the web **[26, 27]**. This API allows web developers to create sophisticated audio processing and synthesis applications directly in the web browser, without requiring any additional software or plugins. This has enabled a new generation of web-based music and audio applications.

Overall, these early experimental web browser APIs helped to push the boundaries of what was possible on the web and paved the way for the development of modern web technologies.

2.2.3 Challenges

Experimental web browser APIs present a number of challenges that must be addressed in order to fully realize their potential in the development of the web. The most fundamental challenges could be summarized as follows:

Volatility and unpredictability:

Experimental web APIs are, by definition, unstable and unpredictable as they are still in development and subject to change **[1, 2]**. As they are not standardized and may not be supported by all browsers, they can be volatile and may break or stop working as the technology evolves. This can create challenges for developers who are using experimental web APIs in their projects, as they may need to constantly update their code to adapt to changes in the API. However, experimental web APIs are still valuable for developers because they provide an opportunity to experiment with new technologies and features before they become mainstream, allowing developers to stay at the forefront of web development. Additionally, by providing feedback and reporting issues with experimental web APIs, developers can help improve the stability and predictability of the APIs over time.

Lack of standardization:

Experimental web APIs are often incomplete or have unstable implementations due to their nature of being in development and subject to change **[1]**. These APIs are not fully tested and may not be fully supported by all browsers or platforms, which can lead to inconsistencies in performance and behavior.

Incomplete or unstable implementations:

Experimental web APIs are often incomplete or unstable in their implementation because they are still in the development phase and subject to change. These APIs are released to developers and the public as a way to gather feedback and refine the API before it is officially standardized and integrated into web browsers. The incomplete or unstable implementation of experimental web APIs can cause issues for developers who rely on these APIs for their projects. Changes to the API may require developers to rewrite their code, which can be time-consuming and frustrating. In some cases, the API may be deprecated entirely, leaving developers with unsupported code.

Potential security risks:

Experimental web APIs can pose potential security risks since they may not have undergone extensive testing or have proper security measures in place **[16, 19]**. These APIs are often not yet standardized, and may not have a clear set of security guidelines or best practices. Developers may not have fully understood the security implications of these APIs and may inadvertently introduce vulnerabilities into their applications. Furthermore, experimental web APIs may not have the same level of scrutiny and review as stable, widely-used APIs. This means that vulnerabilities or exploits may go unnoticed for longer periods of time, potentially allowing attackers to take advantage of them. As such, it is important for developers to carefully evaluate the potential security risks associated with using experimental web APIs and to take appropriate steps to mitigate those risks, such as implementing additional security measures or waiting for the API to become more stable and widely adopted before incorporating it into their applications.

Developers may also face challenges in using these APIs due to the lack of documentation and examples, making it harder to understand their functionality and how to integrate them into their applications. This can lead to frustration and delays in development. To address these challenges, organizations such as the W3C (World Wide Web Consortium) work to develop and promote standards for web APIs **[28]**. Standardization efforts can help ensure that web APIs are implemented consistently across different browsers and provide a stable foundation for web application development.

However, despite the challenges, working with experimental web APIs can also be rewarding. Developers who are early adopters of these APIs can gain a competitive advantage by creating innovative web applications and services that take advantage of new functionality not available to others. As the APIs become more stable and are eventually standardized, early adopters can also benefit from having a head start on the competition.

2.2.4 Use cases and examples

There are several experimental and emerging browser APIs that are being developed and tested by browser vendors. These APIs are intended to provide developers with new and innovative ways to build web applications and enhance the functionality of existing ones.

This section provides examples of experimental web browser APIs that have been developed and their potential use cases, including hardware-related APIs and APIs that improve resource management **[29]**.

Some of these APIs are the following:

1. Web Bluetooth API: This API allows web applications to communicate with Bluetooth Low Energy devices.
2. Web USB API: This API allows web applications to communicate with USB devices.
3. Web MIDI API: This API allows web applications to communicate with MIDI devices.
4. WebVR API: This API provides access to virtual reality devices and allows web applications to create and display virtual reality content.
5. WebXR API: This API provides access to both augmented reality and virtual reality devices and allows web applications to create and display both types of content.
6. Web Speech API: This API allows web applications to use speech recognition and synthesis.
7. Web Authentication API: This API provides a standardized way for web applications to authenticate users using external authentication hardware.
8. Web Payments API: This API allows web applications to process payments.
9. Web NFC API: This API allows web applications to communicate with NFC (Near Field Communication) devices.

2.2.5 The Future

Experimental web browser APIs have the potential to play a significant role in shaping the future of the web **[30]**. As new technologies and hardware emerge, web developers will need access to new APIs in order to create innovative and feature-rich web applications. Experimental web browser APIs can provide developers with access to new functionality before it is standardized, allowing them to experiment with new features and provide feedback to standardization bodies.

In addition, experimental web browser APIs can also drive innovation and competition among web browser vendors. As new APIs are developed, vendors can

compete to provide the most performant and feature-rich implementation, leading to a better overall user experience for web users.

Looking ahead, it is likely that experimental web browser APIs will continue to hold an important role in the development of the web. As new hardware and technologies emerge, web developers will need access to new APIs in order to take advantage of these advancements. Additionally, the continued development of experimental APIs can help to push the standardization process forward, ensuring that the web remains a flexible and innovative platform for the delivery of content and applications.

The potential of experimental web browser APIs is vast, and their impact on the evolution of the web cannot be understated. As such, it is important for web developers, browser vendors, and standardization bodies to continue to invest in the development and implementation of experimental APIs in order to ensure the continued growth and success of the web.

2.3 The Web Browser

2.3.1 Evolution and Use Cases

Web browsers as tools for interacting with the web have undergone significant transformations alongside the evolution of the Web **[10]**. Browsers have evolved significantly since the early days. Some key milestones in their evolution are the following **[9, 10]**:

1. Early web browsers: The first web browser was created in 1990 by Sir Tim Berners-Lee, and was called WorldWideWeb. It was later renamed Nexus. Other early browsers included ViolaWWW, Erwise, and Mosaic.
2. The browser wars: In the mid-1990s, the browser wars began between Microsoft's Internet Explorer and Netscape Navigator. This period saw significant innovation in browsers, as each company tried to outdo the other with new features and capabilities.
3. Standards-based web development: The late 1990s and early 2000s saw the emergence of web standards, which helped to make web development more consistent across browsers. This allowed developers to create websites that

worked well on multiple browsers, rather than having to create separate versions for each browser.

4. Mobile browsing: The advent of smartphones in the late 2000s brought about a new era of browsing, with mobile-optimized websites and mobile-specific browsers. This led to a shift in how websites were designed and developed, with a focus on responsive design and mobile-first development **[31]**.
5. Web apps: In recent years, browsers have become more powerful and capable, allowing for the development of complex web applications that can rival traditional desktop applications in functionality. This has led to the concept of the browser as an application platform, with web apps running entirely within the browser environment **[32, 33]**.

The evolution of web browsers has been closely tied to the emergence of new technologies and standards for web development. In the early days of the web, browsers were simple tools for rendering HTML and displaying images. Over time, browsers have become increasingly sophisticated, adding support for new technologies like JavaScript, CSS, and multimedia capabilities. As web development has become more complex and diverse, the need for standardization and interoperability has become more important. Web APIs have emerged as a way to address this need, providing developers with standardized ways to access hardware and software features of the browser.

Web browsers are used today for a wide variety of purposes. Some of the most common use cases include:

- Browsing the web: The primary use case for web browsers is browsing the web, allowing users to access and view websites and web content.
- Communication: Web browsers can be used for communication, including email, instant messaging, and video conferencing.
- Social Media: Web browsers are commonly used to access social media platforms, allowing users to connect with others and share information.
- Entertainment: Web browsers can be used for entertainment purposes, such as streaming video and music, playing games, and accessing online media.
- Online shopping: Web browsers are commonly used for online shopping, allowing users to browse and purchase products from online retailers.

- Productivity: Web browsers can be used for productivity purposes, such as accessing online tools and software, managing email and calendars, and collaborating on documents.
- Education: Web browsers can be used for educational purposes, such as accessing online courses and educational resources.
- Development: Web browsers can be used by developers for testing and debugging web applications, as well as for accessing development tools and resources.
- Research: Web browsers can be used for research purposes, allowing users to access and search online databases, journals, and other resources.

2.3.2 Categorization

Web browsers can be categorized in several different ways, based on:

- Popularity: This is probably the most common way to categorize web browsers. Popular browsers include Google Chrome, Mozilla Firefox, Apple Safari, Microsoft Edge, and Opera.
- Platform: Browsers can also be categorized based on the platform they run on. For example, there are desktop browsers (such as Chrome, Firefox, and Edge), mobile browsers (such as Safari and Chrome for iOS and Android), and even console browsers (such as the web browser on the Nintendo Switch).
- Features: Browsers can also be categorized based on the features they offer. Some browsers prioritize privacy and security, while others offer advanced developer tools or unique user interfaces.
- Intended Use: Browsers can also be categorized based on their intended use. For example, there are browsers designed specifically for gaming (such as Opera GX), browsers designed for low-end devices (such as UC Browser), and browsers designed for use in education (such as the KidZui browser).
- Ownership: Browsers can also be categorized based on their ownership. Some browsers are developed and maintained by large tech companies, while others are developed by smaller independent teams or open-source communities.
- Rendering Engine: Categorizing web browsers according to their rendering engine can be important for developers and users as it can affect the compatibility of websites and web applications. Different rendering engines may have different

levels of support for experimental web APIs, so it can be helpful to know which browsers are using which engine when considering the use of certain APIs.

In the following section I am offering an analysis of the categorization of web browsers according to the engine they are built upon, because it is important in the context of experimental and emerging APIs and the scope of this essay.

2.3.1 Rendering Engines

Categorizing web browsers based on the rendering engine they use is important in relation to experimental web APIs because different rendering engines have different levels of support for experimental features. Some experimental features may only be available on certain rendering engines or may have different levels of support across different rendering engines. By categorizing web browsers based on their rendering engine, developers can better understand which experimental web APIs are available to them and which may require additional testing or workarounds to implement. This can help ensure that web applications are optimized for different browsers and devices and that the user experience is consistent across different platforms.

There are several web browser engines or web rendering engines used today; some of the most popular ones are **[34]**:

1. Blink: Developed by Google, used by Google Chrome and other Chromium-based browsers.
2. WebKit: Developed by Apple, used by Safari and some other browsers.
3. Gecko: Developed by Mozilla, used by Firefox and other Mozilla-based browsers.
4. Trident: Developed by Microsoft, used by older versions of Internet Explorer.
5. EdgeHTML: Developed by Microsoft, used by Microsoft Edge before switching to Blink.
6. KHTML: An open-source engine used by the Konqueror browser, and its descendant engine, WebKit.
7. Servo: A new, experimental engine developed by Mozilla, written in Rust.

Each web engine has its own strengths and weaknesses, and they may differ in terms of speed, efficiency, stability, and support for web standards and experimental

features. Some web browsers may use a combination of different engines, or may switch between engines depending on the type of content being rendered. According to the above, web browsers can be also categorized according to the web rendering engine or browser engine that they use. This categorization is the following:

- Chromium-based browsers: These are web browsers that use the Blink engine, which was originally developed by Google for the Chrome browser. Examples include Google Chrome, Microsoft Edge, Opera, Vivaldi, Brave, and many others.
- Gecko-based browsers: These are web browsers that use the Gecko engine, which was developed by Mozilla for the Firefox browser. Examples include Firefox, SeaMonkey, and Pale Moon.
- WebKit-based browsers: These are web browsers that use the WebKit engine, which was originally developed by Apple for the Safari browser. Examples include Safari, Brave (on iOS), and others.
- Trident-based browsers: These are web browsers that use the Trident engine, which was developed by Microsoft for Internet Explorer. Examples include older versions of Internet Explorer, such as IE11 and earlier.
- Text-based browsers: These are web browsers that display only text and do not support images or other media. Examples include Lynx, w3m, and Links.

There are also hybrid browsers that use a combination of different engines or switch between engines depending on the type of content being rendered.

2.3.4 Market Share

As can be seen in the graph below **[35]**, Google's Chrome web browser has an overwhelming majority of users, similar to what Microsoft's Internet explorer used to have before its demise. Therefore, it is safe to assume that the implementation and usage of experimental web APIs in Chrome plays an essential role in the adoption of said APIs, and fundamentally alters the landscape in regards to using the web browser as an operating system or an application platform. Every browser has a slightly different method for implementing and using experimental APIs in their experimental branches, however most major browser vendors are members in the W3C (World Wide Web Consortium) which sets the tone and policy for new experimental APIs and features that

will soon make their way into official production versions of the web browsers in the market.

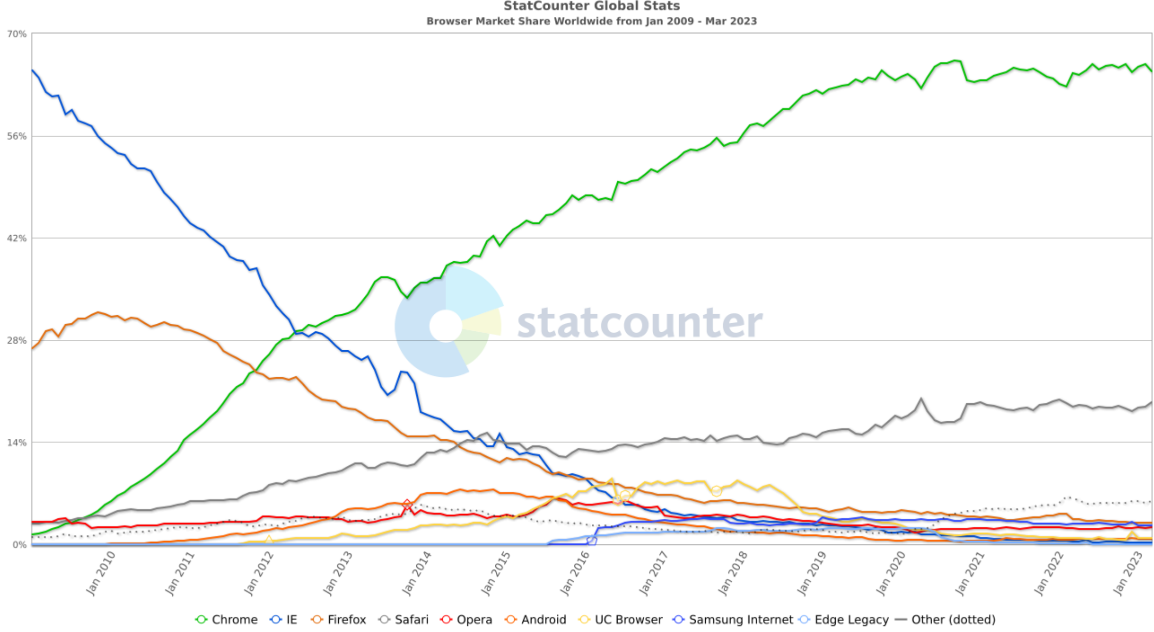

Figure 2-1: Web browser market share 2009-2023 - Statcounter

2.4 The Web Browser as an Operating System

The concept of the web browser as an operating system (OS) refers to the idea that the web browser can be used as a platform for running applications that function similarly to traditional desktop applications, without the need for a separate operating system **[8]**. The concept is that the web browser can provide the underlying infrastructure for running applications, while the applications themselves are built using web technologies such as HTML, CSS, and JavaScript. One of the first people to express this idea was Marc Andreessen, the Netscape co-founder, who stated in 1995: "The Browser Will Be the Operating System" **[36]**.

The web browser has evolved from being a simple tool for browsing static web pages to a powerful application platform capable of running complex web applications. With the emergence of HTML5 and modern web standards, web browsers have become the preferred platform for many developers to create and deploy applications. In fact, some experts have even referred to the web browser as the "operating system of the future" thanks to its ability to run applications and manage resources similar to traditional

operating systems **[37]**. The rise of web-based technologies such as JavaScript and web APIs has rendered it possible to develop highly interactive and sophisticated applications that can run entirely within the web browser.

However, this shift towards web-based applications also raises concerns about security, as the browser is often the primary target for cyber attacks **[16, 19]**. It is important for developers to be aware of potential security risks and to take steps to mitigate these risks when building web applications.

While the idea of the web browser as an operating system is not new, the process is still considered to be in its early stages. However, several examples of web-based operating systems and application platforms already exist, such as Google's Chrome OS **[51]**, Mozilla's WebThings **[52]**, and the open-source Webian Shell **[53]**. As web technologies continue to evolve, it is likely that we will see an increasing number of web-based applications and operating systems in the future.

Overall, the web browser as an operating system offers a number of benefits and challenges for developers, and its evolution is likely to continue to shape the future of computing.

2.4.1 Advantages

Using the browser as an operating system can offer significant benefits for both developers and end-users, by providing increased accessibility **[38]**, cross-platform compatibility, and cost-effectiveness, among other things. More specifically the advantages of using the browser as an operating system include the following:

1. Accessibility: Browsers are ubiquitous, and they are available on virtually every device. This means that applications built using browser-based technologies can be accessed from anyone, anywhere, on any device with a browser, without the need for platform-specific development.
2. Easy deployment: Because browser-based applications can be accessed from anywhere, they are easy to deploy. This means that updates and new features can be rolled out quickly and easily, without the need for lengthy installation processes.
3. Cross-platform compatibility: Browsers provide a standardized platform for application development that works across multiple operating systems and

devices. This means that developers can build applications that work seamlessly across multiple platforms, without the need for platform-specific development.

4. Cost-effective: Browser-based applications can be developed using open-source technologies, which can significantly reduce development costs. Additionally, because they are cross-platform compatible, they can reach a larger audience, which can lead to increased revenue and profitability.
5. Resource efficiency: Browsers are designed to be lightweight and efficient, which means that applications built using browser-based technologies can be more resource-efficient than traditional desktop applications **[39]**.
6. Lower system requirements: Because web-based applications are typically less resource-intensive than traditional desktop applications, they can be run on devices with lower system requirements, making them more accessible to a wider range of users **[40]**.
7. Reduced software installation and maintenance: Because web-based applications can be accessed directly through the web browser, there is no need to install or maintain software on individual devices, reducing overheads and simplifying management.
8. Easier updates: Web-based applications can be updated quickly and easily, with no need for users to download and install updates manually.
9. Greater flexibility: Web-based applications can be easily integrated with other web services and APIs, making it easier to incorporate external data and functionality into the application.

2.4.2 Challenges

While using the web browser as an operating system can offer many advantages, it also presents several challenges that must be carefully considered and addressed by developers. Those challenges include the following:

- Limited system access: Web browsers are designed to run within a sandboxed environment, which limits their access to system resources such as the file system, network interfaces, and hardware devices. This can make it difficult to build applications that require low-level system access.

- Performance limitations: Web browsers are not designed to be high-performance environments, and running resource-intensive applications within the browser can lead to performance issues such as slow load times and laggy user interfaces.
- Security risks: Because web browsers are connected to the internet, they are inherently vulnerable to security threats such as malware, viruses, and phishing attacks. As such, developers must take extra care to build secure applications that do not compromise the user's system or data.
- Limited offline capabilities: While web technologies such as service workers and caching can provide some offline capabilities, web applications still require an internet connection to function fully. This can limit the usefulness of web-based applications in environments with limited or unreliable internet connectivity.
- Lack of standardization: Web technologies and APIs are constantly evolving, and there is often a lack of standardization across different browsers and platforms. This can make it difficult to build cross-platform applications that work consistently across different devices and operating systems.

2.4.3 Essential Features

In order to be used as an operating system, a web browser should have the ability to run web applications, including those that require access to hardware resources like cameras and microphones **[41]**.

In addition to the ability to run web applications, a web browser that is used as an operating system should have a strong security architecture that can protect the system and user data from malicious attacks **[42]**. It should also be able to manage system resources effectively, ensuring that web applications do not use too much CPU, memory, or other system resources, which could slow down the overall performance of the system **[41]**.

Another important feature of a web browser operating system is the ability to work offline and sync data with other devices. This is especially important for users who need to access their data even when they are not connected to the internet **[39]**.

Moreover, the browser operating system should have a reliable and user-friendly interface that allows users to easily access and manage their files and applications. This could include features like a taskbar, file explorer, and a control panel.

Finally, a browser operating system should have a strong development community, with a wide range of developers working on various aspects of the system, including security, performance, and compatibility with different hardware configurations. This will ensure that the system remains up-to-date and responsive to user needs. To summarize, a web browser should have the following features in order to be used as an operating system would:

- The ability to run web applications, including those that require access to hardware resources like cameras and microphones.
- A file system API that allows users to create and manage files locally.
- Support for offline access to web applications.
- A Task Manager.
- Support for multiple windows and tabs.
- The ability to manage and configure system settings like display resolution and network connections.
- Security features like sandboxing and data encryption to protect user data and prevent malicious attacks.

2.4.4 Where we are now

The current state of the browser as an operating system (OS) is still evolving. While many applications can now be run entirely within a web browser, there are still limitations and challenges that prevent it from fully replacing traditional operating systems like Windows, MacOS, or Linux.

One major challenge is the inability of web browsers to directly access hardware resources, such as drivers or device-specific APIs. This makes it difficult for browsers to provide the same level of performance and functionality as a native operating system. Additionally, there are still limitations in terms of system-level functionality such as access to the file system or low-level system settings.

However, with the continued development of experimental web APIs and the increasing use of web-based technologies for developing applications, the browser as an operating system is becoming a more feasible option. Some operating systems, such as Chrome OS and Firefox OS, have already been developed with a focus on using the web browser as the primary interface for running applications. Overall, while it is not yet a

complete replacement for traditional operating systems, the browser as an OS has the potential to become an increasingly viable option in the future.

2.5 Experimental Web API Implementation

Web browsers vary in their approach to adopting experimental web APIs. Some browsers are more aggressive in adopting new APIs while they are still in the experimental phase, while others take a more cautious approach.

Browsers that are more aggressive in adopting experimental web APIs may offer them as experimental features or experimental flags, which developers can enable to test out new functionality **[43]**. These browsers may also provide developer tools and documentation to help developers use these experimental APIs.

Other browsers may wait until experimental web APIs are more mature and standardized before adopting them. These browsers may be more focused on providing stable and consistent experiences for their users and may be less willing to risk breaking changes or compatibility issues by adopting experimental APIs too early.

In general, web browsers are more likely to adopt experimental web APIs if they are popular or widely used among developers, or if they provide significant benefits or improvements to web applications. However, it is important to note that even after an experimental API is adopted by a browser, it may still undergo changes or be removed in later versions as it evolves and becomes more standardized.

Developers who want to use experimental web APIs should be aware of the risks involved, including potential compatibility issues and changes to the API in future versions, and should test their applications thoroughly across a range of browsers and devices to ensure that they work as intended.

2.5.1 Implementation in Chromium

Chromium is the open-source project that serves as the foundation for Google Chrome. Implementation of experimental web apis happens in the following stages **[44]**.

First, Chromium maintains a list of experimental features that are available for developers to test out. These features can be enabled using special flags, which can be

accessed via the browser's settings or through command line flags. These flags allow developers to test out new features and functionality before they are officially released, and to provide feedback and bug reports to the Chromium team.

Chromium also maintains a Canary build, which is a bleeding-edge version of the browser that includes the latest experimental features and APIs. This build is updated daily and is intended for developers who want to stay on the cutting edge of web development and test out the latest features as soon as they become available.

In addition to these experimental features, Chromium also includes a number of developer tools that make it easier to test and debug web applications that use experimental APIs. These tools include the Chrome DevTools, which provide a range of debugging and profiling tools for web developers, as well as the Chrome Web Platform Status site, which provides information on the status of various web APIs in different versions of Chromium and other browsers.

Overall, Chromium provides a range of tools and resources to help developers test and experiment with new web APIs, and to provide feedback and bug reports to the Chromium team as these APIs continue to evolve and mature.

2.5.2 Implementation in Chrome

Google Chrome, which is based on the open-source Chromium project, implements experimental web APIs in a similar way to Chromium **[45]**.

Chrome maintains a list of experimental features that are available for developers to test out. These features can be enabled using special flags, which can be accessed via the browser's settings or through command line flags. These flags allow developers to test out new features and functionality before they are officially released, and to provide feedback and bug reports to the Chrome development team.

Chrome also includes a number of developer tools that make it easier to test and debug web applications that use experimental APIs. These tools include the Chrome DevTools, which provide a range of debugging and profiling tools for web developers, as well as the Chrome Web Platform Status site, which provides information on the status of various web APIs in different versions of Chrome and other browsers.

In addition, Chrome offers a number of channels for developers who want to stay on the cutting edge of web development and test out the latest features as soon as they become available. These channels include the stable channel, which includes stable releases of the browser, as well as the beta, dev, and canary channels, which offer progressively more bleeding-edge releases with newer and more experimental features.

Overall, Chrome provides a range of tools and resources to help developers test and experiment with new web APIs, and to provide feedback and bug reports to the Chrome development team as these APIs continue to evolve and mature.

2.5.3 Implementation in Firefox

Firefox is an open-source web browser developed by Mozilla, also implements experimental web APIs in a similar way to Chromium and Chrome **[46]**.

Firefox maintains a list of experimental features that are available for developers to test out. These features can be enabled using special flags, which can be accessed via the browser's about:config settings or through command line flags. These flags allow developers to test out new features and functionality before they are officially released, and to provide feedback and bug reports to the Firefox development team.

Firefox also includes a number of developer tools that make it easier to test and debug web applications that use experimental APIs. These tools include the Firefox Developer Tools, which provide a range of debugging and profiling tools for web developers, as well as the Mozilla Developer Network (MDN), which provides documentation and tutorials on various web APIs and other web development topics.

In addition, Firefox offers a number of channels for developers who want to stay on the cutting edge of web development and test out the latest features as soon as they become available. These channels include the release channel, which includes stable releases of the browser, as well as the beta, nightly, and developer edition channels, which offer progressively more bleeding-edge releases with newer and more experimental features.

Overall, Firefox provides a range of tools and resources to help developers test and experiment with new web APIs, and to provide feedback and bug reports to the Firefox development team as these APIs continue to evolve and mature.

2.5.4 Implementation in Safari

Safari implements experimental web APIs in a similar way to other web browsers. First, the Safari team identifies and evaluates new web APIs proposed by standards organizations such as the W3C and WHATWG. Once an API is deemed useful and feasible, the team begins implementing it in the browser **[47]**.

Safari typically introduces new experimental web APIs in beta versions of the browser, which are released to developers and early adopters for testing and feedback. As the API matures and gains broader support, it may be included in a stable release of Safari and made available to all users.

Safari also provides a way for developers to enable experimental web APIs that are still in development or not yet widely supported. This is done through the "Experimental Features" menu in Safari's Develop menu. Developers can enable these features to test and experiment with new APIs before they are widely adopted.

Overall, Safari, like other web browsers, strives to support a wide range of web APIs to enable developers to create powerful and innovative web applications.

How the rendering engine affects API implementation

Different rendering engines may support different versions of APIs, or may not support certain APIs at all. This means that web developers need to be aware of which rendering engine their target audience is using and make sure that their code is compatible with that engine.

Also, different rendering engines may implement the same API in slightly different ways. This can lead to inconsistencies in how web pages are rendered, and may require web developers to use browser-specific code to ensure that their pages look and function as intended on all browsers.

Finally, the performance and efficiency of a rendering engine can affect how quickly and smoothly APIs are executed. Some rendering engines may be better optimized for certain types of APIs, or may have better overall performance, which can affect how well web applications and APIs perform on those browsers.

The World Wide Web Consortium (W3C)

The World Wide Web Consortium (W3C) is a global community that works together to develop web standards and guidelines to ensure the long-term growth and stability of the Web **[28]**. It is made up of member organizations, which are typically companies and organizations with an interest in web technologies, as well as individual members.

Major members and contributors of the W3C include:

1. Google
2. Microsoft
3. Mozilla
4. Apple
5. IBM
6. Adobe
7. Facebook
8. Amazon
9. Intel
10. Samsung

In addition to these large technology companies, the W3C also has many smaller organizations and individuals as members, who contribute to the development of web standards and guidelines in a variety of ways. Membership in the W3C allows organizations and individuals to participate in the development of web standards and to stay up-to-date with the latest trends and technologies in web development.

Overall, the W3C membership is diverse and represents a broad range of interests and perspectives in the web development community. This diversity helps to ensure that web standards and guidelines are developed in a collaborative and inclusive manner, with input from a wide range of stakeholders.

3. Methodology

In this subsection, the methodology of this research is presented, aiming to elaborate and present a new and more in-depth analysis of experimental and emerging web APIs. It aims to collect, compare and analyze data that provide an overview on the state of progress for these specific APIs and present a vision of the mobility of the web platform and its progress that can be shared among users and web application developers. Data is strictly collected through Spring of 2023 so as to present the most up-to-date information.

For the purposes of this thesis, experimental and emerging web APIs that are directly, or closely related to machine hardware have been identified, collected and analyzed. These specific APIs allow the browser and web applications access to hardware that either allows for greater functionality and new features, or improves hardware management, ultimately promoting the goal of the web browser as the operating system. These experimental and emerging technologies were identified and subsequently collected from the following sources:

- W3C Specifications: All standards and drafts for Web APIs **[54]**
- Mozilla Developer Network (MDN): Web APIs **[55]**
- Chrome Developer Portal: Documentation, articles and blogs about experimental APIs **[56]**

From the above, the main information source was the W3C. Three reasons contributed to this decision:

- The W3C is the organization that is responsible for the standardization of web technologies and all new and relative information is discussed and recorded there.
- Written specifications for every API can be found on the site in their full form.
- All major browser vendors, as well as major tech corporations have members in the organization and any new technologies are discussed, written and finalized there.

The other two sources of information were very useful as well in order to get more condensed information about the APIs and a lot of useful examples. By using the above information, the following experimental and emerging APIs were identified:

1. Audio Output Devices API
2. Background Sync API

3. Barcode Detection API
4. Compute Pressure API
5. Web Bluetooth API
6. Keyboard API
7. Presentation API
8. Web NFC API
9. WebGPU API
10. Web HID API
11. WebUSB API
12. WebXR Device API

These experimental and emerging APIs are presented in alphabetical order and analyzed according to six thematic sub-categories. These sub-categories are:

- General Description: This analysis involves a basic description of the API and its practical usefulness in as simple terms as possible.
- Interfaces and/or Extensions: A list of programmatic interfaces for the API that can be used in order to enable the functionality that it offers. A short description of the interface is also provided.
- Security Requirements: An analysis of the requirements for the security of the system that uses the particular API. Usually, these requirements are coming from the W3C specification for the API. In some cases, some security requirements are mentioned that are considered best practices. This section helps as well in order to identify potential problems that could hinder the further development and adoption of specific APIs.
- Implementation Example: An example implementation of the API, written in Javascript. A short description and analysis of the code is provided in order for the reader to get a better understanding of what the specific parts of the code do.
- Browser Compatibility: An analysis of the level of support at this time across different browsers. The browsers chosen are: Google Chrome, Microsoft Edge, Opera, Mozilla Firefox and Safari by Apple. Compatibility is reported for Desktop versions of these browsers, as well as their respective mobile versions. One reason for choosing these browsers is that they represent the three most used browser engines: Blink (Chrome, Edge, Opera), Gecko (Firefox) and WebKit

(Safari). Another reason is that these browsers are the most popular browsers used today. This helps to get a general understanding and overview of the adoption rate of experimental APIs across different browser ecosystems and of the way that these new technologies will shape the future of the web. Another important goal is to recognise disagreements among developers of different organizations about the way the APIs should be further developed.

- Usage statistics: This section analyzes statistics about API usage among developers and users across different sources. The aim is to get an overview of the progress made in experimental API development and to get an overview for which APIs are more useful and used at this particular point in time.

Main sources for browser compatibility were the following:

- Can I Use website: The web site "...provides up-to-date browser support tables for support of front-end web technologies on desktop and mobile web browsers." **[57]**
- MDN Web Docs: Under each API, MDN provides compatibility tables in comparison across a wide range of different browsers. **[58]**

Usage statistics were collected from three main sources:

- Chrome Platform Status: HTML & JavaScript usage metrics **[59]**, by using the search function provided by the site. The search was conducted by using supported properties of collected APIs.
- "Can I Use" website **[57]**: By using the search function of the website for specific APIs, one can view usage statistics regarding those web APIs. All experimental APIs are listed and tracked on the website
- "State of JS" website and survey: The "State of JavaScript" is an annual developer survey of the JavaScript ecosystem. One of the topics surveyed is the use of Browser API knowledge and usage. Some statistics can be found regarding a few experimental APIs **[60]**.

3.1. Experimental and Emerging Web APIs

As the web evolves, web browsers are becoming more powerful and are increasingly seen as an operating system in their own right. With this in mind, hardware-related experimental web APIs are being developed that allow web applications to interact with hardware components such as cameras, microphones, and sensors. These APIs can unlock a range of new possibilities for web applications, including augmented and virtual reality experiences, as well as innovative applications for the Internet of Things (IoT). In this context, I will explore experimental and emerging web APIs that are related to hardware and the concept of the browser as an operating system. By using the methodology described in the previous chapter, the following experimental web APIs related to hardware have been identified and subsequently collected and analyzed. The APIs are presented in alphabetical order.

3.1.1 Audio Output Devices API

General Description

The Audio Output Devices API is a browser API that allows web developers to access and manage audio output devices directly from within their web applications **[61]**. This API provides a simple and standardized way for developers to detect and select audio output devices, such as speakers or headphones, and to control the volume and other settings of those devices. As of May 2023, it is an Editor’s Draft on W3C **[62]**.

The Audio Output Devices API is particularly useful for web applications that require audio playback, such as media players or online games. By using this API, developers can ensure that audio is played through the correct output device, and that users can easily switch between different audio devices as needed. Key features of the Audio Output Devices API are:

1. Detection of audio output devices: The API allows developers to detect all available audio output devices on a user's system, including built-in speakers, external speakers, and headphones.
2. Selection of audio output devices: The API allows developers to specify which audio output device to use for audio playback, giving users more control over their audio experience.

3. Control of audio settings: The API allows developers to control the volume, balance, and other settings of audio output devices, ensuring that audio is played at the appropriate level and with the desired balance.
4. Support for multiple audio streams: The API supports multiple audio streams, allowing developers to play different audio streams through different output devices simultaneously.

The Audio Output Devices API is an important tool for web developers who need to ensure reliable and high-quality audio playback in their web applications. It provides a standardized way for developers to interact with audio output devices, making it easier to create web applications that work across a wide range of platforms and devices.

An example implementation of the Audio Output Devices API could involve a web-based media player application that allows users to select and control their audio output devices. Here are the steps involved in implementing this feature using the API:

1. Detect audio output devices: The first step is to use the Audio Output Devices API to detect all available audio output devices on the user's system. This can be done by calling the `navigator.mediaDevices.enumerateDevices()` method, which will return a list of all available audio devices.
2. Allow users to select audio output device: Once the available audio devices have been detected, the user can be presented with a list of options to select from. This can be done using a dropdown menu or other interface element that allows the user to choose their preferred audio output device.
3. Control audio settings: After the user has selected their preferred audio output device, the application can use the Audio Output Devices API to control the volume and other settings of that device. This can be done using the `AudioContext.createGain()` method to create a gain node, which can then be used to adjust the volume of the audio output.
4. Play audio through selected device: Finally, the media player application can use the Audio Output Devices API to ensure that audio is played through the selected output device. This can be done using the `AudioContext.destination` property to specify the output device for the audio stream.

Overall, the Audio Output Devices Browser API is a powerful tool for web developers who need to ensure reliable and high-quality audio playback in their web applications. By providing a standard way to detect and select audio output devices, this API makes it easier to create web applications that work across a wide range of platforms and devices, while also giving users more control over their audio experience.

Extensions

The Audio Output Devices API extends the following APIs, by adding the listed features:

`MediaDevices.selectAudioOutput()`: This method prompts the user to select a specific audio output device, for example a speaker or headset.

`HTMLMediaElement.setSinkId()`: This method sets the ID of the audio device to use for output, which will be used if permitted.

`HTMLMediaElement.sinkId`: This property returns the unique ID of the audio device being used for output, or an empty string if the default user agent device is being used.

Interfaces

The Audio Output Devices API provides several interfaces to interact with audio output devices, the most important of which are:

1. `MediaDevices`: This interface represents a collection of audio input and output devices available on the user's system. It provides methods to enumerate and query the available devices.
2. `MediaDeviceInfo`: This interface represents a media input or output device. It provides properties to obtain information about the device such as the device ID, device label, and device kind (audio input or audio output).
3. `MediaStreamTrack`: This interface represents a single audio track in a media stream. It provides methods to start and stop audio capture and to set the volume and other audio properties.

4. `AudioDestinationNode`: This interface represents the final destination of an audio stream. It provides methods to control the volume and other audio properties of the output stream.
5. `AudioNode`: This interface represents an audio-processing node in the audio graph. It provides methods to connect and disconnect nodes in the audio graph, and to process audio data.

Security requirements

Access to `MediaDevices.selectAudioOutput()` is subject to the following constraints:

- It may only be used in a secure context.
- Access may be gated by the speaker-selection HTTP Permission Policy.
- Transient user activation is required. The user has to interact with the page or a UI element for this feature to work.

The user must explicitly grant permission to use the audio output device through a user-agent specific mechanism, or have previously granted permission. Note that if access is denied by a permission policy it cannot be granted by a user permission.

User permission to set the output device is also implicitly granted if the user has already granted permission to use a media input device in the same group, using `MediaDevices.getUserMedia()`.

Other methods/properties also require a secure context and the speaker-selection permission policy.

The permission status can be queried using the Permissions API method `navigator.permissions.query()`, passing a permission descriptor with the speaker-selection permission.

Implementation Example

```
// Get all available audio output devices
navigator.mediaDevices.enumerateDevices().then(function(devices
```

```
) {
  var audioOutputDevices = devices.filter(function(device) {
      return device.kind === 'audiooutput';
  });

  // Display a list of available audio output devices
  var deviceList = document.getElementById('device-list');
  audioOutputDevices.forEach(function(device) {
      var option = document.createElement('option');
      option.value = device.deviceId;
      option.text = device.label;
      deviceList.appendChild(option);
  });

  // Set the selected output device
  var selectedDeviceId = deviceList.value;

  // Create an AudioContext object
  var audioContext = new AudioContext();

  // Set the output device for the AudioContext object
  var audioOutput =
audioContext.createMediaStreamDestination();
  var stream = audioOutput.stream;
  var audioOutputDevice =
audioOutputDevices.find(function(device) {
      return device.deviceId === selectedDeviceId;
  });
  if (audioOutputDevice) {
      stream.getAudioTracks().forEach(function(track) {
      track.applyConstraints({
      deviceId: audioOutputDevice.deviceId
      });
      });
  }
  audioContext.destination = audioOutput;

  // Play audio through the selected output device
  var audio = new Audio('example-audio.mp3');
  audio.play();
});
```

In this example, the `navigator.mediaDevices.enumerateDevices()` method is used first, in order to get a list of all available audio output devices. We then filter the list to include only audio output devices and display them in a dropdown menu on the web page.

When the user selects an audio output device from the dropdown menu, we create an `AudioContext` object and set the output device for that object using the `createMediaStreamDestination()` method. We then create an `Audio` object and play an example audio file through the selected output device.

This example implementation shows how the Audio Output Devices API can be used to detect and select audio output devices, control their settings, and ensure that audio is played through the correct device.

Browser Compatibility

Implementation among browsers varies between the three different extensions of the API: `MediaDevices.selectAudioOutput()`,`HTMLMediaElement.setSinkId()` and `HTMLMediaElement.sinkId.`
Firefox Nightly supports all three extensions from version 88, however the feature is behind the `media.setsinkid.enabled` preference which needs to be set to `true`. Google Chrome, Microsoft Edge and Opera browsers support two out of the three, `HTMLMediaElement.setSinkId()` and `HTMLMediaElement.sinkId.` All support is only on the desktop versions of those browsers. No support exists at this time for Mobile browsers because of a limitation in Android **[63]**. Safari does not support the API in either its desktop or mobile version. The tables below show browser support for Audio Output Devices API **[61, 64, 65]**:

Audio Output Devices API: `MediaDevices.selectAudioOutput()`			
Browser	Version	Desktop Support	Mobile Support
Google Chrome	Not Supported	No	No
Microsoft Edge	Not Supported	No	No
Opera	Not Supported	No	No
Mozilla Firefox	Nightly	Yes	No

Safari	Not supported	No	No

Table 3-1: MediaDevices.selectAudioOutput() browser support

Audio Output Devices API: `HTMLMediaElement.setSinkId()`			
Browser	Version	Desktop Support	Mobile Support
Google Chrome	49+	Yes	No
Microsoft Edge	17+	Yes	No
Opera	36+	Yes	No
Mozilla Firefox	Nightly	Yes	No
Safari	Not supported	No	No

Table 3-2: HTMLMediaElement.setSinkId() browser support

Audio Output Devices API: `HTMLMediaElement.sinkId`			
Browser	Version	Desktop Support	Mobile Support
Google Chrome	49+	Yes	No
Microsoft Edge	17+	Yes	No
Opera	36+	Yes	No
Mozilla Firefox	Nightly	Yes	No
Safari	Not supported	No	No

Table 3-3: HTMLMediaElement.sinkId browser support

Usage Statistics

According to the website caniuse.com **[64, 65]**:

- `MediaDevices.selectAudioOutput()` has a global usage percentage across all users of 0%
- `HTMLMediaElement.setSinkId()` and `HTMLMediaElement.sinkId` both show a global percentage of 29.68%

Data from Google's Chrome Platform Status exists only for the `HTMLMediaElementSetSinkId` extension. They show a very small percentage of page loads in Chrome, that at its highest point in May 2021 reached 0.25% **[66]**.

The chart below shows the percentage of page loads in Chrome that use this feature at least once. Data is across all channels and platforms. Newly added use counters that are not on Chrome stable yet only have data from the Chrome channels they're on.

Figure 3-1: HTMLMediaElementSetSinkId percentage of page loads over time in Chrome.

Below is a table that shows the top five sample URLs from Chrome Platform Status **[66]**.

`HTMLMediaElementSetSinkId`: Top Five sample URLs
https://portal.video.quironsalud.es/
https://sip.myvideo.ge.com/
https://telehealth.health.qld.gov.au/
https://www.cabpolidiagnostico.it/
https://conference.meet.health.nsw.gov.au/

Table 3-4: HTMLMediaElementSetSinkId top Five sample URLs

3.1.2 Background Synchronization API

General Description

The Background Sync Browser API is an important web browser API that enables web applications to synchronize data in the background, even if the user has closed the web page or gone offline **[67, 68, 70]**. This API is a powerful tool for web developers who need to provide a seamless offline experience for their users. The API has the status of "Unofficial Draft" on the W3C **[67]**.

One of the key benefits of the Background Sync API is that it provides a way for web applications to update their data even when the user is not actively using the application. This makes it possible to create web applications that work reliably across a wide range of devices and network conditions, and ensures that the user's data is always up-to-date, even if they lose their network connection or switch to a different device.

An important feature of the Background Sync API is that it is resilient to network failures. For example, if the device goes offline or loses its network connection, the API will automatically retry the sync operation when the network becomes available again. This ensures that the user's data is always synced, even in challenging network conditions.

Extensions

The following additions to the Service Worker API provide an entry point for setting up background synchronization:

- `ServiceWorkerRegistration.sync` - Read only

Returns a reference to the `SyncManager` interface for registering tasks to run once the device has network connectivity.

- `ServiceWorkerGlobalScope.sync` - event

An event handler fires whenever a `sync` event occurs. This happens as soon as the network becomes available.

Interfaces

The Background Sync API is designed to enable web developers to defer actions until a user has stable connectivity. This API provides interfaces that allow web pages to register and dequeue background sync tasks that can be executed even when the web page or the browser is closed.

The main interfaces provided by the Background Sync API are:

1. `SyncManager`: This interface is responsible for managing sync events and their lifecycle. It is responsible for registering, fetching, and removing sync events.
2. `SyncEvent`: This interface represents a sync event, which is a request to perform some action when the device is online. It contains a tag, which can be used to identify the event, and a `lastChance` attribute that indicates whether this is the last chance to process the event before it is discarded.
3. `ExtendableEvent`: This interface is used as a base interface for several other interfaces, including `SyncEvent`. It represents an event that can be extended with additional information.
4. `ExtendableMessageEvent`: This interface extends `ExtendableEvent` and represents an event that contains a message.
5. `ExtendableEventInit`: This interface represents the initialization parameters for an `ExtendableEvent`. It can be extended to include additional parameters for other events.
6. `SyncEventInit`: This interface represents the initialization parameters for a `SyncEvent`. It extends `ExtendableEventInit` and includes a tag and a `lastChance` attribute.
7. `ServiceWorkerGlobalScope`: This interface represents the global scope of a service worker, which can be used to register a sync event and handle the event when it fires.

Security Requirements

The Background Sync API allows web applications to synchronize data in the background, even when the user is not actively interacting with the application. However,

this API can also pose security risks if not used properly. To ensure security, the following requirements should be followed when using the Background Sync API:

1. Use HTTPS: The Background Sync API is only available on secure (HTTPS) connections. This ensures that any data transferred between the client and server is encrypted and cannot be intercepted by third parties.
2. Limit access to sensitive data: Only store data that is necessary for background sync and limit access to sensitive data. Also, ensure that any sensitive data is encrypted before being stored on the device.
3. Validate input data: Ensure that the input data received from the server is valid before processing it. This will prevent any malicious data from being processed and executed.
4. Use proper authentication and authorization: Authenticate and authorize users before allowing them to access the background sync feature. This ensures that only authorized users can access the feature and synchronize data.
5. Keep track of synchronization status: Keep track of the synchronization status and inform the user if there are any errors or problems with the synchronization process.

Implementation Specifics

In terms of implementation, the Background Sync API is relatively easy to use for web developers. The API provides a simple way to register a sync event and handle the sync operation in a service worker. Web developers can use JavaScript to register the sync event and implement the sync logic, and then update the user interface to provide a seamless offline experience for users.

An example implementation of the Background Sync API could involve a web-based note-taking application that allows users to create and edit notes even when they are offline **[69]**. Here are the steps involved in implementing this feature using the API:

1. Register a service worker: The first step is to register a service worker for the web application. This can be done using the `navigator.serviceWorker.register()` method, which registers a JavaScript file as a service worker.

2. Implement background sync logic: Once the service worker is registered, it can be used to implement background sync logic. This can be done using the `SyncManager.register()` method, which registers a sync event that will be triggered when the device comes back online. In the event handler, the application can retrieve any unsynced notes from a local database or cache, and then send them to the server.
3. Set up user interface: After the background sync logic is implemented, the web application can be updated to provide a seamless offline experience for users. This can involve displaying a message to the user when they are offline, allowing them to create and edit notes, and automatically syncing those notes in the background when the user comes back online.

Implementation Example

The following code demonstrates how to register a sync event using the Background Sync API:

```
navigator.serviceWorker.ready.then(function(registration) {
  if ('sync' in registration) {
      registration.sync.register('sync-notes').then(function() {
      console.log('Sync event registered');
      }).catch(function() {
      console.error('Failed to register sync event');
      });
  }
});
```

In this example, we use the `navigator.serviceWorker.ready` method to wait until the service worker is ready, and then we check if the `sync` event is supported by the browser. If it is supported, we register a sync event named `sync-notes` using the `registration.sync.register()` method. When the device comes back online, this event will be triggered and the application can sync any unsynced notes to the server.

Browser Compatibility

The tables below show support for the Background Sync API across various browsers **[71, 68]**.

Background Sync API: `SyncManager`, `getTags`, `register`			
Browser	Version	Desktop Support	Mobile Support
Google Chrome	49+	Yes	Yes
Microsoft Edge	79+	Yes	Yes
Opera	36+	Yes	Yes
Mozilla Firefox	Not supported	No	No
Safari	Not supported	No	No

Table 3-5: SyncManager, getTags, register browser support

Background Sync API: `in service workers`			
Browser	Version	Desktop Support	Mobile Support
Google Chrome	61+	Yes	Yes
Microsoft Edge	79+	Yes	Yes
Opera	Not supported	No	No
Mozilla Firefox	Not supported	No	No
Safari	Not supported	No	No

Table 3-6: Background Sync in service workers browser support

Usage Statistics

According to caniuse.com **[71]**, the Background Sync API has a global usage percentage of 72-75%, depending on the subsystem. This reflects the relatively good support that the API has in web browsers at this time.

According to Google's Chrome Platform Status **[72, 73]**, `BackgroundSync` and `PeriodicBackgroundSync`, have the following page loads in Chrome over the last three years:

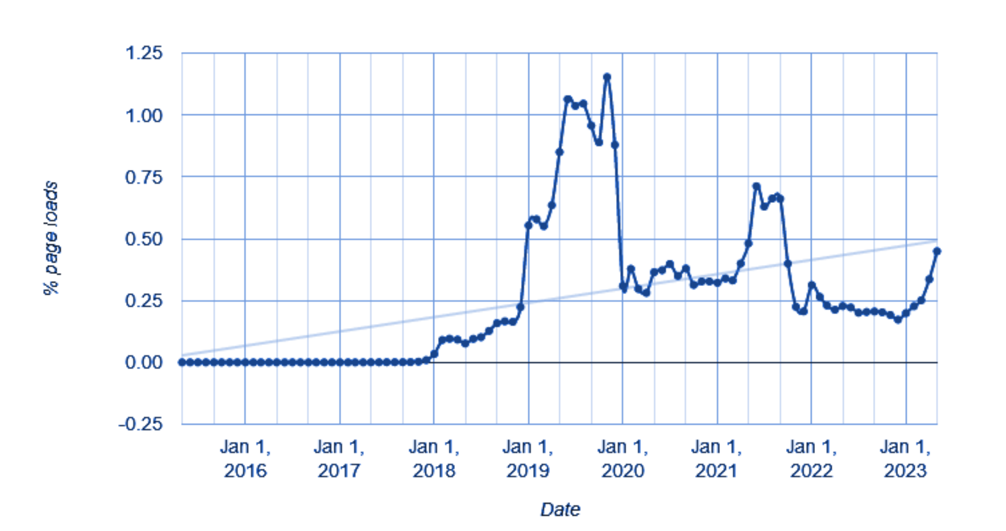

figure 3-2: BackgroundSync percentage of page loads over time in Chrome.

In the table below, the top five sample URLs for Background Sync are listed **[72]**.

`BackgroundSync`: Top five sample URLs
http://www.trivago.dk/
http://www.trivago.cl/
http://www.trivago.hk/
http://www.trivago.co.uk/
http://www.trivago.ru/

Table 3-7: BackgroundSync top five sample URLs

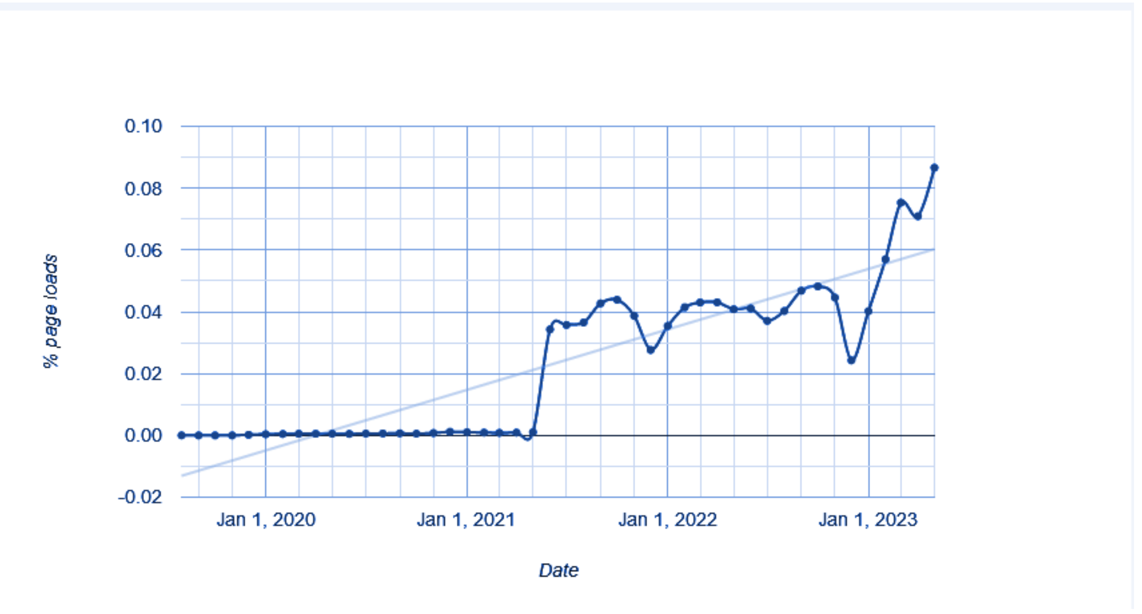

Figure 3-3: PeriodicBackgroundSync percentage of page loads over time in Chrome.

`PeriodicBackgroundSync`: Top five sample URLs
https://m.gelonghui.com/
https://app.nanno.com/
https://tidymails.com/
https://vivekananda.live/
https://driver.eurotunnelfreight.com/

Table 3-8: PeriodicBackgroundSync top five sample URLs

3.1.3 Barcode Detection API

General Description

The Barcode Detection Browser API is a relatively new web browser API that provides a way for web applications to detect and decode barcodes using the device's camera **[74, 75]**. This API is an important tool for developers who need to create web applications that can read barcodes, such as retail or inventory management applications. It is a part of the Shape Detection API and it is designated as a W3C Community Group Draft Report **[74]**.

The Barcode Detection API works by capturing a video stream from the device's camera and analyzing it for barcode patterns. When a barcode is detected, the API provides information about the type of barcode (e.g. UPC, QR code, etc.) and the decoded value of the barcode.

One of the key benefits of the Barcode Detection API is that it provides a way for web applications to perform barcode scanning without the need for a native app. This can significantly reduce the development time and cost of creating barcode scanning applications, as developers can leverage the power of the web platform to build their applications.

Another important feature of the Barcode Detection API is that it is designed to be fast and efficient. The API is optimized for real-time processing of video streams, and can detect barcodes quickly and accurately, even in challenging lighting conditions.

The barcode detection web API supports several 1D and 2D barcode formats, including EAN-13, EAN-8, UPC-A, UPC-E, Code-39, Code-93, Code-128, ITF, Codabar, QR Code, Data Matrix, and PDF417 **[74, 75, 76]**.

Overall, the Barcode Detection Browser API is an important tool for web developers who need to create web applications that can read barcodes. By providing a fast, efficient, and platform-agnostic way to perform barcode scanning, this API can significantly reduce the development time and cost of creating barcode scanning applications, and help businesses to streamline their inventory management and retail operations.

Interfaces

The Barcode Detection API provides two main interfaces:

1. `BarcodeDetector` interface: This interface provides methods for detecting barcodes in an image. It has the following methods:
    - `constructor()`: Creates a new instance of `BarcodeDetector`.
    - `detect(image: ImageBitmapSource): Promise<Barcode[]>`: Takes an `ImageBitmapSource` object and returns a promise that resolves to an array of `Barcode` objects, each representing a detected barcode in an image.
2. `Barcode` interface: This interface represents a detected barcode and provides information about its type and data. It has the following properties:
    - `rawValue`: A string representing the raw, unprocessed data encoded in the barcode.
    - `text`: A string representing the processed, human-readable data encoded in the barcode.
    - `format`: A string representing the format of the barcode (e.g. "QR_CODE", "EAN_13", etc.).
    - `cornerPoints`: An array of four `DOMPoint` objects representing the corners of the bounding box of the barcode in the image.

Note that the `BarcodeDetector` interface is not available in workers, so barcode detection can only be performed in the main thread.

Security Requirements

The Barcode Detection API has security requirements that are similar to other web APIs. As it deals with sensitive user information, it is important to ensure that this data is not accessible by unauthorized parties.

One of the key security requirements for the Barcode Detection API is that it must only be accessible over HTTPS connections. This is because HTTPS encrypts all data transmitted between the user's device and the web server, making it much more difficult for attackers to intercept and access sensitive information.

Another important security requirement is that the API must not allow access to any information that is not explicitly authorized by the user. This means that the user must grant explicit permission for the web application to access the camera and perform barcode detection.

The API should also limit the amount of information that is stored locally, and ensure that any data that is stored is encrypted to prevent unauthorized access. Additionally, the API must be designed to prevent injection attacks and other types of web-based attacks that could be used to bypass security measures and gain access to sensitive information.

Implementation Example

In terms of implementation, the Barcode Detection API can be used in conjunction with the `getUserMedia` API to access the device's camera and capture a video stream **[76]**. Below is an example implementation with JavaScript:

```
const barcodeDetector = new BarcodeDetector({ formats:
['ean_13', 'qr_code'] });
const video = document.querySelector('video');

navigator.mediaDevices.getUserMedia({ video: true })
  .then(stream => {
      video.srcObject = stream;
      video.onloadedmetadata = () => {
      video.play();
      const canvas = document.createElement('canvas');
      canvas.width = video.videoWidth;
      canvas.height = video.videoHeight;
      const ctx = canvas.getContext('2d');

      const detectBarcode = () => {
      ctx.drawImage(video, 0, 0, canvas.width, canvas.height);
      barcodeDetector.detect(canvas)
            .then(barcodes => {
            if (barcodes.length > 0) {
            console.log(`Detected barcode type:
${barcodes[0].format}`);
            console.log(`Detected barcode value:
${barcodes[0].rawValue}`);
```

```
            }
            requestAnimationFrame(detectBarcode);
            })
            .catch(err => {
            console.error(err);
            requestAnimationFrame(detectBarcode);
            });
      };
      detectBarcode();
      };
  })
  .catch(err => {
      console.error(err);
  });
```

In this example, we first create a new instance of the `BarcodeDetector` class and specify which barcode formats we want to detect. We then use the `getUserMedia` API to access the device's camera and capture a video stream, and create a canvas element to draw the video frames onto.

We then define a function called `detectBarcode` that draws the current video frame onto the canvas, and calls the detect method of the `BarcodeDetector` object to detect any barcodes in the frame. If a barcode is detected, we log its type and decoded value to the console.

Finally, we call the `detectBarcode` function in a loop using the `requestAnimationFrame` API, to continuously process video frames and detect any barcodes that appear.

Browser Compatibility

The Barcode Detection API is still experimental and not yet available in all web browsers. As of April 2023, it is supported by Google Chrome, Microsoft Edge and Opera mobile web browsers, while it has only partial support in the respective desktop versions. It is not supported in Firefox, Safari, or any other web browsers at this time. The support for this API is also dependent on the operating system, with some features not being available on certain platforms. The feature depends on built-in face detection support which is only provided by Android and macOS at the moment. Support is shown on the table below **[75, 77]:**

Barcode Detection API: `BarcodeDetector, BarcodeDetector() constructor, detect, getSupportedFormats`			
Browser	Version	Desktop Support	Mobile Support
Google Chrome	88+ | 83+	Partial	Yes
Microsoft Edge	89+	Partial	Yes
Opera	69+ | 59+	Partial	Yes
Mozilla Firefox	Not supported	No	No
Safari	Not supported	No	No

Table 3-9: BarcodeDetector, BarcodeDetector() constructor, detect, getSupportedFormats browser support

Usage Statistics

The fact that the Barcode Detection API is (at least partially) supported in most major web browsers, as well as the fact that it is particularly useful on mobile devices, gives it a relatively high usage percentage. According to caniuse.com, global mobile browser usage is 43.34% **[77]**. On the other hand, the website also reports a global usage of 28.71% in partially supported desktop browsers. The total percentage of desktop and mobile usage is at 72.05%

However, according to Chrome Platform Status page loads for `BarcodeDetectorDetect` appear to be minimal, just touching 0.0005% **[78]**

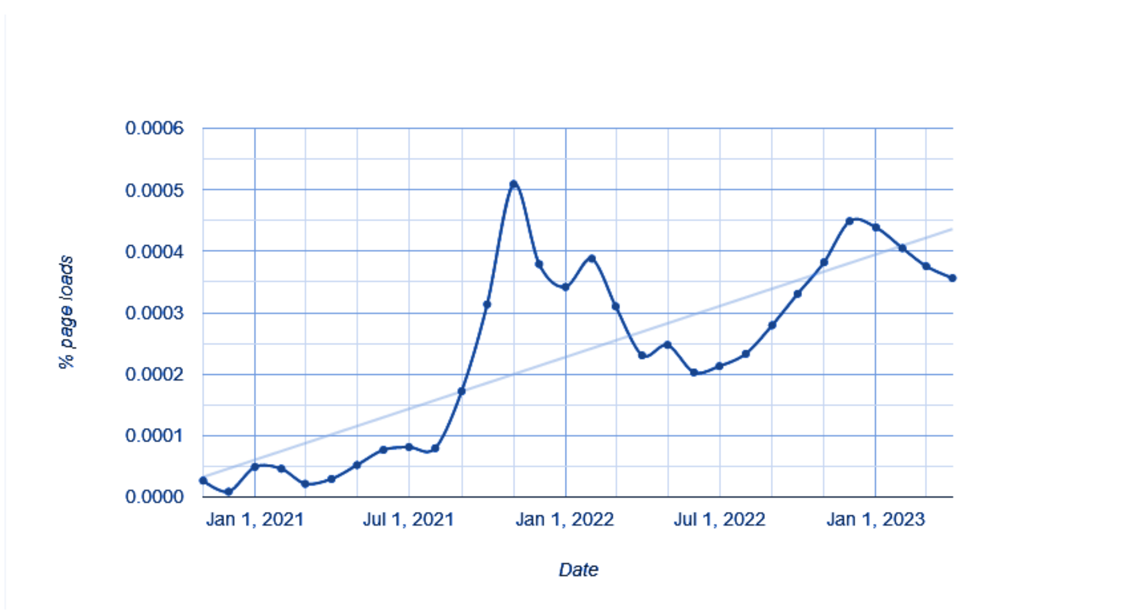

Figure 3-4: BarcodeDetectorDetect percentage of page loads over time in Chrome.

Similar results appear below for `ShapeDetection_BarcodeDetectorConstructor` **[79]**

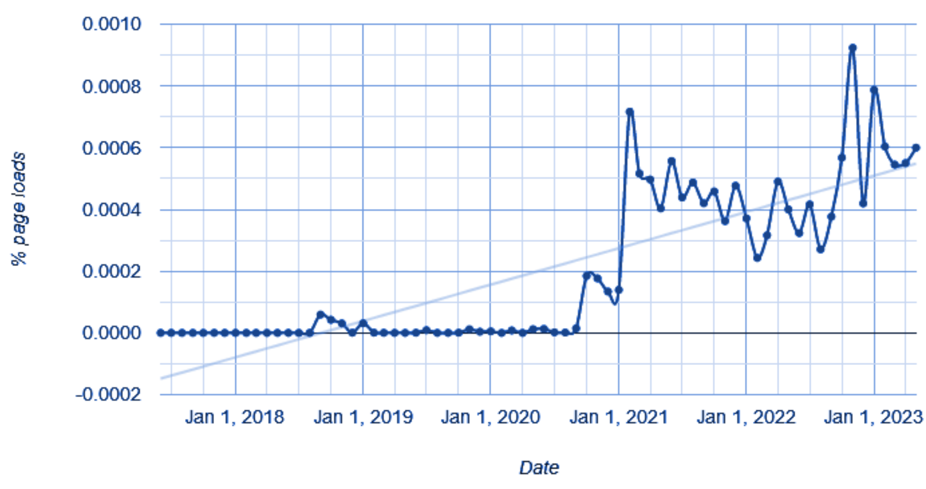

Figure 3-5: ShapeDetection_BarcodeDetectorConstructor percentage of page loads over time in Chrome.

No sample URLs exist for Barcode Detection API **[78]**:

`BarcodeDetectorDetect`: Top five sample URLs
NO DATA AVAILABLE

Table 3-10: BarcodeDetectorDetect top five sample URLs

3.1.4 Compute Pressure API

General Description

The Compute Pressure API provides a way for software to detect the level of pressure on a system, and can use the appropriate hardware metrics to ensure optimal processing performance without overburdening the system **[81, 82]**. This ensures that users can make the most of the processing power available to them without causing undue stress on the system.

The Compute Pressure API is particularly useful for improving the performance of real-time applications such as video games and video conferencing, which are considered "soft" applications. These types of applications may experience reduced quality if the system is pushed beyond a certain limit, but it usually does not result in a complete system failure. By using the Compute Pressure API, these applications can adjust their workload based on the level of CPU pressure, resulting in better overall performance.

The Compute Pressure API is designed to facilitate adaptation decisions for video conferencing and video games **[82]**. For video conferencing, it can adjust the number of video feeds displayed, reduce the quality of video processing, skip non-essential processing, and adjust quality-versus-speed settings for video and audio encoding. For video games, it can use lower-quality assets, disable non-essential effects, and adjust quality-versus-speed settings in the rendering engine. These adaptations can be made by tracking thermal and CPU pressure states of the main thread and workers being used, which can be affected by other apps and sites.

Particular scenarios in which the Compute Pressure API could prove to be useful:

In Video Conferencing:

- Adjust the number of video feeds shown simultaneously during calls with many participants.
- Reduce the quality of video processing (video resolution, frames per second).
- Skip non-essential video processing, such as some camera filters.
- Disable non-essential audio processing, such as WebRTC noise suppression.

- Turn quality-versus-speed and size-versus-speed knobs towards "speed" in video and audio encoding (in WebRTC, WebCodecs, or software encoding).

In Video Games:

- Use lower-quality assets to compose the game's video (3D models, textures, shaders) and audio (voices, sound effects).
- Disable effects that result in less realistic non-essential details (water, cloth, fire animations, skin luminance, glare effects or physical simulations that don't impact gameplay).
- Tweak quality-versus-speed knobs in the game's rendering engine (shadows quality, texture filtering, view distance).

Interfaces

The Compute Pressure API can be run in the following contexts:

- Window or main thread
- Dedicated Worker
- Shared Worker

The Compute Pressure API defines two new interfaces.

- `PressureObserver`: An object to observe the compute pressure of any number of sources at a predefined sample rate. First iteration in Chromium exposes `cpu` as `source`.
- `PressureRecord`: Describes the pressure trend at a specific moment of transition. Objects of this type can only be obtained in two ways: as an input to your PressureObserver callback, or by calling the `takeRecords()` method on the `PressureObserver` instance.

When a `PressureObserver` object is created, it's configured to watch the pressure of supported sources, at a given sample rate. The supported sources can be individually observed or unobserved at any time during the lifetime of the

`PressureObserver` object. The sample rate cannot be changed after the creation of the object.

- `PressureObserver(callback, options)`: Creates a new `PressureObserver` object which will invoke a specified callback function when it detects that a change in the values of the source being observed has happened.

The constructor takes a mandatory callback function and optional options, as parameters.

- `callback()`: The callback is called with an array of unread `PressureRecord` objects.
- `PressureObserverOptions`: Contains the sample rate, `sampleRate` in Hz, at which the user requests updates.

Methods

- `PressureObserver.observe(source)`: Tells the 'PressureObserver' which source to observe.
- `PressureObserver.unobserve(source)`: Tells the 'PressureObserver' to stop observing a source.
- `PressureObserver.disconnect()`: Tells the 'PressureObserver' to stop observing all sources.
- `PressureObserver.takeRecords()`: Returns a sequence of records, since the last callback invocation.
- `static PressureObserver.supportedSources()` (read only): Returns supported source types by the hardware.

Parameters

- `source`: The source to be observed, for example: `cpu`.
    - In the current version of Compute Pressure, only `cpu` is supported.

The `PressureRecord` interface of the Compute Pressure API describes the pressure trend of a source at a specific moment of transition.

Instance Properties

- `PressureRecord.source` (Read-only): Returns a string representing the origin source from which the record is coming.
- `PressureRecord.state` (Read-only): Returns a string representing the pressure state recorded.
- `PressureRecord.time` (Read-only): Returns a number representing a high resolution timestamp.

Security Requirements

The policy, according to the W3C Working Draft, is that information exposure should be minimized, because: "Exposing hardware related events related to low-level details such as exact CPU utilization or frequency increases the risk of harming the user's privacy" **[84]**.

To mitigate this risk, no such low level details should be exposed.

At a high level, the information exposed is reduced by the following steps:

- **Rate-limiting**: The user agent notifies the application of changes in the information it can learn. Change notifications are rate-limited.
- **Same-origin context [80]**: The feature is only available in same-origin contexts by default, but can be extended to third-party contexts such as iframes via a permission policy.

Rate-limiting change notifications

The objective is to prevent the observation of the exact moment when a value changes between two states. Once the pressure observer is activated, it will be called with initial values and then called when the values change, but with a limit on the frequency of subsequent calls. The most recent value is reported during each call. To protect user privacy, the API will recommend a rate limit of one call per second for the active window and one call per 10 seconds for other windows. Jittering the call timings across origins is also recommended to prevent device identification across multiple origins. These measures also enhance user security by preventing timing attacks and limiting the performance overhead of the API. The user agent can implement rate limiting, or it can

be done by adjusting the polling or sampling rate of the underlying hardware counters, if accessed through a higher-level framework.

No side-channels

The Compute Pressure API poses a risk of identifying users across multiple sites if precise or unique values are accessible by different sites that do not share the same origin **[80]**. If two sites observe the same pressure state and timestamp, it can be assumed that they are being used by the same user on the same device. To prevent this, the API restricts reporting pressure state changes to one origin at a time. Typically, this is achieved by only reporting changes to the focused page. However, video conferencing sites, which are major users of this API, need to ensure that video streams and effects do not negatively affect the system. In these cases, the site may not be focused, but pressure state changes should still be reported.

- One scenario is when the user is taking notes during a meeting and the video conferencing site is not in the foreground. In this case, the video stream may only be visible in a small window within the main application.
- Another one, is when the user is sharing an external application window, such as a presentation or the entire screen, which is typically indicated by some form of user interface element.

For this reason, the API considers these two cases to have higher priority than whether the site is focused.

Same-origin contexts

By default, the Compute Pressure API only allows data delivery to documents that have the same origin as the initiator of an active picture-in-picture session, documents that are capturing the system focus, or the currently focused document. These qualifying documents can delegate data delivery to child documents that are part of their navigation hierarchy.

Implementation Example

First, we run a check if the Compute Pressure API is supported:

```
if ('PressureObserver' in globalThis) {
  // The Compute Pressure API is supported.
}
```

Then, we create the pressure observer by calling its constructor with a callback function to be run whenever there is a pressure update:

```
const observer = new PressureObserver(
  (records) => { /* ... */ },
  { sampleRate: 0.5 }
);
```

A sample rate, `sampleRate`, of 0.5 Hz, means that there will be updates at most every two seconds.

If the sample rate requested cannot be served by the system. The system will provide samples at the best suitable rate that exists. For example if the rate of 2 Hz is requested, but the system can only provide samples at maximum 1 Hz, 1 Hz will be selected.

Then, we start a pressure observer. For each source, we call `observer.observe(source)`.

```
observer.observe("cpu");
```

In this example the `cpu` is the pressure source we are interested in. For now, it is the only source available. In the future, there may be other sources such as `gpu`, `power` or `thermals`.

To stop observing a source, we use the `unobserve()` method, as in the following example:

```
observer.unobserve("cpu");
```

In order to "unobserve" all sources at once, we have to use the `disconnect()` method, as in the following example:

```
observer.disconnect();
```

Finally, we can retrieve pressure records with a callback function, which will be invoked every time a change is happening in the pressure state:

```
function callback(records) {
 const lastRecord = records[records.length - 1];
 console.log(`Current pressure ${lastRecord.state}`);
 if (lastRecord.state === "critical") {
   // Reduce workers load by 4.
 } else if (lastRecord.state === "serious") {
   // Reduce workers load by 2.
 } else {
   // Do not reduce.
 }
}

const observer = new PressureObserver(callback, { sampleRate: 1
});
await observer.observe("cpu");
```

We can also force the reading of `PressureRecord` by calling the `takeRecords()` method.

The `takeRecords()` method of the `PressureObserver` interface returns an array of `PressureRecords` objects stored in the pressure observer, emptying it out.

The most common use case for this is to immediately fetch all pending pressure records, not yet processed by the observer's callback function, prior to disconnecting the observer, so that any pending records can be processed when shutting down the observer.

Calling the method below clears the pending records list, so the callback will not be run:

```
const observer = new PressureObserver(
 (records) => { /* Do something with records. */ },
 { sampleRate: 1 }
);

await observer.observe("cpu");

setTimeout(() => {
 // Forced records reading.
 const records = observer.takeRecords();
```

```
 observer.disconnect();
 // Do something with the last records if any.
}, 2000);
```

Browser Compatibility and Usage Statistics

The Compute Pressure API is still in the very early stages of development and as such, it is not implemented on any browsers.

It is specified as a Working Draft by the W3C **[81]**. According to Google Chrome Developers Portal, the API is the process of gathering feedback and iterating on the design. Next steps will include an origin trial and finally an official launch as experimental technology at some point in the future **[82]**.

3.1.5 Web Bluetooth API

General Description

The Web Bluetooth API is a web browser API that provides a way for web applications to interact with nearby Bluetooth devices **[83]**. This API is useful for creating web applications that need to communicate with Bluetooth devices, such as IoT devices, smart watches, or fitness trackers.

The Web Bluetooth API works by exposing the Bluetooth functionality of the device to web applications through a set of JavaScript APIs **[87]**. This allows developers to discover nearby Bluetooth devices, connect to them, and exchange data with them directly from their web applications.

One of the key benefits of the Web Bluetooth API is that it allows web applications to access Bluetooth devices without requiring a native app. This can significantly reduce the development time and cost of creating Bluetooth-enabled applications, as developers can leverage the power of the web platform to build their applications.

Another important feature of the Web Bluetooth API is that it provides a way for web applications to interact with a wide range of Bluetooth devices, regardless of the underlying hardware or operating system. This makes it easier for developers to create cross-platform Bluetooth applications that can run on a variety of devices and operating systems.

Overall, the Web Bluetooth Browser API is an important tool for web developers who need to create web applications that can communicate with Bluetooth devices. By providing a platform-agnostic way to perform Bluetooth communication, this API can significantly reduce the development time and cost of creating Bluetooth-enabled applications, and help businesses to build more flexible and scalable IoT solutions.

Interfaces

The Web Bluetooth API provides interfaces for web applications to discover and communicate with nearby Bluetooth Low Energy (BLE) devices. It consists of two main interfaces:

1. `Bluetooth`: The main entry point for using the Web Bluetooth API. This interface provides methods for scanning and connecting to nearby BLE devices.
2. `BluetoothDevice`: Represents a single BLE device that has been discovered by the `Bluetooth` interface. This interface provides methods for connecting, reading, writing, and receiving notifications from the device.

In addition to these main interfaces, there are also several supporting interfaces for interacting with Bluetooth-related objects:

1. `BluetoothRemoteGATTServer`: Represents a remote GATT server on a BLE device. This interface provides methods for discovering services and characteristics on the server, and for reading, writing, and receiving notifications from the characteristics.
2. `BluetoothRemoteGATTService`: Represents a GATT service on a remote GATT server. This interface provides methods for discovering characteristics on the service.

3. `BluetoothRemoteGATTCharacteristic`: Represents a GATT characteristic on a remote GATT server. This interface provides methods for reading, writing, and receiving notifications from the characteristic.
4. `BluetoothUUID`: A helper object that provides UUIDs for various Bluetooth profiles and services, such as the Heart Rate Monitor profile or the Generic Attribute service.

Security Requirements

The Web Bluetooth API allows web applications to communicate with nearby Bluetooth Low Energy (BLE) devices. As such, it has some security requirements to ensure the safety of users and their devices **[83]**:

1. Secure Context: The Web Bluetooth API requires a secure context, which means that it can only be used on web pages served over HTTPS or on localhost. This helps prevent man-in-the-middle attacks.
2. User Permission: The API requires user permission to access BLE devices, and the user is shown a prompt to allow or deny access. This ensures that the user is aware of the application's access to their devices and can choose to allow or deny it.
3. Device Pairing: The API uses the standard BLE pairing process to establish a secure connection between the device and the web application. This ensures that the communication between the device and the web application is encrypted and secure.
4. Service UUID: The Web Bluetooth API requires that the UUID of the Bluetooth service being accessed is known and specified in the code. This prevents unauthorized access to other services on the device.
5. Timeouts: The API has built-in timeouts for operations, which helps prevent denial-of-service attacks.

By adhering to these security requirements, the Web Bluetooth API can help ensure that web applications can communicate with nearby Bluetooth devices in a safe and secure manner.

Implementation Example

In terms of implementation, the Web Bluetooth API can be used to perform a variety of Bluetooth-related tasks, such as discovering nearby Bluetooth devices, connecting to a specific device, and exchanging data with the device **[86]**. Below is an example implementation using JavaScript:

```
navigator.bluetooth.requestDevice({
  filters: [
     { services: ['heart_rate'] }
  ]
})
.then(device => {
  console.log('Device found: ' + device.name);

  device.addEventListener('gattserverdisconnected', event => {
     console.log('Device disconnected');
  });

  return device.gatt.connect();
})
.then(server => {
  console.log('Connected to GATT server');

  return server.getPrimaryService('heart_rate');
})
.then(service => {
  console.log('Heart rate service found');

  return service.getCharacteristic('heart_rate_measurement');
})
.then(characteristic => {
  console.log('Heart rate measurement characteristic found');

  characteristic.startNotifications().then(_ => {
     console.log('Notifications started');


characteristic.addEventListener('characteristicvaluechanged',
     event => {
     const value = event.target.value;
```

```
      const heartRate = value.getUint8(1);
      console.log('Heart rate: ' + heartRate);
      });
    });
})
.catch(error => {
  console.log('Error: ' + error);
});
```

In this example, we first call the `requestDevice` method of the `navigator.bluetooth` object to discover nearby Bluetooth devices that support the `heart_rate` service. Once a device is found, we connect to its GATT server and retrieve the `heart_rate` service and `heart_rate_measurement` characteristic.

We then start listening for notifications from the `heart_rate_measurement` characteristic using the `startNotifications` method, and add an event listener to the characteristic to handle incoming data. When a notification is received, we extract the heart rate value from the characteristic data and log it to the console.

Browser Compatibility

The Web Bluetooth API is currently supported by major browsers on mobile platforms, including Google Chrome, Microsoft Edge, Opera, and Samsung Internet **[90]**. However, support is still in a partial state for the `Bluetooth` interface as far as desktop browsers are concerned. It is important to note that support for the API is still limited, and not all devices and platforms are supported. Additionally, due to the security risks associated with Bluetooth connections, there are strict security requirements for the API, and it is only accessible to websites that are served over HTTPS. Below are the support tables for the Bluetooth API **[90]**:

Web Bluetooth API: `Bluetooth`			
Browser	Version	Desktop Support	Mobile Support
Google Chrome	56+	Partial	Yes
Microsoft Edge	79+	Partial	Yes

Opera	43+	Partial	Yes
Mozilla Firefox	Not supported	No	No
Safari	Not supported	No	No

Table 3-11: Bluetooth browser support

Web Bluetooth API: `availabilitychanged`			
Browser	Version	Desktop Support	Mobile Support
Google Chrome	56+	Yes	No
Microsoft Edge	79+	Yes	No
Opera	43+	Yes	No
Mozilla Firefox	Not supported	No	No
Safari	Not supported	No	No

Table 3-12: availabilitychanged browser support

Web Bluetooth API: `getAvailability`			
Browser	Version	Desktop Support	Mobile Support
Google Chrome	78+	Yes	Yes
Microsoft Edge	79+	Yes	Yes
Opera	65+ | 56+	Yes	Yes
Mozilla Firefox	Not supported	No	No
Safari	Not supported	No	No

Table 3-13: getAvailability browser support

Web Bluetooth API: `getDevices`			
Browser	Version	Desktop Support	Mobile Support
Google Chrome	Not supported	No	No
Microsoft Edge	Not supported	No	No
Opera	Not supported	No	No
Mozilla Firefox	Not supported	No	No

Safari	Not supported	No	No

Table 3-14: getDevices browser support

Web Bluetooth API: `requestDevice`			
Browser	Version	Desktop Support	Mobile Support
Google Chrome	56+	Yes	Yes
Microsoft Edge	79+	Yes	Yes
Opera	43+	Yes	Yes
Mozilla Firefox	Not supported	No	No
Safari	Not supported	No	No

Table 3-15: requestDevice browser support

Usage Statistics

According to the State of JavaScript 2020 survey, only 6.8% of respondents reported using the Web Bluetooth API **[91]**.

According to caniuse.com Web Bluetooth API shows global usage among all browsers desktop and mobile of 43.46%. Notable exception is the `getDevices` interface that shows a percentage of 0% **[90]**.

Page load statistics from Chrome Platform Status show the following **[91, 92, 93]**:

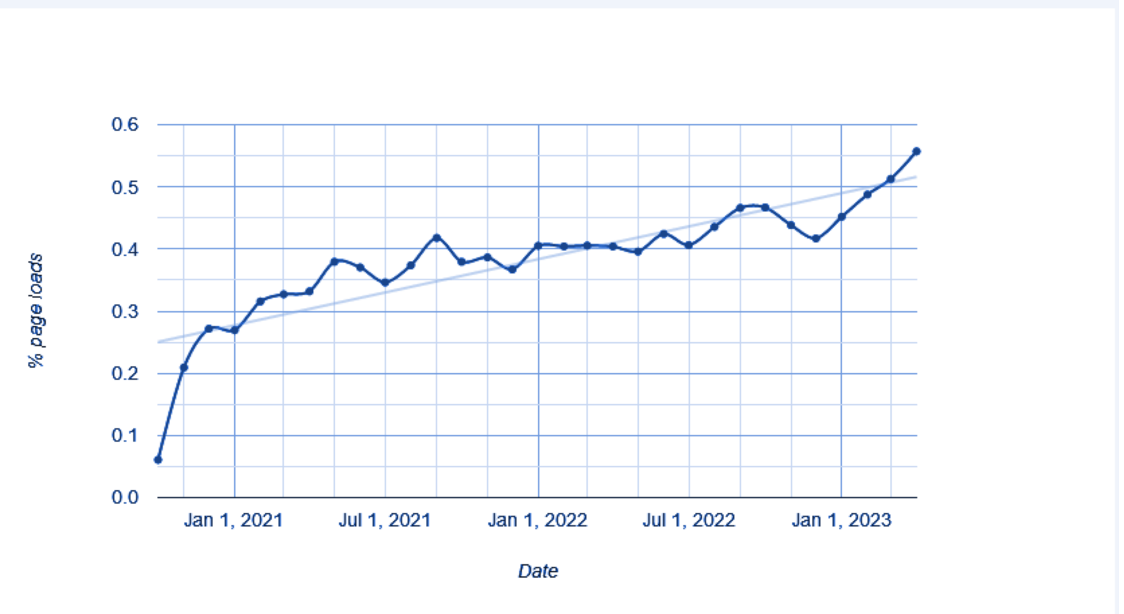

Figure 3-6: WebBluetoothGetAvailability percentage of page loads over time in Chrome.

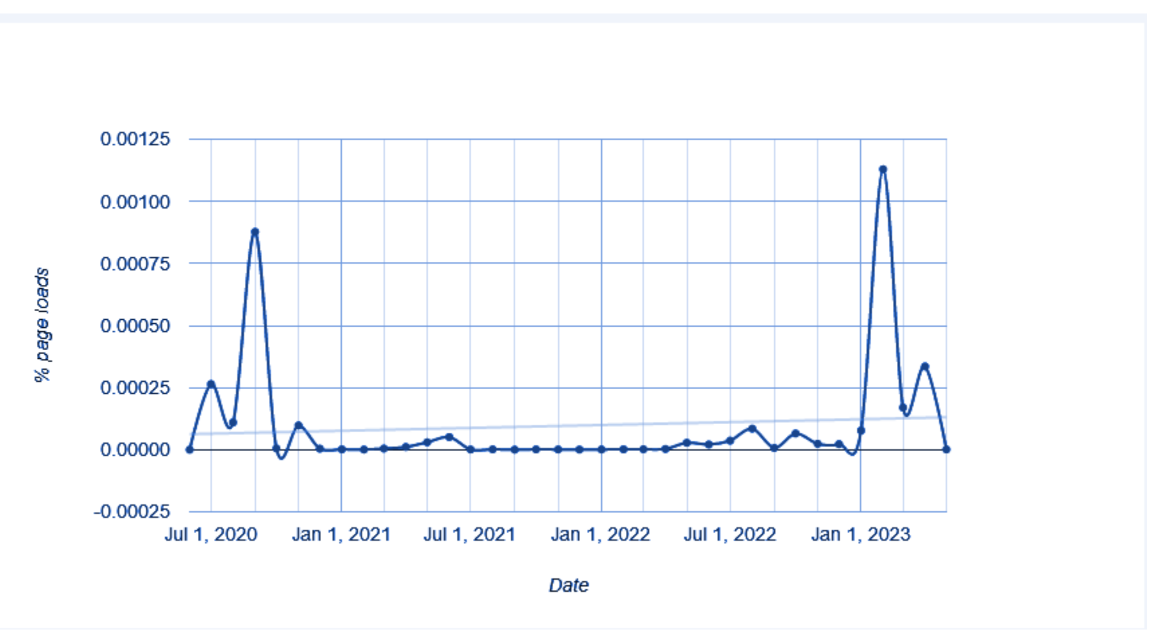

Figure 3-7: WebBluetoothGetDevices percentage of page loads over time in Chrome.

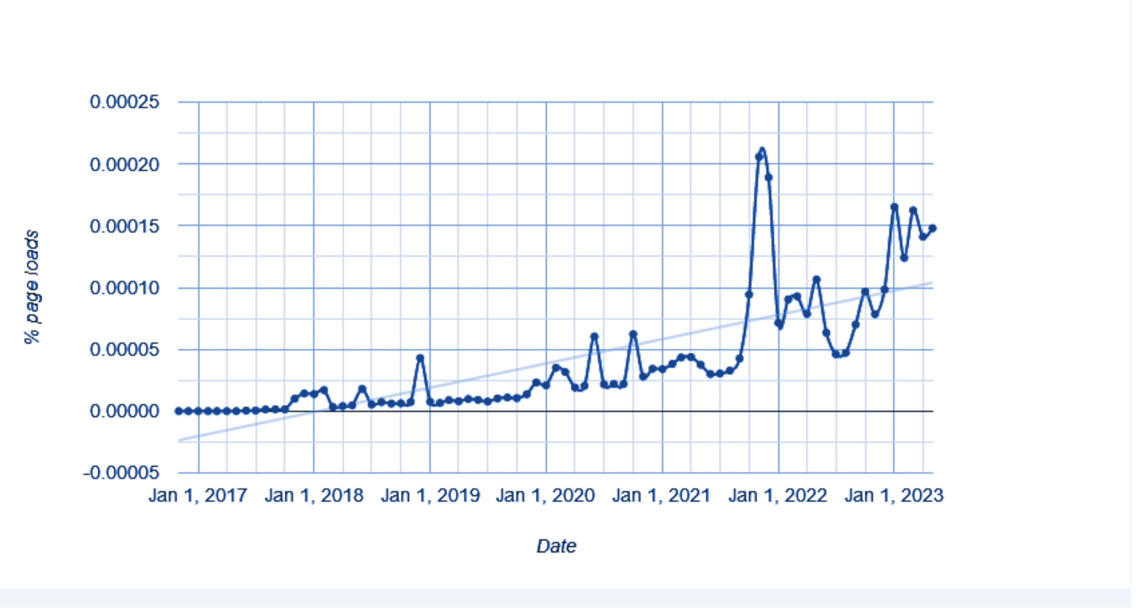

Figure 3-8: WebBluetoothRequestDevice percentage of page loads over time in Chrome.

Google Chrome page loads for interfaces that are supported, all tell the same story: Very low percentages of page loads across all features.

No data exist for sample URLs that use the API, in Chrome page loads **[91, 92, 93]**:

`WebBluetoothGetAvailability`, `WebBluetoothGetDevices`, `WebBluetoothRequestDevice`: Top five sample URLs
NO DATA AVAILABLE

Table 3-16: WebBluetoothGetAvailability, WebBluetoothGetDevices, WebBluetoothRequestDevice top five sample URLs

All available statistics suggest that the adoption of this API is still relatively low. The API has been available for several years now and has been steadily gaining adoption. As more and more devices become Bluetooth-enabled, the potential uses for this API will continue to grow. Additionally, as developers become more familiar with the capabilities of the Web Bluetooth API, we may see an increase in its usage in web applications.

3.1.6 Keyboard API

General Description

The Keyboard API is a web browser API that provides a way for web applications to access information about the physical keyboard attached to the user's device **[94]**. This API allows developers to capture and respond to keyboard events, such as key presses and releases, and to query the current state of the keyboard, such as which keys are currently pressed.

The Keyboard API uses two W3C Group Community Draft Reports: **Keyboard Lock** and **Keyboard Map** specifications **[95, 96]**

It works by exposing a set of JavaScript events that correspond to keyboard events, such as `keydown`, `keyup`, and `keypress`. These events provide information about the key that was pressed, such as its key code and whether it was a modifier key, such as the shift or control key.

In addition to keyboard events, the Keyboard API also provides a way for developers to query the current state of the keyboard using the `Keyboard` interface. This

interface also allows developers to determine which keys are currently pressed and whether specific modifier keys, such as the shift or control key, are also pressed.

One of the key benefits of the Keyboard API is that it provides a standardized way to access information about the physical keyboard across different devices and browsers. This can help developers to create cross-platform web applications that respond to keyboard input in a consistent and predictable manner.

The Keyboard Web API allows web developers to listen for and handle keyboard events that occur in a web page. Keyboard events include keydown, keyup, and keypress events. The Keyboard Web API provides a way to capture and respond to these events in a web page, allowing developers to create customized keyboard controls for their applications.

With the Keyboard Web API, web developers can listen for keyboard events from specific keys or key combinations, such as Ctrl+C or Shift+Enter, and then trigger specific actions based on those events. This can be useful for a variety of web applications, such as text editors, games, or productivity tools.

Overall, the Keyboard API is an important tool for web developers who need to create web applications that respond to keyboard input. By providing a standardized way to access information about the physical keyboard across different devices and browsers, this API can help developers to create cross-platform web applications that respond to keyboard input in a consistent and predictable manner.

Interfaces

The Keyboard Web API consists of three interfaces:

- `Keyboard`: Provides functions that retrieve keyboard layout maps and toggle capturing of key presses from the physical keyboard.
- `KeyboardLayoutMap`: A map-like object with functions for retrieving the string associated with specific physical keys.
- `navigator.keyboard` (Read only): Returns a `Keyboard` object which provides access to functions that retrieve keyboard layout maps and toggle capturing of key presses from the physical keyboard.

The `Keyboard` interface provides the following properties:

- `layoutMap`: Returns a `Map` object that maps each physical key to an array of strings, each of which represents a symbol that the key can produce. The first element of the array is the primary symbol, followed by any number of alternate symbols.
- `onkeyup`: An event handler property that gets called whenever a physical key on the keyboard is released. It takes an event object of type `KeyboardEvent`.
- `onkeydown`: An event handler property that gets called whenever a physical key on the keyboard is pressed down. It takes an event object of type `KeyboardEvent`.

The `KeyboardEvent` interface provides additional properties that describe the event that was generated, such as `keyCode`, `key`, and `code`. It also provides methods for manipulating the default action of the event, such as `preventDefault()` and `stopPropagation()`.

Security Requirements

The Keyboard Web API doesn't have specific security requirements, as it's considered a low-risk API **[94, 95, 96]**. However, as with any API that interacts with user input, it's important to consider the potential security risks associated with using it.

For example, if a web application uses the Keyboard API to capture user input, it could potentially capture sensitive information such as passwords or credit card numbers. Therefore, it's important for developers to implement appropriate security measures such as encryption and secure storage to protect user data.

Additionally, as with any API, it's important to ensure that the code implementing the Keyboard API is secure and free from vulnerabilities such as cross-site scripting (XSS) or injection attacks. Developers should follow best practices for web application security and keep up to date with any security advisories or patches for the Keyboard API.

Implementation Example

```
// Add an event listener for the 'keydown' event
window.addEventListener('keydown', function(event) {
  // Log the key code of the key that was pressed
  console.log(event.keyCode);

  // Check if the shift key is pressed
  if (event.shiftKey) {
      console.log('Shift key is pressed');
  }

  // Check if the control key is pressed
  if (event.ctrlKey) {
      console.log('Control key is pressed');
  }
});
```

In this example, we add an event listener for the `keydown` event to the `window` object. When a key is pressed, the event listener logs the key code of the pressed key to the console, and checks whether the shift or control key is also pressed using the `shiftKey` and `ctrlKey` properties of the `event` object.

Browser Compatibility

The Keyboard Web API is supported by many modern web browsers including Google Chrome, Microsoft Edge, and Opera. Support for specific methods or properties of the API varies slightly between browsers. Below are the compatibility tables that show support among various browsers **[94, 97]**:

Keyboard API: `api.Keyboard`			
Browser	Version	Desktop Support	Mobile Support
Google Chrome	68+	Yes	Yes
Microsoft Edge	79+	Yes	Yes
Opera	55+ | 48+	Yes	Yes
Mozilla Firefox	Not Supported	No	No
Safari	Not Supported	No	No

Table 3-17: api.Keyboard browser support

Keyboard API: `api.KeyboardLayoutMap`			
Browser	Version	Desktop Support	Mobile Support
Google Chrome	69+	Yes	Yes
Microsoft Edge	79+	Yes	Yes
Opera	56+ | 48+	Yes	Yes
Mozilla Firefox	Not Supported	No	No
Safari	Not Supported	No	No

Table 3-18: api.KeyboardLayoutMap browser support

Usage Statistics

Caniuse.com, as of April 2023, reports that the Keyboard Web API has a global usage percentage of 72.66% across all users **[97]**.

Chrome Platform Status reports that `api.KeyboardLayoutMap` has a high percentage of page loads in Chrome. Loads are recorded at 3% in the first months of 2023 **[98, 99, 100]**:

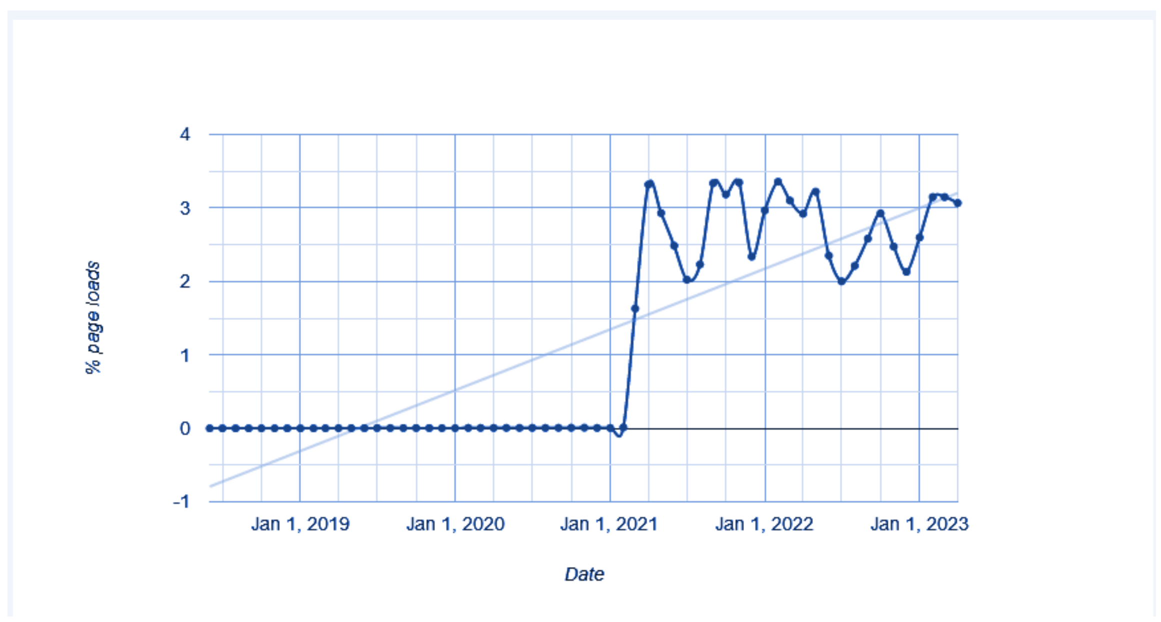

Figure 3-9: KeyboardApiGetLayoutMap percentage of page loads over time in Chrome.

The table below shows the top five sample URLs for the Keyboard API **[98]**:

`KeyboardApiGetLayoutMap`: Top five sample URLs
https://purchase.sea.com/
https://ironmountain-sso.prd.mykronos.com/
https://email.bol.uol.com.br/
https://purecosmetics.com/
https://dombarber.app/

Table 3-19: KeyboardApiGetLayoutMap top five sample URLs

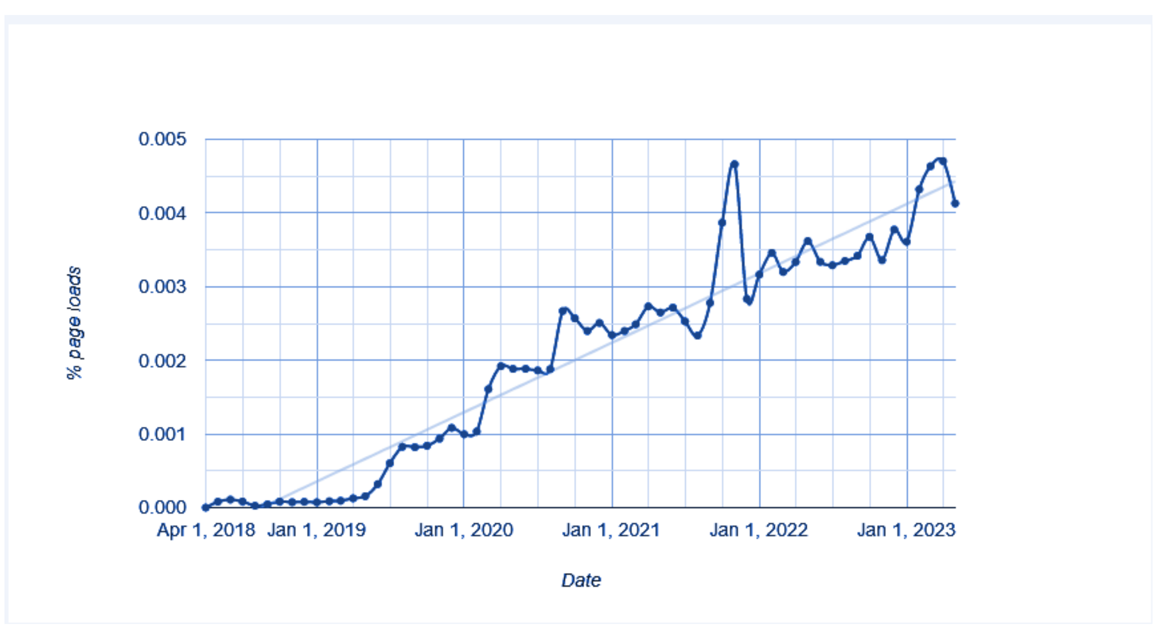

Figure 3-10: KeyboardApiLock percentage of page loads over time in Chrome.

`KeyboardApiLock`: Top five sample URLs
https://www.depedtrends.com/
https://www.greatbhajan.live/
https://www.omamore.com/
https://www.bellajamal.com/
https://www.sweetytangyspicy.com/

Table 3-20: KeyboardApiLock top five sample URLs

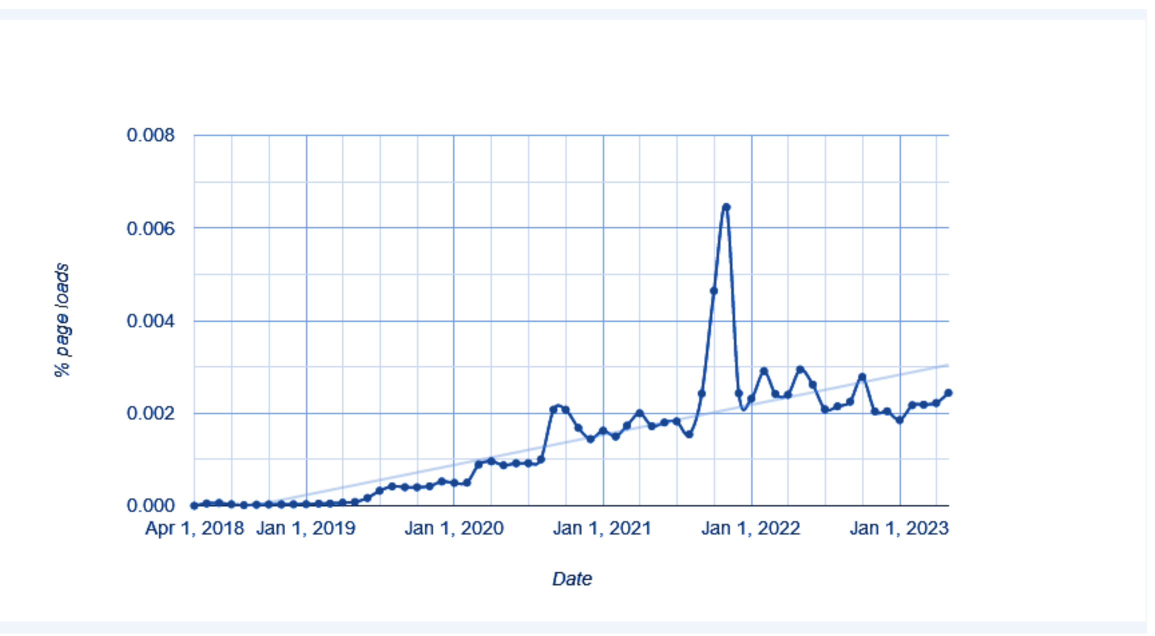

Figure 3-11: KeyboardApiUnlock percentage of page loads over time in Chrome.

`KeyboardApiUnlock`: Top five sample URLs
https://gxc.gg/
https://www.gxc.gg/
https://livesurf.ru/
https://metagallery.prugio.com/
NONE

Table 3-21: KeyboardApiUnlock top five sample URLs

3.1.7 Presentation API

General Description

The Presentation API is a web browser API that enables web applications to display content on external screens or projectors. This API provides a standard interface for discovering available presentation displays and presenting content on them **[101, 102]**.

The Presentation API works by exposing a `navigator.presentation` object in JavaScript, which provides a set of methods for discovering and controlling presentation displays. Developers can use this API to find available presentation displays and initiate a presentation session by passing a URL or a DOM element to the `navigator.presentation.startSession()` method. Once the session is started, the API provides methods for controlling the presentation, such as navigating to a different URL or displaying different content.

One of the key benefits of the Presentation API is that it provides a standardized way for web applications to display content on external screens or projectors. This can be useful in a variety of scenarios, such as for presentations or digital signage. By using the Presentation API, developers can create web applications that can seamlessly integrate with external displays and provide a better user experience.

Overall, the Presentation API is an important tool for web developers who need to display content on external screens or projectors. By providing a standardized way to discover available presentation displays and present content on them, this API can help developers to create web applications that seamlessly integrate with external displays and provide a better user experience.

Interfaces

The Presentation API is a web browser API that allows web applications to access external displays or presentation devices and to control their behavior. The API provides interfaces that enable web applications to initiate a presentation session, access and manage the available displays, and control the presentation behavior, such as navigating between slides and displaying content.

The Presentation API includes the following interfaces:

1. `NavigatorPresentation` interface: This interface provides access to the presentation displays and sessions. It allows the web application to start a presentation session, query the available displays, and listen for changes to the display list.
2. `Presentation` interface: This interface represents a presentation session. It provides methods for controlling the presentation, such as displaying content on the external display, navigating between slides, and ending the session.
3. `PresentationConnection` interface: This interface represents a communication channel between the web application and the presentation display. It provides methods for sending and receiving messages and events, such as slide change requests and connection status changes.
4. `PresentationReceiver` interface: This interface represents a receiver that can handle incoming presentation requests from other devices. It allows web applications to receive and handle presentation requests from other devices, such as mobile phones or tablets.
5. `PresentationRequest` interface: This interface Initiates or reconnects to a presentation made by a controlling browsing context.

Security Requirements

The Presentation Web API, like any other web API, has security requirements to ensure that it is not misused and that user data is protected. Some security requirements for the Presentation Web API are:

1. Origin checks: The Presentation Web API only works on sites that have the same origin as the presenting page. This means that only web pages from the same domain and protocol as the presenting page can access the API.
2. User permission: The Presentation Web API requires user permission before it can be used. This ensures that users are aware that a website is trying to access their presentation displays, and can choose to allow or deny the request.
3. HTTPS: The Presentation Web API requires HTTPS to work. This ensures that the communication between the presenting page and the receiver page is encrypted, protecting the user's data from eavesdropping and tampering.

4. User data protection: The Presentation Web API should not be used to collect or transmit user data without their consent. Websites should implement appropriate measures to ensure that user data is protected.
5. CORS: The Cross-Origin Resource Sharing (CORS) policy is enforced for the Presentation Web API. This means that the API can only be used by websites that have been explicitly allowed by the server through CORS headers.

By following these security requirements, the Presentation Web API can be used securely in web applications without compromising user data or system security.

Implementation Example

```
// Check if the Presentation API is available
if ('presentation' in navigator) {
  // Find available presentation displays
  navigator.presentation.defaultRequest = {
      available: true
  };

navigator.presentation.requestSession().then(function(session)
{
      // Get the available presentation displays
      var displays = session.getAvailableDisplays();

      // Choose the first display
      var display = displays[0];

      // Load the content to display
      var url = 'https://example.com/my-presentation';
      display.postMessage({action: 'load', url: url});
  });
}
```

In this example, we first check whether the Presentation API is available in the `navigator` object. If it is available, we set the `defaultRequest` object to indicate that we are looking for available presentation displays. We then call the `requestSession()` method to initiate a presentation session. Once the session is started, we get the available displays using the `getAvailableDisplays()` method and choose the first display.

Finally, we load the content to display by sending a message to the display using the `postMessage()` method.

Browser Compatibility

The Presentation API is supported by several web browsers, including Google Chrome, MIcrosoft Edge, Opera and Samsung Internet **[103, 101]**.

Presentation API: `Presentation`, `defaultRequest`			
Browser	Version	Desktop Support	Mobile Support
Google Chrome	47+	Yes	Yes
Microsoft Edge	79+	Yes	Yes
Opera	34+	Yes	Yes
Mozilla Firefox	Not supported	No	No
Safari	Not supported	No	No

Table 3-22: Presentation, defaultRequest browser support

Presentation API: `receiver`			
Browser	Version	Desktop Support	Mobile Support
Google Chrome	59+	Yes	Yes
Microsoft Edge	79+	Yes	Yes
Opera	46+ | 43+	Yes	Yes
Mozilla Firefox	Not supported	No	No
Safari	Not supported	No	No

Table 3-23: receiver browser support

Usage Statistics

According to caniuse.com, the Presentation API shows the following global usage percentages **[103]**:

- Presentation API: 73.17%
- PresentationRequest: 73.17%
- PresentationReceiver: 72.9%
- PresentationConnection: 73.17%

According to Chrome Platform Status, the Presentation API has had a steady percentage of 5-6% in page loads during the last few years **[104, 105]**. Below are graphs that show page loads, as well as sample URLs **[104, 105]**.

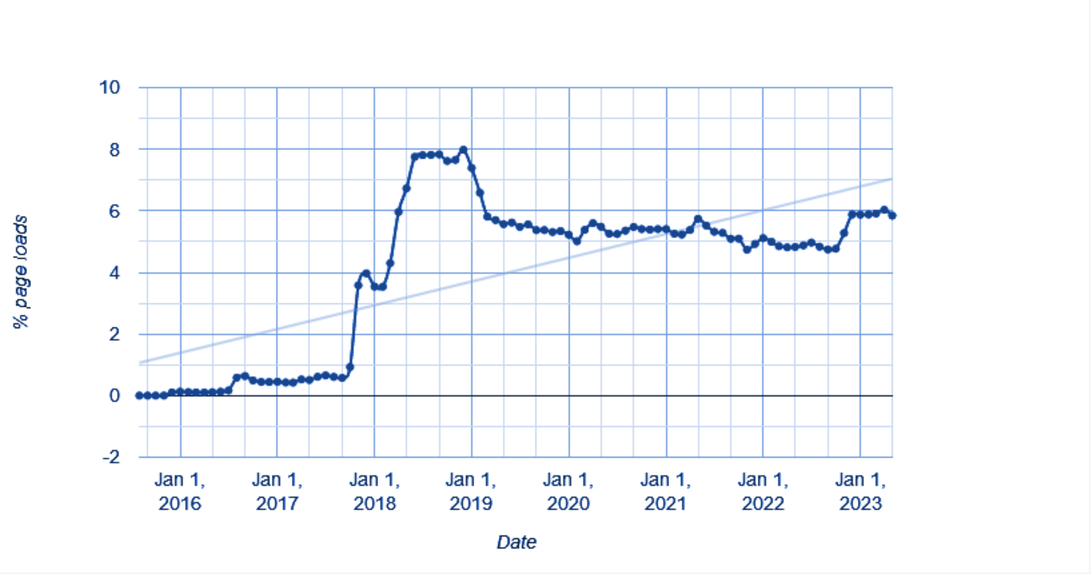

Figure 3-12: PresentationDefaultRequest percentage of page loads over time in Chrome.

PresentationDefaultRequest: Top five sample URLs
https://orientation.conestogac.on.ca/
https://www.paramedicineboard.gov.au/
https://www.zagori.gov.gr/
https://anuer.org/
https://grandes-maurieres.chiens-de-france.com/

Table 3-24: PresentationDefaultRequest top five sample URLs

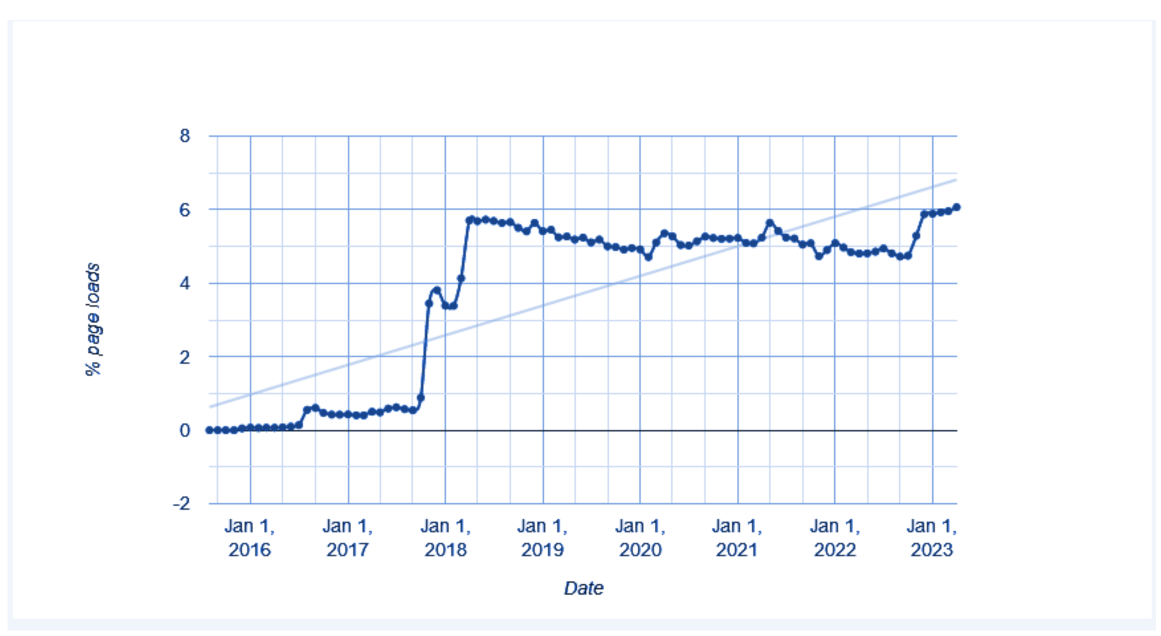

Figure 3-13: PresentationAvailabilityChangeEventListener percentage of page loads over time in Chrome.

`PresentationAvailabilityChangeEventListener`: Top five sample URLs
https://www.shredrack.com/
https://netspeed.com.br/
https://www.neurosystem.it/
https://www.creciendo.com/
https://survivalpandas.blogspot.com/

Table 3-25: PresentationAvailabilityChangeEventListener top five sample URLs

3.1.8 Web NFC API

General Description

The Web NFC API allows web applications to read and write Near Field Communication (NFC) tags. NFC tags are small wireless devices that can be embedded in everyday objects, such as posters, business cards, and products **[106, 107]**. It provides web applications with the ability to interact with these tags, opening up new possibilities for web-based services and applications. The API has the status of Draft Community Group Report on W3C **[107]**.

The Web NFC API is designed to be simple and easy to use. It provides a set of methods and events that allow web applications to detect and communicate with NFC tags. The API can be used with a wide range of NFC tags, including NDEF (NFC Data Exchange Format) tags, which are the most common type of NFC tag **[106]**.

Interfaces

The Web NFC API defines several interfaces that allow web applications to interact with NFC devices:

1. `NDEFReader`: This interface represents an NFC reader device that can read NFC data in the NDEF format. It provides methods to check if an NFC device is available, start and stop reading, and event handlers to handle the reading process.
2. `NDEFMessage`: This interface represents an NDEF message, which is a standard format for storing and exchanging data on NFC tags or devices. It provides methods to get and set the NDEF records that make up the message.
3. `NDEFRecord`: This interface represents an NDEF record, which is a unit of data in an NDEF message. It provides methods to get and set the data, type, and other metadata associated with the record.
4. `NDEFWriteOptions`: This interface represents options for writing NDEF messages to an NFC tag. It provides properties to specify the message to write, the size of the tag, and the write behavior.
5. `NDEFWriter`: This interface represents an NFC writer device that can write NDEF messages to an NFC tag. It provides methods to check if an NFC device is

available, write a message to a tag, and event handlers to handle the writing process.

6. `NFC`: This interface represents the global scope for the WebNFC API. It provides methods to get an `NDEFReader` or `NDEFWriter` device, check if the API is supported, and event handlers to handle changes in the availability of NFC devices.

Together, these interfaces provide a comprehensive set of tools for web developers to work with NFC devices in their applications.

Security Requirements

The Web NFC API involves sensitive user information, such as the user's location and the data transmitted between the device and the NFC reader. Therefore, it is crucial to ensure that the Web NFC API is secure and provides users with a safe and trustworthy experience **[107]**. Here are some of the security requirements for the Web NFC API:

1. User Consent: The Web NFC API must not be used without the user's consent. Whenever a website or application intends to use the API, it should obtain the user's permission first.
2. Data Privacy: The API should ensure that the data transmitted between the device and the NFC reader is secure and cannot be intercepted by unauthorized parties.
3. Origin Validation: The Web NFC API should only be accessible from secure origins (HTTPS). This will prevent attackers from exploiting vulnerabilities in the API to execute malicious code or steal sensitive data.
4. Permissions Model: The Web NFC API should have a robust permissions model that allows users to manage their data and revoke permissions at any time.
5. Security Audits: The Web NFC API should be regularly audited for vulnerabilities and weaknesses, and any issues should be addressed immediately.

Implementation Example

Below is an example of how to use the Web NFC API in a web application to read an NFC tag:

```
// Check if the Web NFC API is supported
```

```
if ('NDEFReader' in window) {

  // Create a new NFC reader
  const nfcReader = new NDEFReader();

  // Add an event listener for when a tag is detected
  nfcReader.addEventListener("reading", ({ message,
serialNumber }) => {

      // Read the data from the tag
      const data = message.records[0].data;

      // Do something with the data
      console.log(`NFC tag detected: ${data}`);

  });

  // Start the NFC reader
  nfcReader.scan({ keepSessionOpen: true })
      .then(() => console.log("Ready to read NFC tags..."))
      .catch((error) => console.log(`Error: ${error}`));

} else {
  console.log("Web NFC API not supported.");
}
```

In this example, the code first checks if the Web NFC API is supported in the browser. If it is, it creates a new NDEFReader object and adds an event listener for when an NFC tag is detected. When a tag is detected, the code reads the data from the tag and logs it to the console. Finally, the code starts the NFC reader and logs a message to indicate that it's ready to read NFC tags.

Note that this example only works on Android devices with Chrome or Opera web browsers that support the Web NFC API. Other devices and web browsers do not support the API and will require different code to read and write NFC tags.

Browser Compatibility

The Web NFC API is currently supported on Chrome for Android, Opera for Android, and Samsung Internet. It is not supported on desktop browsers **[106, 110]**.

Web NFC API: NDEFReader, NDEFReader(), makeReadOnly, reading, readingerror, scan, write, *Secure context required*			
Browser	Version	Desktop Support	Mobile Support
Google Chrome	89+	No	Yes
Microsoft Edge	Not supported	No	No
Opera	63+	No	Yes
Mozilla Firefox	Not supported	No	No
Safari	Not supported	No	No

Table 3-26: NDEFReader, NDEFReader(), makeReadOnly, reading, readingerror, scan, write browser support

Given this opportunity, it is important to note that Safari is strongly opposed to the specification of the API, and will in fact not implement it in their browser, mainly due to serious security concerns. They state in a written response on the WebKit mailing lists that: "...we think exposing direct hardware access to the web is a bad idea and compromises the device-independence of the web platform **[108]**.

Similarly, the Mozilla Foundation has a strong disagreement regarding the API and it is not likely that it will be implemented in its current state. Mozilla's stance is also negative due to security concerns **[109]**.

Usage Statistics

According to the State of JS 2020 survey, which at the time polled over 23,000 web developers, the web NFC API had a usage rate of 5.5%. This suggests that the API is not widely used by developers at the moment.

Statistics from caniuse.com show the Web NFC API at 40.89% global usage among all users, while NDEF features are at 43.27% **[110]**.

On Google's Chrome Platform Status, page loads that use the WebNFC API are on the margin of error, barely above 0 on certain dates. Highest recorded page loads is in January 2022 with 0.0008% **[111, 112]**.

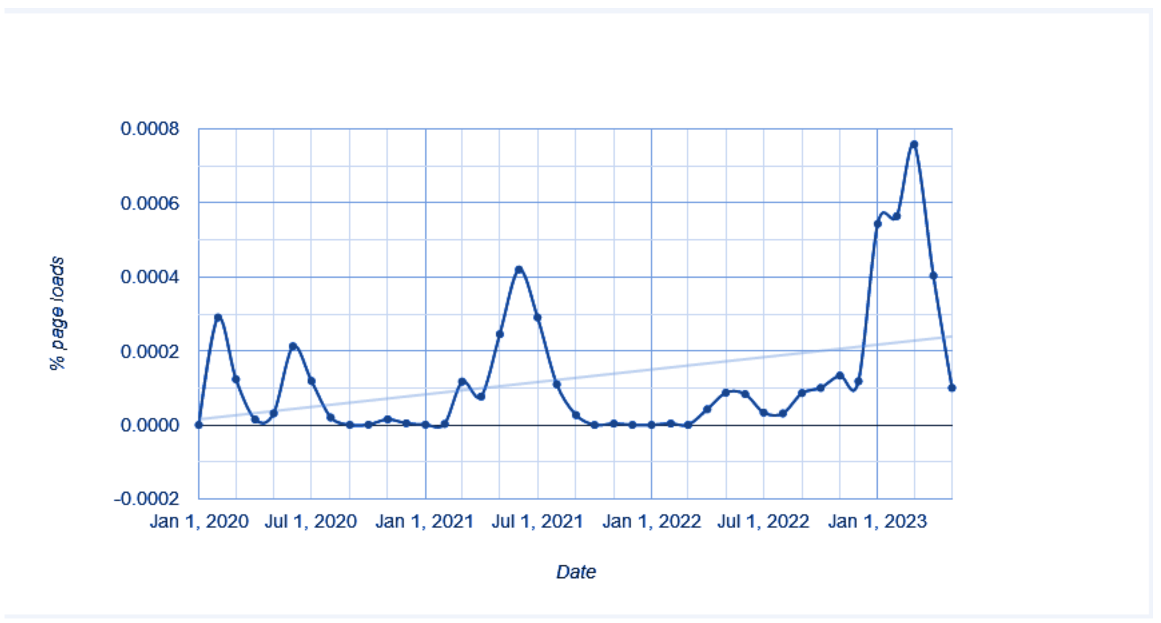

Figure 3-14: WebNfcAPI percentage of page loads over time in Chrome.

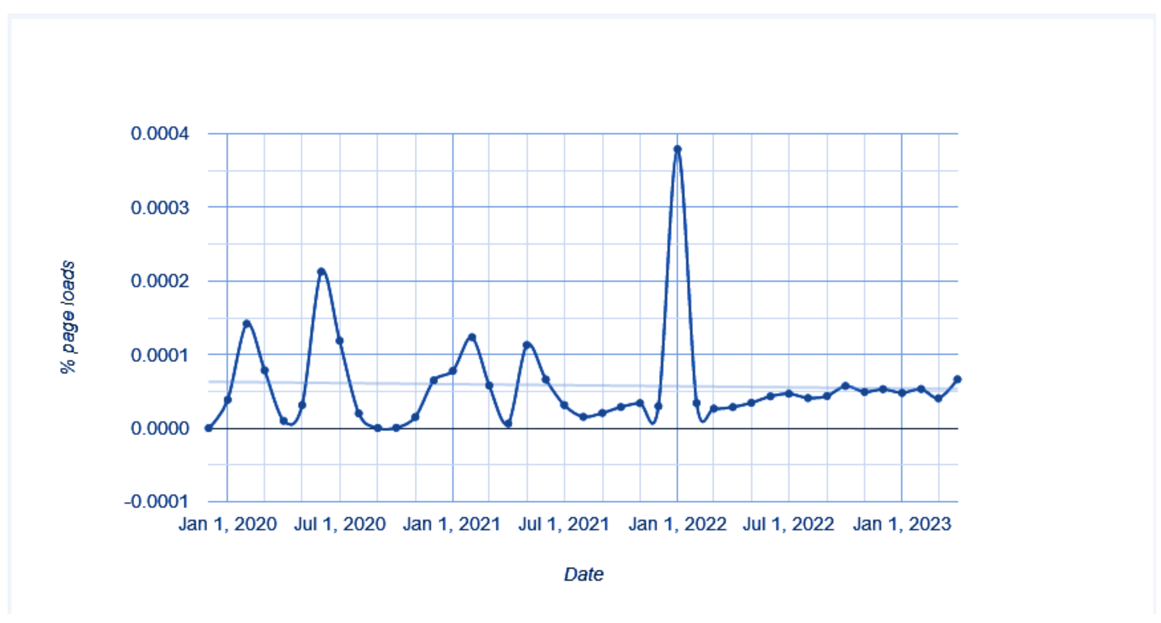

Figure 3-15: WebNfcNdefReaderScan percentage of page loads over time in Chrome.

`WebNfcAPI`, `WebNfcNdefReaderScan`: Top five sample URLs
NO DATA AVAILABLE

Table 3-27: WebNfcAPI, WebNfcNdefReaderScan top five sample URLs

As a final note, the usage of the Web NFC API may be higher among specific types of websites or web applications that require proximity-based interactions, such as mobile payment applications or event check-in systems. It is also possible that usage of the Web NFC API will increase as more devices with NFC capabilities become available and more web developers become familiar with its capabilities.

3.1.9 WebGPU API

General Description

The WebGPU API is a W3C Working Draft that enables web developers to use the underlying system's GPU (Graphics Processing Unit) to carry out high-performance computations and draw complex images that can be rendered in the browser **[114]**.

WebGPU is the successor to WebGL, providing better compatibility with modern GPUs, support for general-purpose GPU computations, faster operations, and access to more advanced GPU features.

The Web GPU API is a browser API that provides low-level access to graphics and computing capabilities of the device's GPU (Graphics Processing Unit) in a way that is more efficient than the existing WebGL API **[113]**. It is designed to enable high-performance graphics and compute operations, such as rendering complex 3D scenes, performing machine learning tasks, and accelerating scientific simulations **[113, 114]**.

Web GPU is built on top of modern graphics APIs such as DirectX 12 and Vulkan, and it exposes a similar programming model to those APIs. This allows developers to write highly performant code that can run across multiple platforms and devices, including desktop and mobile devices.

The Web GPU API is designed to be simple, safe, and fast, and it supports features such as asynchronous execution, explicit synchronization, and fine-grained memory management **[115]**. It is still a relatively new API, and it is currently under development by the WebGPU working group, which includes representatives from major browser vendors and GPU manufacturers.

Overall, the Web GPU API promises to bring high-performance graphics and computing capabilities to the web platform, enabling new types of web applications and experiences that were previously not possible.

There are several layers of abstraction between a device GPU and a web browser running the WebGPU API as can be seen in the following diagram **[117]**:

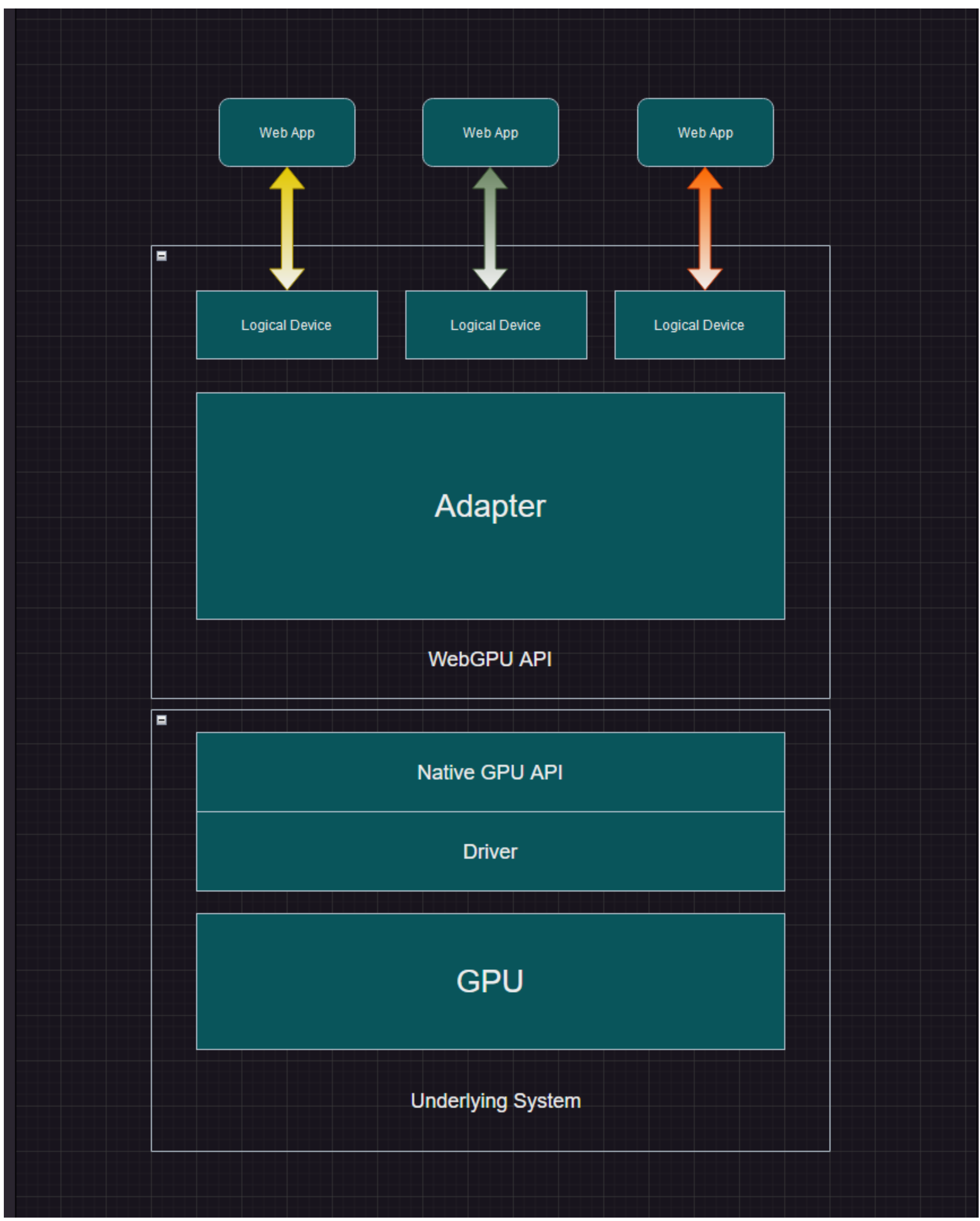

Figure 3- 16: Web GPU API in the GPU-Web Application Stack

Physical devices have GPUs. Most devices only have one GPU, but some have more than one. Different GPU types are available:

- Integrated GPUs, which live on the same board as the CPU and share its memory.
- Discrete GPUs, which live on their own board, separate from the CPU.
- Software "GPUs", implemented on the CPU.

A native GPU API, which is part of the OS (e.g. Metal on macOS), is a programming interface allowing native applications to use the capabilities of the GPU. API instructions are sent to the GPU (and responses received) via a driver. It is possible for a system to have multiple native OS APIs and drivers available to communicate with the GPU, although the above diagram assumes a device with only one native API/driver.

A browser's WebGPU implementation handles communicating with the GPU via a native GPU API driver **[115]**. A WebGPU adapter effectively represents a physical GPU and driver available on the underlying system, in your code.

A logical device is an abstraction via which a single web app can access GPU capabilities in a compartmentalized way. Logical devices are required to provide multiplexing capabilities. A physical device's GPU is used by many applications and processes concurrently, including potentially many web apps. Each web app needs to be able to access WebGPU in isolation for security and logic reasons.

Interfaces

The Web GPU API includes several interfaces that provide access to the GPU's graphics and compute capabilities. The most important of these interfaces are:

1. `GPUDevice`: Represents the device's GPU and provides methods for creating and managing GPU resources such as buffers, textures, and pipelines.
2. `GPURenderPassEncoder`: Provides methods for encoding commands to render a frame, including setting the render target, setting the vertex and index buffers, and drawing primitives.
3. `GPUComputePassEncoder`: Provides methods for encoding commands to execute a compute shader, including setting the compute pipeline, setting shader inputs and outputs, and dispatching the shader.

4. `GPUBuffer`: Represents a block of memory on the GPU, and provides methods for reading and writing data to and from the buffer.
5. `GPUTexture`: Represents an image or texture on the GPU, and provides methods for creating and manipulating the texture.

These interfaces, along with others provided by the Web GPU API, enable developers to create and execute highly performant graphics and compute operations on the GPU. They offer a low-level and efficient way to access the GPU, and provide fine-grained control over how graphics and compute tasks are executed.

Security Requirements

The Web GPU API has several security requirements that must be met in order to protect the user's system and data **[114]**. These requirements include:

1. Origin and Permissions: As with other web APIs, the Web GPU API requires that the user grant permission for the application to access the GPU. Additionally, the GPU resources created by the application are bound to the origin of the application, and cannot be accessed by other origins.
2. Memory Safety: The Web GPU API is designed to ensure memory safety, which is critical for preventing security vulnerabilities such as buffer overflows and memory leaks. The API includes features such as automatic memory management and bounds checking to help ensure the safety of the GPU resources.
3. Driver Isolation: The Web GPU API requires that the browser isolate the GPU driver and prevent it from accessing other system resources. This helps prevent malicious drivers from compromising the user's system.
4. Secure Transport: The Web GPU API requires that all GPU data be transmitted securely, using secure transport protocols such as HTTPS. This helps protect the user's data from interception and tampering.
5. GPU Feature Isolation: The Web GPU API requires that GPU features be isolated from each other, to prevent one feature from accessing or modifying the data of another feature. This helps prevent data leakage and other security vulnerabilities.

By meeting these security requirements, the Web GPU API provides a secure and reliable way for web applications to access the GPU, enabling high-performance graphics and compute operations while protecting the user's system and data.

Implementation Example

```
// Request permission to access the GPU
const adapter = await navigator.gpu.requestAdapter();

// Create a GPU device using the selected adapter
const device = await adapter.requestDevice();

// Create a swap chain for presenting the output
const swapChain = device.createSwapChain(canvas, {
devicePixelRatio });

// Define the vertex and fragment shaders
const vertexShaderCode = `
      // Define vertex shader code here
`;
const fragmentShaderCode = `
      // Define fragment shader code here
`;

// Compile the vertex and fragment shaders into GPU code
const vertexShaderModule = device.createShaderModule({ code:
vertexShaderCode });
const fragmentShaderModule = device.createShaderModule({ code:
fragmentShaderCode });

// Define the vertices and indices for the 3D model
const vertices = [...];
const indices = [...];

// Create a buffer to store the vertex data
const vertexBuffer = device.createBuffer({
      size: vertices.byteLength,
      usage: GPUBufferUsage.VERTEX,
      mappedAtCreation: true,
});
new Float32Array(vertexBuffer.getMappedRange()).set(vertices);
vertexBuffer.unmap();

// Create a buffer to store the index data
const indexBuffer = device.createBuffer({
      size: indices.byteLength,
```

```
      usage: GPUBufferUsage.INDEX,
      mappedAtCreation: true,
});
new Uint16Array(indexBuffer.getMappedRange()).set(indices);
indexBuffer.unmap();

// Create a render pipeline for rendering the 3D model
const renderPipeline = device.createRenderPipeline({
      vertex: {
      module: vertexShaderModule,
      entryPoint: 'main',
      buffers: [
            {
                  arrayStride: 3 * 4, // Size of each vertex (3
floats)
                  attributes: [
                  {
                        shaderLocation: 0, // Corresponds to
'a_position' in vertex shader
                        offset: 0,
                        format: 'float32x3',
                  },
                  ],
            },
      ],
      },
      fragment: {
      module: fragmentShaderModule,
      entryPoint: 'main',
      targets: [
            {
                  format: swapChain.format,
            },
      ],
      },
      primitive: {
      topology: 'triangle-list',
      },
});

// Draw the 3D model using the render pipeline
const commandEncoder = device.createCommandEncoder();
```

```
const textureView = swapChain.getCurrentTexture().createView();
const renderPassDescriptor = {
      colorAttachments: [
      {
            view: textureView,
            loadValue: { r: 0, g: 0, b: 0, a: 1 },
            storeOp: 'store',
      },
      ],
};
const passEncoder =
commandEncoder.beginRenderPass(renderPassDescriptor);
passEncoder.setPipeline(renderPipeline);
passEncoder.setVertexBuffer(0, vertexBuffer);
passEncoder.setIndexBuffer(indexBuffer, 'uint16');
passEncoder.drawIndexed(indices.length);
passEncoder.endPass();
device.queue.submit([commandEncoder.finish()]);
```

The above example creates a simple 3D model and renders it using the Web GPU API. The API is used to create a device, swap chain, shaders, buffers, and a render pipeline, which are then used to draw the model onto a canvas element. This example shows how the Web GPU API can be used to perform high-performance graphics operations in a web browser.

Browser Compatibility

The WebGPU API is a relatively new API and is not yet fully supported by all major web browsers. Below is a summary of its current browser compatibility **[113, 118]**:

- Google Chrome: Chrome has been shipping WebGPU since version 94, but it is still considered an experimental feature behind a flag.
- Mozilla Firefox: Firefox currently does not support WebGPU, but the browser has been working on implementing the API, and has only partial support in Firefox Nightly Build.
- Apple Safari: Safari also does not currently support WebGPU, but like Firefox, they have expressed interest in adding support in the future.

- Microsoft Edge: Microsoft Edge has experimental support for WebGPU behind a flag in the Canary and Dev channels.

It's worth noting that WebGPU is still a work in progress, and the level of support may change as the API continues to evolve. Additionally, even if a browser does not support WebGPU, there may be alternative APIs or fallbacks that can be used to achieve similar functionality.

WebGPU API: GPU, getPreferredCanvasFormat, requestAdapter			
Browser	Version	Desktop Support	Mobile Support
Google Chrome	113	In Development	No
Microsoft Edge	113	In Development	No
Opera	Not supported	No	No
Mozilla Firefox	NIghtly	Partial	No
Safari	Not supported	No	No

Table 3-28: GPU, getPreferredCanvasFormat, requestAdapter browser support

Usage Statistics

As of May 2023, the WebGPU API is still relatively new and not yet widely adopted. It was first introduced in Chrome 94 in September 2021 and is currently supported by Chrome, Edge and Firefox, but only in development builds **[113, 118]**.

According to the statistics from caniuse.com, as of May 2023, the WebGPU API has a global browser support of approximately 55% **[118]**. This means that 55% of all web users have a browser that supports this API. However, it is important to note that this does not mean that all of these users have the API enabled or that web developers are actively using it. According to usage statistics from the same source, the usage of WebGPU globally is exceptionally low, measuring only 0.02% **[118]**.

The above statistic appears to be accurate when compared with the statistics provided by Chrome Platform Status page loads. The website shows usage just barely

touching 0.01%, which is even lower than the usage statistic provided by caniuse.com. Below is the chart for WebGPU page loads in Chrome **[119, 120]**:

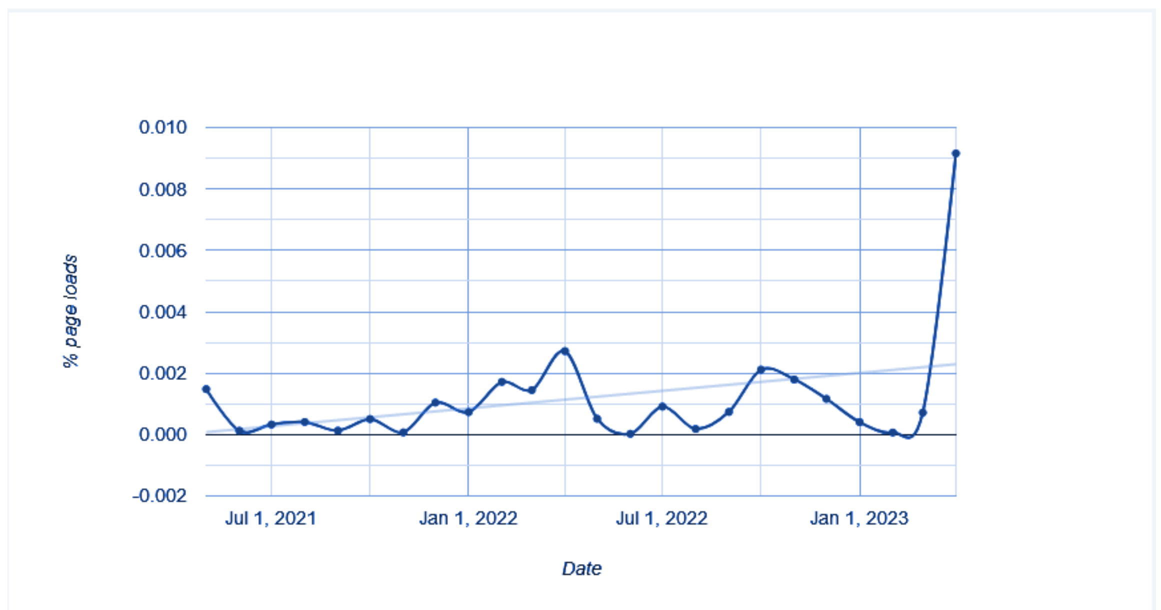

Figure 3-17: WebGPU percentage of page loads over time in Chrome.

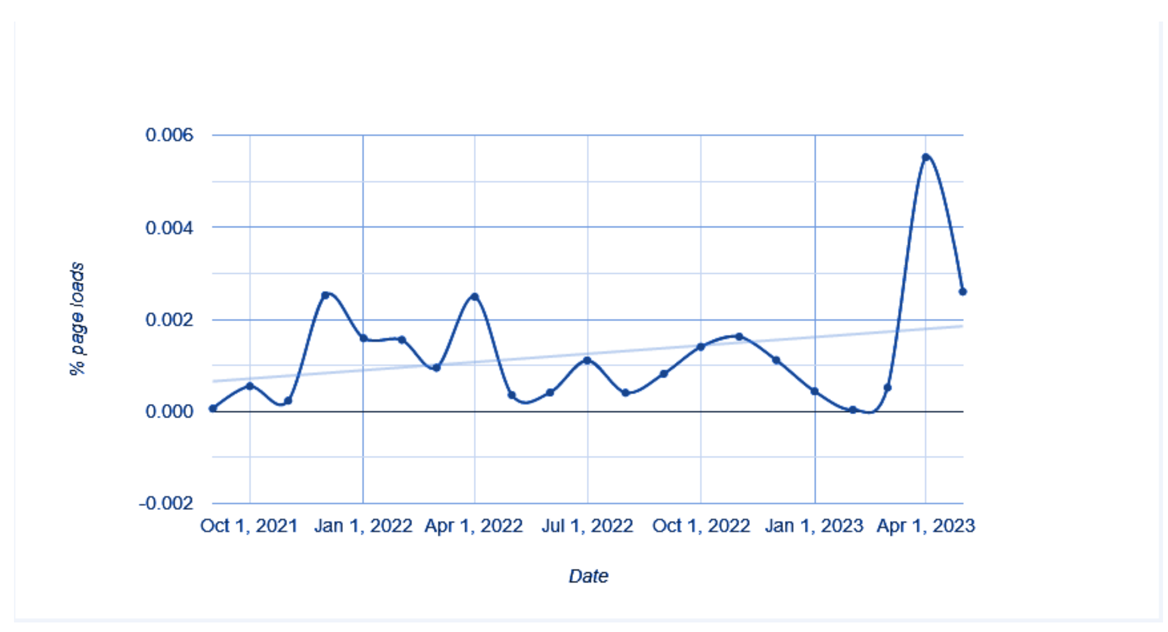

Figure 3-18: HTMLCanvasElement_WebGPU: Percentage of page loads over time in Chrome.

WebGPU, HTMLCanvasElement_WebGPU: Top five sample URLs
NO DATA AVAILABLE

Table 3-29: WebGPU, HTMLCanvasElement_WebGPU top five sample URLs

The WebGPU API is still in development and is likely to see further improvements and updates in the future. As it becomes more mature and gains wider support, it is expected that its adoption will increase as well.

3.1.10 WebHID API

General Description

The WebHID API is a web browser API that allows web applications to access human interface devices (HID), such as keyboards, mice, gamepads, and other input devices, directly from a web page without the need for a native application or driver **[121, 122]**.

With the WebHID API, web developers can create web applications that can interact with HID devices and provide users with a more natural and intuitive input experience. For example, a web-based game could use the API to support gamepads, or a web-based music application could use it to enable users to control playback with a physical media controller **[123]**.

The WebHID API is currently a draft specification and is not yet fully supported by all web browsers. It is being developed by the W3C Web Platform Incubator Community Group as part of the wider effort to expand the capabilities of web applications and improve their performance and user experience **[122]**.

A Human Interface Device (HID) is a type of device that takes input from or provides output to humans. It also refers to the HID protocol, a standard for bi-directional communication between a host and a device that is designed to simplify the installation procedure **[123]**. The HID protocol was originally developed for USB devices but has since been implemented over many other protocols, including Bluetooth.

Interfaces

The WebHID API provides several interfaces for communicating with human interface devices (HID) connected to a user's computer. These interfaces include:

1. `HID`: This interface represents a single HID device and provides methods for opening and closing the device, reading input reports, and sending output reports.
2. `HIDCollection`: This interface represents a collection of HID devices and provides methods for getting a list of devices in the collection, opening and closing devices, and registering for connection and disconnection events.
3. `HIDConnectionEvent`: This interface represents a connection or disconnection event for an HID device.
4. `HIDDevice`: This interface extends the HID interface and provides additional methods for getting information about the device, including its product ID, vendor ID, and serial number.
5. `HIDInputReportEvent`: This interface represents an input report event for an HID device and provides methods for getting the input report data and the device that generated the report.
6. `HIDOutputReport`: This interface represents an output report for an HID device and provides methods for setting the report data and sending the report to the device.

Security Requirements

The WebHID API follows the same security requirements as other web APIs to ensure that user privacy and security are not compromised. The security requirements of the WebHID API are:

1. Origin checks: The WebHID API requires a secure context, which means that it can only be accessed from HTTPS websites. The API also checks the origin of the website before granting access to a HID device.
2. User consent: The user must grant explicit permission for the website to access the HID device. The browser will display a prompt asking for permission to access the device, and the user must grant access for the API to work.

3. User notifications: The user must be notified when a website tries to access a HID device. This is to prevent malicious websites from accessing devices without the user's knowledge.
4. Limited access: The WebHID API only provides access to a limited set of features of the HID device, such as reading and writing reports. It does not allow direct access to the device's firmware or low-level settings that could compromise the security of the device or the user's system.

By following these security requirements, the WebHID API ensures that user privacy and security are maintained while still allowing websites to access HID devices.

Implementation Example

```
// Request access to a device with a given vendor ID and
product ID
navigator.hid.requestDevice({vendorId: 0x1234, productId:
0x5678})
  .then(device => {
     console.log(`Device connected: ${device.productName}`);
     // Open the device for input/output
     return device.open();
  })
  .then(device => {
     // Send a report to the device
     const reportId = 0x01;
     const data = new Uint8Array([0x01, 0x02, 0x03]);
     return device.sendReport(reportId, data);
  })
  .then(() => {
     console.log("Report sent successfully.");
  })
  .catch(error => {
     console.error(`Failed to send report: ${error}`);
  });
```

In this example, we request access to a device with a specific vendor ID and product ID using the `navigator.hid.requestDevice()` method. Once a device is connected, we open it for input/output using the `device.open()` method. We then send

a report to the device using the `device.sendReport()` method, which takes a report ID and data as arguments.

It's worth noting, as stated in the "Security Requirements" section, that this example assumes that the user has granted permission to the website to access HID devices, as this API requires explicit user permission.

Browser Compatibility

The WebHID API has limited browser support as of April 2023. Currently, it is only supported by Chromium-based browsers, including Google Chrome, Microsoft Edge, and Opera, only on desktop platforms. It is not yet supported on mobile platforms or other browsers such as Firefox or Safari **[121, 124]**.

WebHID API: `HID`, `connect`, `disconnect`, `getDevices`, `requestDevice`			
Browser	Version	Desktop Support	Mobile Support
Google Chrome	89+	Yes	No
Microsoft Edge	89+	Yes	No
Opera	75+	Yes	No
Mozilla Firefox	Not supported	No	No
Safari	Not supported	No	No

Table 3-30: HID, connect, disconnect, getDevices, requestDevice browser support

Usage Statistics

The latest available developer statistics for WebHID's usage come from the State of JS survey conducted in 2021. According to the survey, only 6.3% of respondents have reported that they have used the webHID API.

This statistic indicates that its adoption is still relatively low. This is likely due to the fact that the WebHID API is not widely supported by web browsers, even more so at the time that the survey was conducted.

According to the caniuse.com website, WebHID API is shown to have a global usage percentage of 28.18% across all users **[124]**.

Chrome Platform Status reports that the WebHID API is very rarely loaded on pages in Chrome, just barely touching 0.1% at the beginning of 2023 **[125, 126, 127]**.

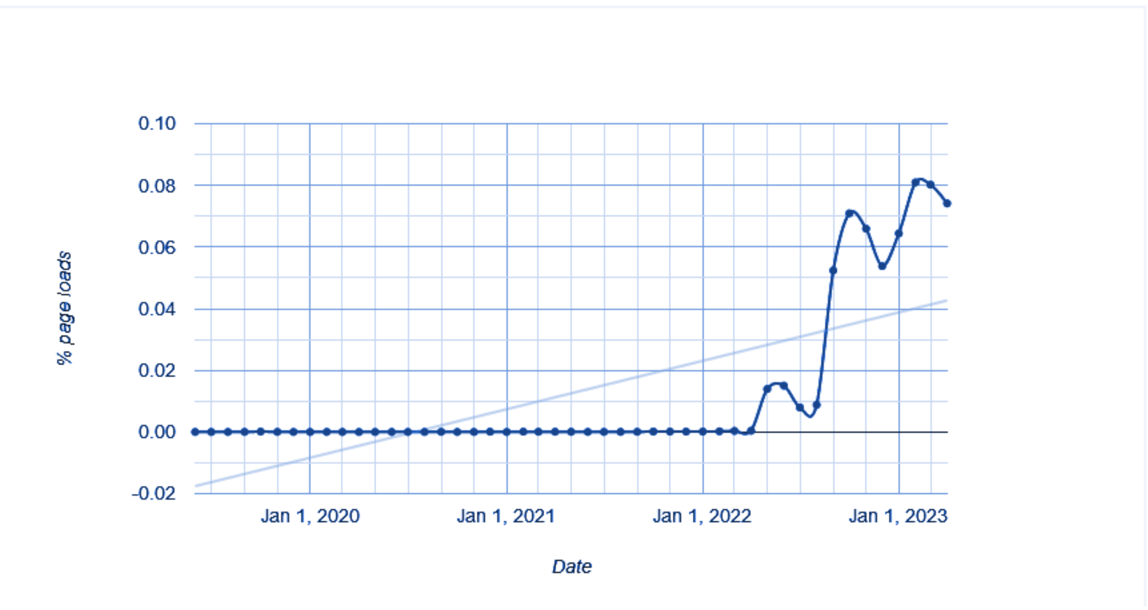

Figure 3-19: HidGetDevices percentage of page loads over time in Chrome.

`HidGetDevices`: Top five sample URLs
https://www.petersontuners.com/
https://app.pushground.com/
https://beta.webchartmd.com/
https://www.iqs.edu/
https://www.ecigstats.org/

Table 3-31: HidGetDevices top five sample URLs

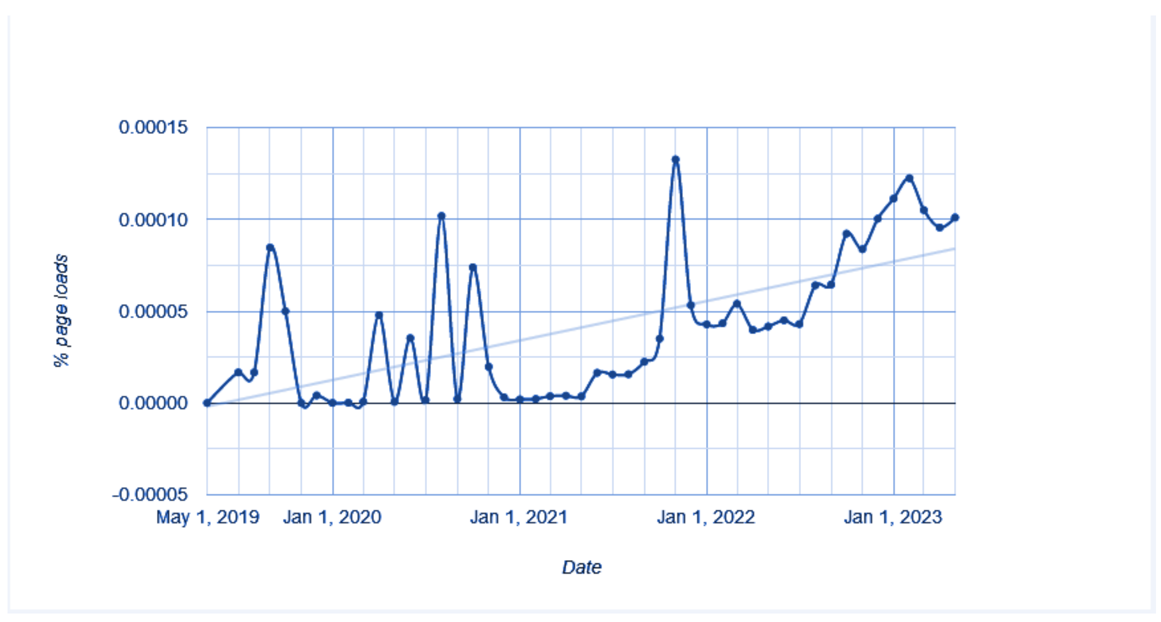

Figure 3-20: HidRequestDevice percentage of page loads over time in Chrome.

HidRequestDevice: Top five sample URLs
NO DATA AVAILABLE

Table 3-32: HidRequestDevice top five sample URLs

As the API continues to gain support from more web browsers and developers become more familiar with its capabilities we may see increased adoption in the future. Additionally, the increasing popularity of Internet of Things (IoT) devices and the need for web applications to interface with them could lead to increased adoption of the WebHID API in the future as well.

3.1.11 WebUSB API

General Description

The WebUSB API is an experimental browser API that is a Community Group Draft Report at W3C **[129]**. It allows web applications to communicate with USB devices. With this API, web applications can request permission to connect to a USB device, select the device from a list of available devices, read and write data to the device, and receive notifications when the device is plugged in or unplugged **[128, 129]**.

The WebUSB API is useful for a wide range of web applications that need to communicate with USB devices, such as game controllers, serial devices, smart cards, and more. By providing direct access to USB devices, the WebUSB API enables web applications to interact with the physical world in ways that were previously not possible.

To use the WebUSB API, the user must first grant permission to the web application to access the USB device. Once permission is granted, the web application can use the API to communicate with the device. The API provides methods for listing the available devices, selecting a device to connect to, reading and writing data to the device, and receiving notifications when the device is plugged in or unplugged **[130]**.

Overall, the WebUSB API opens up new possibilities for web applications to interact with USB devices and extends the capabilities of the web platform beyond the traditional boundaries of the browser.

Interfaces

The main interfaces of WebUSB API are:

1. `USB`: The main interface that represents a USB device. It provides methods to open a device, get device information, and transfer data to and from the device.
2. `USBDevice`: An interface that represents a connected USB device. It provides information about the device such as its vendor and product IDs, serial number, and device configuration.
3. `USBConfiguration`: An interface that represents a configuration of a USB device. It provides information about the configuration such as its interface and endpoint settings.

4. `USBInterface`: An interface that represents an interface of a USB device. It provides information about the interface such as its alternate settings and endpoints.
5. `USBEndpoint`: An interface that represents an endpoint of a USB device. It provides information about the endpoint such as its type, direction, and transfer characteristics.

All WebUSB Interfaces:

- `USB`: Provides attributes and methods for finding and connecting USB devices from a web page.
- `USBConnectionEvent`: This event type is passed to `USB.onconnect` or `USB.ondisconnect` when the user agent detects a new USB device has been connected to, or disconnected from the host.
- `USBDevice`: Provides access to metadata about a paired USB device and methods for controlling it.
- `USBInTransferResult`: Represents the result from requesting a transfer of data from the USB device to the USB host.
- `USBIsochronousInTransferPacket`: Represents the status of an individual packet from a request to transfer data from the USB device to the USB host over an isochronous endpoint.
- `USBIsochronousInTransferResult`: Represents the result from requesting a transfer of data from the USB device to the USB host.
- `USBIsochronousOutTransferPacket`: Represents the status of an individual packet from a request to transfer data from the USB host to the USB device over an isochronous endpoint.
- `USBIsochronousOutTransferResult`: Represents the result from requesting a transfer of data from the USB host to the USB device.
- `USBConfiguration`: Provides information about a particular configuration of a USB device and the interfaces that it supports.
- `USBInterface`: Provides information about an interface provided by the USB device.
- `USBAlternateInterface`: Provides information about a particular configuration of an interface provided by the USB device.

- `USBEndPoint`: Provides information about an endpoint provided by the USB device.

Security Requirements

The WebUSB API has some security requirements that must be met in order to ensure user safety and prevent unauthorized access. These include:

1. Origin restrictions: The API can only be accessed from a secure context (HTTPS) and only if the user grants explicit permission to the site.
2. User notification: Before the WebUSB API can be used, the user must be notified about the device that is being accessed and asked for permission to access it.
3. Device filtering: WebUSB only allows access to devices that have been explicitly paired with the user's computer. This prevents malicious sites from accessing arbitrary USB devices.
4. Time-limited access: The API only provides temporary access to USB devices, and the user can revoke access at any time.
5. Same-origin policy: The API only allows access to USB devices that are connected to the same origin as the web page that is using the API.

By implementing these security requirements, the WebUSB API ensures that users have control over which devices can be accessed and prevents malicious actors from gaining unauthorized access to sensitive USB devices.

Implementation Example

```
// Request access to the USB device
navigator.usb.requestDevice({ filters: [{ vendorId: 0x1234 }]
})
  .then(device => {
    console.log(`Connected to ${device.productName}
(${device.vendorId})`);

    // Open the device
    return device.open();
  })
  .then(device => {
    // Select a configuration
```

```
    return device.selectConfiguration(1);
  })
  .then(() => {
    // Claim an interface
    return device.claimInterface(0);
  })
  .then(() => {
    // Send a control transfer
    return device.controlTransferOut({
      requestType: 'class',
      recipient: 'interface',
      request: 0x22,
      value: 0x01,
      index: 0x02
    });
  })
  .catch(error => {
    console.error(error);
  });
```

This example requests access to a USB device with a vendor ID of 0x1234. Once a device is connected, it opens the device, selects a configuration, claims an interface, and sends a control transfer to the device **[130]**. Note that this is a simple example and that real-world implementations may require additional error handling and error checking

Browser Compatibility

The WebUSB API is currently supported by Google Chrome, version 61 and later, on desktop and 112 and later on Android platforms. The API is also supported by Edge and Opera, as well as Samsung Internet Browser **[133]**.

However, the WebUSB API is not supported by other major web browsers such as Firefox and Safari. Firefox in particular has adopted a "Harmful" position in regards to the API **[131]**.

WebUSB API: `getDevices`, `requestDevice`			
Browser	Version	Desktop Support	Mobile Support
Google Chrome	61+ | 112+	Yes	Yes

Microsoft Edge	79+	Yes	Yes
Opera	48+ | 73+	Yes	Yes
Mozilla Firefox	Not supported	No	No
Safari	Not supported	No	No

Table 3-33: getDevices, requestDevice browser support

Usage Statistics

Caniuse.com reports that the WebUSB API has a global usage percentage of 73.71 among all users, indicating high support **[132]**.

According to the Chrome Platform Status report, less than 0.01% of page loads on Chrome are using the API in May 2023 **[134, 135]**. The low usage could be due to limited browser support and security concerns, as the API allows web applications to interact with USB devices connected to the user's computer, potentially exposing sensitive information or allowing unauthorized access to the device.

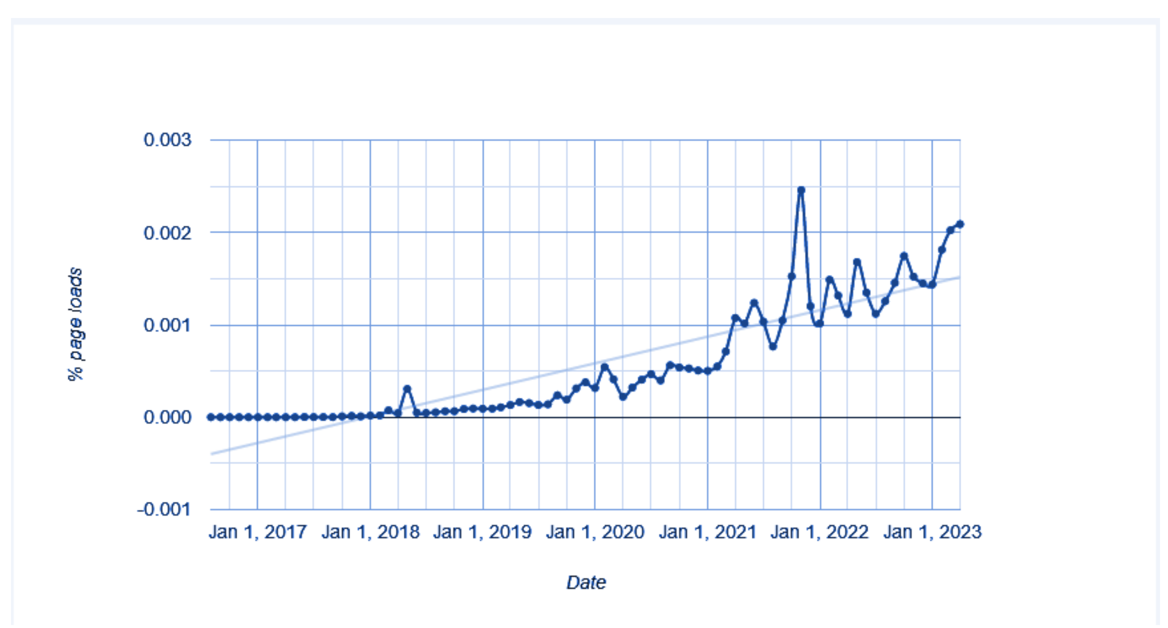

Figure 3-21: UsbGetDevices percentage of page loads over time in Chrome.

UsbGetDevices: Top five sample URLs
https://brevent.sh/
https://beta.shapeshift.com/

https://ui.perfetto.dev/
https://eberjegyzek.allamkincstar.gov.hu/
https://makecode.microbit.org/

Table 3-34: UsbGetDevices top five sample URLs

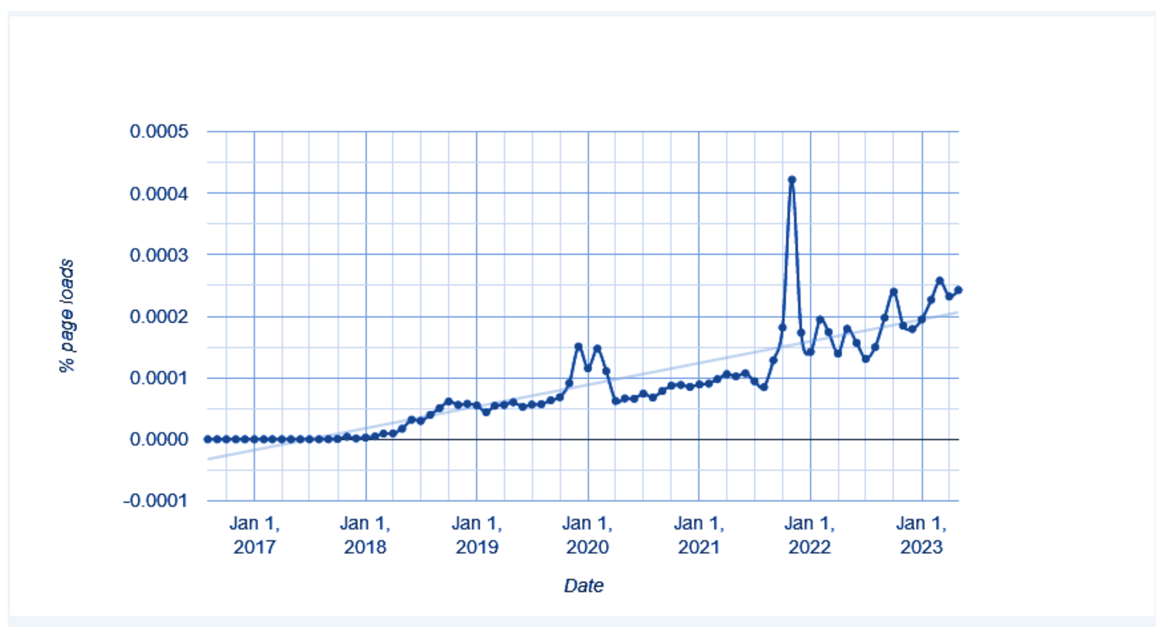

Figure 3-22: UsbRequestDevice percentage of page loads over time in Chrome.

`UsbRequestDevice`: Top five sample URLs
https://cryptyques.com/
https://sandbox.katon.io/
https://furymint.cryptyques.com/
NONE
NONE

Table 3-35: UsbRequestDevice top five sample URLs

3.1.12 WebXR Device API

General Description

The WebXR Device API is a web browser API, designated as a Candidate Recommendation Snapshot by the W3C **[136]**, that provides access to immersive computing devices, such as virtual reality (VR) headsets, augmented reality (AR) headsets, and other types of immersive devices **[136, 137]**. The API allows developers to create web-based applications that can interact with these devices, providing users with a more immersive and engaging experience.

The API was developed by the WebXR Device API Community Group, which includes representatives from companies such as Google, Mozilla, and Microsoft **[136]**. The API is designed to be compatible with a wide range of immersive devices, including both tethered and mobile devices.

The WebXR Device API provides a variety of features and capabilities for working with immersive devices. These include support for tracking the position and orientation of the user's head and hands, support for rendering 3D scenes and objects in real-time, and support for user input via hand and gesture recognition.

The older WebVR API is now deprecated and in essence replaced by WebXR **[142]**. One of the reasons was that the WebVR API was designed solely to support Virtual Reality (VR), WebXR provides support for both VR and Augmented Reality (AR) on the web **[138]**. Support for AR functionality is added by the WebXR Augmented Reality Module.

Interfaces

The WebXR Device API provides several interfaces for developers to create immersive experiences using virtual reality (VR) and augmented reality (AR) on the web.

The core interfaces of the WebXR Device API are:

1. `XRSystem`: This interface provides information about the available XR devices, as well as methods to request and exit an immersive session.

2. `XRSession`: This interface represents an active XR session, providing information such as the session type, environment parameters, and rendering context. It also provides methods for rendering to the session and managing session lifecycle events.
3. `XRFrame`: This interface represents a single frame of an XR session, providing access to the current pose of the user's head and hand controllers, as well as the ability to render and update the scene.
4. `XRInputSource`: This interface represents a physical input device, such as a VR headset or hand controller, providing information about its type, state, and position in the XR space.
5. `XRSpace`: This interface represents a coordinate space within an XR session, allowing developers to perform position and orientation transformations and to define the origin of the scene.
6. `XRRigidTransform`: This interface represents a transformation matrix that describes the position and orientation of an object in the XR space.
7. `XRWebGLLayer`: This interface represents a WebGL rendering layer within an XR session, providing access to the underlying WebGL context and the ability to render to the layer.

Security Requirements

The WebXR Device API has some security requirements to protect users from malicious actors. The following are some of the security requirements:

1. User consent: The user must grant explicit permission for the application to access the WebXR device. This permission must be granted on a per-session basis.
2. HTTPS: WebXR API requires the use of HTTPS for secure communication between the application and the device. This ensures that the data is encrypted in transit and cannot be intercepted by unauthorized parties.
3. Isolation: The WebXR device must be isolated from other resources in the browser to prevent malicious actors from accessing or manipulating the device.
4. Origin checks: The WebXR device must be accessed from a trusted origin. The browser must ensure that the WebXR API is only accessible from secure and trusted origins.

5. Content Security Policy (CSP): The application must have a CSP that limits the sources of content to prevent the injection of malicious code.

These security requirements help to ensure that WebXR devices are only accessed by trusted applications and that user data is protected.

Implementation Example

Below is an example of how the WebXR Device API can be implemented in a web application with JavaScript:

```
// Get the button that will start the AR session
const arButton = document.querySelector('#ar-button');

// Add a click event listener to the button
arButton.addEventListener('click', () => {
  // Check if the WebXR API is supported
  if (navigator.xr) {
      // Request an AR session
      navigator.xr.requestSession('immersive-ar', {
      // Specify the required features for the AR session
      requiredFeatures: ['hit-test'],
      // Specify the optional features for the AR session
      optionalFeatures: ['dom-overlay'],
      }).then((session) => {
      // Use the AR session to render the scene
      // ...
      // End the session when the user is done
      session.end();
      }).catch((error) => {
      // Handle any errors that may occur
      console.error(error);
      });
  } else {
      // The WebXR API is not supported
      console.error('WebXR not supported');
  }
});
```

In this example, the code first gets a button element with the ID "ar-button" using the `querySelector` method. Then, a click event listener is added to the button using the `addEventListener` method.

When the button is clicked, the code checks if the WebXR API is supported by checking if the `navigator.xr` property is defined. If the API is supported, an AR session is requested using the `requestSession` method. The method takes two arguments: The first is the type of session to request, which in this case is "immersive-ar" for an augmented reality session. The second argument is an object that specifies the required and optional features for the session. In this example, the required feature is "hit-test" for detecting the position and orientation of the device in the real world, and the optional feature is "dom-overlay" for overlaying HTML elements on top of the AR scene. If the session is successfully created, the code can use it to render the AR scene. When the user is done with the session, the `end` method is called to end the session. Finally, if the WebXR API is not supported, an error message is logged to the console.

Browser Compatibility

The WebXR Device API is very extensive. Many of its features are still in development and don't have finalized specifications yet. It is in active development and it is at least partially supported by most modern browsers, including Google Chrome, Mozilla Firefox, Microsoft Edge, and Oculus Browser **[139]**. However, the level of support may vary depending on the platform and the version of the browser.

- Google Chrome: Supports WebXR Device API on desktop and mobile versions, including the ability to render virtual and augmented reality experiences. Some features, such as hand tracking and motion controllers, are not available on all platforms.
- Mozilla Firefox: Supports WebXR Device API on desktop and mobile versions behind a developer flag. It has similar capabilities to Chrome.
- Microsoft Edge: Supports WebXR Device API on desktop and mobile versions, with similar capabilities to Chrome.
- Safari: Supports the API on the desktop version of the browser behind a flag. The mobile version of Safari does not support the WebXR Device API.

- Oculus Browser: Supports WebXR Device API on the Oculus Quest platform for virtual and augmented reality experiences.
- Samsung Internet: Supports WebXR Device API on Samsung Galaxy devices for virtual and augmented reality experiences.

Overall, support for the WebXR Device API is becoming increasingly widespread and robust, particularly for virtual and augmented reality experiences on desktop and mobile devices **[137, 139, 140]**.

WebXR API: `xr`			
Browser	Version	Desktop Support	Mobile Support
Google Chrome	79+ | 112+	Partial	Partial
Microsoft Edge	79+	Partial	Partial
Opera	66+ | 73+	Partial	Partial
Mozilla Firefox*	77+ | 110+	In Development	In Development
Safari**	13+	In Development	No

Table 3-36: xr browser support

*Can be enabled in Firefox behind the `dom.vr.webxr.enabled` flag.

**Can be enabled in Safari with the `WebXR Device API` experimental feature.

Usage Statistics

The State of JavaScript 2022 survey reports that only 2.5% of developers have used the WebXR API in their applications **[144]**. The usage results for the 2021 survey were at 2% **[145]**. This indicates that while there appears some growth in developer usage, the overall adoption remains low. Another interesting statistic that comes from the survey is that 82.4% of web developers have not even heard of the API, while 15.2% know about the API but have not used it in their applications.

Consistent with the results of the State of JS survey are the findings on the Google Chrome Platform Status page loads that show a monthly average of 2 - 2.5% during 2021 and a drop to about 1 - 1.5% in 2022 **[141]**.

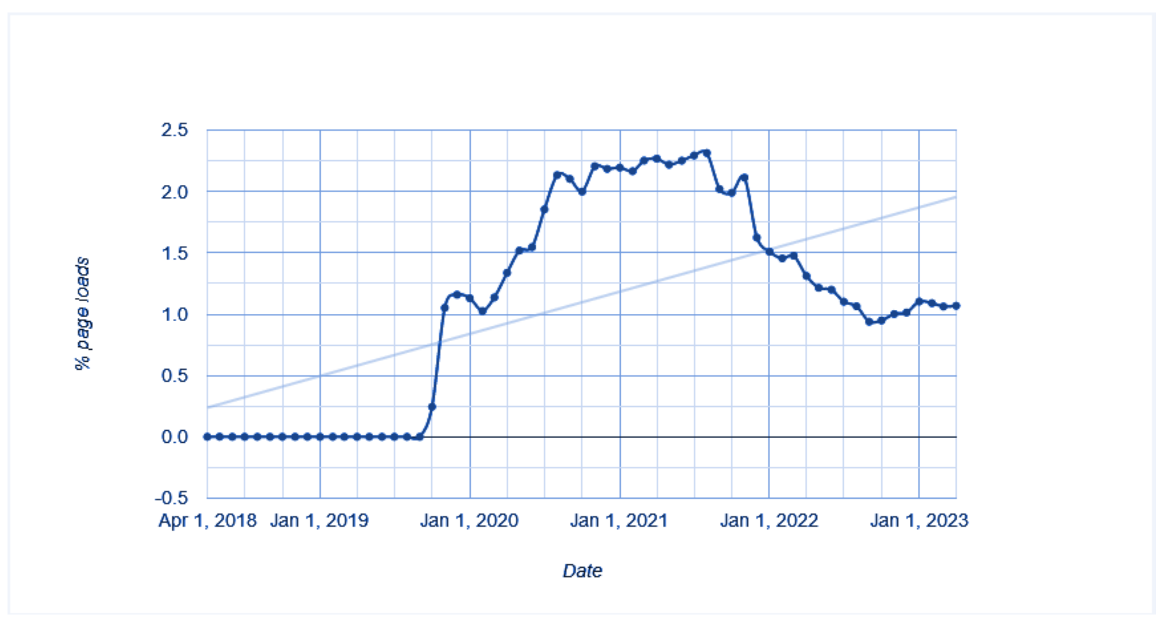

Figure 3-23: NavigatorXR percentage of page loads over time in Chrome.

`NavigatorXR`: Top five sample URLs
https://number12cider.com/
https://bmgmusic.sourceaudio.com/
https://www.lochassociates.co.uk/
https://hikariauto.livedoor.blog/
https://eunoiareview.wordpress.com/

Table 3-37: NavigatorXR top five sample URLs

According to the data, it appears that only a small fraction of web users can use applications that rely on the WebXR Device API. However, it is plausible that this API may become more popular in the future as virtual and augmented reality applications become more widespread.

Reference support table for experimental and emerging web APIs in major desktop and mobile browsers

Below is an overview of the the support for the various APIs that were presented in the previous section, in a single table:

Audio Output Devices	`selectAudioOutput()`	No	No	No	Nightly	No	No	No	No	No	No
	`setSinkId()`	49+	17+	36+	Nightly	No	No	No	No	No	No
	`sinkId`	49+	17+	36+	Nightly	No	No	No	No	No	No
Background Sync	`SyncManager, getTags, register`	49+	79+	36+	No	No	49+	79+	36+	No	No
	`In service workers`	61+	79+	No	No	No	61+	79+	No	No	No
Barcode Detection	`BarcodeDetector, BarcodeDetector() constructor, detect, getSupportedFormats`	88+	89+	69+	No	No	83+	89+	69+	No	No
Web Bluetooth	`Bluetooth`	56+	79+	43+	No	No	56+	79+	43+	No	No
	`availabilitychanged`	56+	79+	43+	No	No	No	No	No	No	No
	`getAvailability`	78+	79+	65+	No	No	78+	79+	56+	No	No
	`getDevices`	No	No	No	No	No	No	No	No	No	No

	`requestDevice`	56+	79+	43+	No	No	56+	79+	43+	No	No
Keyboard	`Keyboard`	68+	79+	55+	No	No	68+	79+	48+	No	No
	`KeyboardLayoutMap`	69+	79+	56+	No	No	69+	79+	48+	No	No
Presentation	`Presentation, defaultRequest`	47+	79+	34+	No	No	47+	79+	34+	No	No
	`receiver`	59+	79+	46+	No	No	59+	79+	43+	No	No
Web NFC	`NDEFReader, NDEFReader(), makeReadOnly, reading, readingerror, scan, write`	No	No	No	No	No	89+	No	63+	No	No
WebGPU	`GPU, getPreferredCanvasFormat, requestAdapter`	113	113	No	NIghtly	No	No	No	No	No	No
WebHID	`HID, connect, disconnect, getDevices, requestDevice`	89+	89+	75+	No	No	No	No	No	No	No
WebUSB	`getDevices, requestDevice`	61+	79+	48+	No	No	112+	79+	73+	No	No
WebXR	`xr`	79+	79+	66+	77+	13+	112+	79+	73+	110+	No
Color Legends: Green=Supported (Version), Red=Not Supported, Yellow=Partial Support or In Development											

Table 3-38: Reference support table for experimental and emerging web APIs in major browsers

Adoption Table of Experimental and Emerging Web APIs

Below is a table that ranks the APIs presented in the previous section according to Chrome page loads in the Spring of 2023. The APIs are ranked from highest percentage of total page loads in the Chrome web browser to the lowest.

Rank	API	Percentage
1	Presentation API	6%
2	Keyboard API	3%
3	WebXR Device API	1%
4	Web Bluetooth API	0.6%
5	Background Sync API	0.5%
6	Audio Output Device API	0.15%
7	WebHID API	0.08%
8	Web GPU API	0.01%
9	WebUSB API	0.002%
10	Barcode Detection API	0.0006%
11	Web NFC API	0.0001%
12	Compute Pressure API	No Data

Table 3-39: Adoption Table of Experimental and Emerging Web APIs

From the data in the table, three major categories seem to emerge. The first category that can clearly be recognized, is the most popular experimental and emerging APIs, according to their page loads in Chrome. These are the Presentation API, the Keyboard API and the WebXR API. All of them show percentages of page loads in Chrome above 1% which is impressive for the fact that these APIs are still considered experimental and not production-ready. This potentially shows that there is a real need that these APIs satisfy among users that need their functionality. It could also potentially mean that more developers are leveraging these APIs in order to build useful applications that in turn users employ in order to fulfill their computing needs.

A second category that emerges from this table is the APIs that have “medium” adoption. Namely, Web Bluetooth, Background Sync and Audio Output Devices. These are APIs that show potential for the future as there appears to be adequate support from developers in writing web applications that support these APIs. Users also appear to be interested in the functionality that these APIs offer.

The third category is the APIs that have small percentages of page loads, below 0.1%. These APIs seem to have not gained traction among developers and users. Some of them are in active development, so a little more time is needed before they can be deployed by web application developers, while others like WbNFC, have caused controversy in the community and do not have the support from all browser vendors for the foreseeable future. Regardless, the APIs in the last category are still in flux and their path forward still remains to be seen.

The above table and the data collected can be used in order to identify likely trends for the development of web applications, as well as the further development of web APIs. It is a useful observation on the mobility of the web platform in 2023 and a snapshot that can potentially be used in the future, in order to draw useful conclusions on the progress towards the goal of the web browser as an operating system.

4. Epilogue

4.1 Conclusions

The implementation and usage of experimental and emerging web APIs, alongside the evolution of the web and web browsers highlights the importance of innovation. It also accentuates collaboration in the development of technology, as well as the need for ongoing efforts to address challenges in order to ensure the continued growth and success of the web platform. As it becomes evident from the collected data of this research during the spring of 2023, the continued evolution of experimental and emerging web APIs will most likely lead to even more powerful and sophisticated applications and services. However, this growth will also require major efforts in order to address challenges such as standardization and security.

Support for experimental APIs is a continuous and ongoing effort on the part of web browser vendors and developers. However, as the research shows, the Chrome browser is where most of the development and implementation is materialized. Chrome is by far the most popular web browser and has the support of Google with its seemingly endless amount of resources. This domination in the Web ecosystem by Chrome could also present a challenge for the shaping of future web APIs, as it seems probable that web developers will feel inclined to abandon other browser engines and focus on the most popular. The open-source Chromium project that serves as a basis for Chrome seems to mitigate this challenge for the time being, but it is not certain if it will continue to do so in the long run, any more than the open-source AOSP (Android Open Source Project) has fostered diversity in the mobile OS ecosystem.

A very important role in this ongoing process is played, and will be played more so in the future, by the World Wide Web Consortium (W3C). The consortium promotes open standards and interoperability and helps foster a diverse ecosystem of web APIs, where developers can choose from a variety of APIs that work across different platforms and browsers. This helps ensure that developers have access to the tools and technologies they need to build innovative web applications that can reach a wide audience. To achieve this goal, the W3C publishes web standards and guidelines that help ensure that web technologies are consistent, accessible, and interoperable. They also provide a forum

for developers, browser vendors, and other stakeholders to collaborate and discuss new technologies and standards.

4.2 Limits of the research

While the research was in no way limited in regards to availability of relevant information and written specifications for experimental APIs, it is limited in terms of gathering statistical usage and development data due to several reasons.

First, experimental web APIs are not standardized, and their usage is not widespread. Therefore, it is difficult to collect sufficient data on their usage and performance across different platforms and devices.

Second, experimental web APIs are often used by early adopters and tech enthusiasts, who represent a very small portion of the overall user base. Therefore, the usage data collected from these users may not be representative of the general population.

Third, many experimental web APIs are still in the early stages of development, and their functionality and features are subject to change. This makes it difficult to provide accurate statistics on their usage and performance over time.

4.3 Future Expansions

The topic of this master's thesis can be expanded in terms of various aspects; performance comparisons, compatibility, security risks, user experience, user interface design and standardization efforts. Further expansions in the study of experimental web browser APIs related to hardware and the overall concept of the browser as an operating system could include the following:

- **In-depth analysis of the performance and resource utilization of experimental hardware-related web APIs compared to traditional desktop applications.**

This analysis could involve a comprehensive examination of how these APIs perform and utilize system resources in different scenarios. It could aim to evaluate the feasibility

and effectiveness of using web APIs for interacting with hardware devices, such as cameras, microphones, sensors, and other peripherals.

This type of analysis typically involves conducting various experiments and measurements to assess the performance and resource utilization of these APIs. Key metrics that are often considered include response time, latency, throughput, CPU usage, memory usage, power consumption, and network utilization. These measurements help in understanding how efficiently the experimental web APIs interact with hardware devices and how they compare to traditional desktop applications in terms of performance and resource usage.

The analysis may also involve benchmarking the experimental web APIs against established performance standards and industry best practices. This allows for a comprehensive evaluation of their performance capabilities and provides insights into their potential for real-world usage.

Furthermore, the analysis may explore different scenarios and use cases to understand how the experimental hardware-related web APIs perform under various conditions. This could involve assessing their performance with different hardware configurations, network environments, and levels of user interaction.

The results of such analysis would provide valuable insights into the strengths, limitations, and potential improvements of experimental hardware-related web APIs. They would undoubtedly help developers, researchers, and stakeholders make informed decisions regarding the adoption and further development of these APIs. Additionally, this analysis would contribute to the overall advancement of web-based technologies and their integration with the Internet of Things (IoT) ecosystem.

- **Investigation of the compatibility of experimental web APIs with various hardware configurations and operating systems.**

This investigative study could involve assessing how these APIs function across different devices, architectures, and software environments. It could aim to understand the extent to which the experimental APIs can seamlessly interact with diverse hardware configurations and operating systems, ensuring broad compatibility and consistent performance. This involves testing the experimental web APIs on a range of hardware

devices, including desktop computers, laptops, tablets, smartphones, and IoT devices. Different hardware configurations, such as varying processing power, memory capacity, graphics capabilities, and input/output capabilities, are taken into account to evaluate how the APIs perform under different resource constraints.

Furthermore, compatibility testing includes assessing the behavior of experimental web APIs across various operating systems, such as Windows, macOS, Linux, Android, and iOS. This involves verifying if the APIs are able to function properly, adhere to standard specifications, and provide consistent results across different platforms. This could also involve adapting the APIs to work with specific hardware configurations or optimizing their performance for different operating systems. The goal is to ensure that the experimental web APIs can be reliably used across a wide range of devices and platforms, enhancing their versatility and usability.

The results of this investigation would provide insights into the compatibility challenges associated with experimental web APIs and help guide developers in making informed decisions about their implementation and deployment. It would also highlight the importance of ensuring cross-platform compatibility and robustness when designing and evolving experimental web APIs.

- **Examination of the security risks associated with the use of experimental hardware-related web APIs in web applications**.

This type of study could involve a thorough analysis of potential vulnerabilities and threats that may arise when utilizing these APIs in real-world scenarios.

One aspect of the examination is identifying and assessing the specific security risks that experimental hardware-related web APIs may introduce. This includes evaluating potential attack vectors, such as unauthorized access to hardware devices, data breaches, privacy concerns, and the potential for malicious code execution. Understanding these risks helps in developing appropriate security measures and mitigations to protect against potential exploits.

Additionally, the examination could involve evaluating the security measures implemented within the experimental hardware-related web APIs themselves. This includes reviewing the API's design, authentication mechanisms, data handling practices,

and any security features or protocols that are in place to ensure secure communication between the web application and the hardware device.

The analysis should also consider the broader security implications of integrating hardware-related web APIs into web applications. This includes assessing how the APIs interact with other components of the application ecosystem, such as backend servers, databases, and user interfaces, and evaluating the potential impact on the overall security posture of the system. The examination could also include exploring best practices and guidelines for secure implementation and usage of experimental hardware-related web APIs. This may involve recommendations for developers on how to properly handle user permissions, implement secure communication protocols, perform input validation, and apply other security measures to mitigate potential risks.

By conducting a comprehensive examination of the security risks associated with experimental hardware-related web APIs, developers and security professionals can gain insights into the potential vulnerabilities and take proactive measures to safeguard web applications and users' data. This examination could play a crucial role in ensuring the secure and responsible adoption of these APIs in the evolving landscape of web development

- **Study of the impact of experimental web APIs on user experience and user interface design in web applications.**

This study would involve several key aspects to assess their influence. Below are some components that could be included in such a study:

1. User Experience Evaluation: Conducting user testing and gathering feedback from participants who interact with web applications that leverage experimental web APIs. The study would assess factors such as ease of use, intuitiveness, efficiency, and overall satisfaction with the user experience.
2. Performance Analysis: The study would measure and compare the performance of web applications using experimental web APIs against those without these APIs. This analysis would include factors such as page load times, responsiveness, smoothness of animations or transitions, and resource consumption.

3. Usability Testing: Usability testing would focus on evaluating the usability of web applications incorporating experimental web APIs. This includes examining the learnability, efficiency, error prevention, and user control aspects of the interface design.
4. User Interface Design Evaluation: The study would analyze the impact of experimental web APIs on the visual design and layout of web applications. It would assess whether the APIs enable the creation of more visually appealing, interactive, and engaging user interfaces.
5. User Perception and Feedback: Feedback from users, collected through surveys, interviews, or questionnaires, would be crucial to understanding their perceptions of the web applications utilizing experimental web APIs. This feedback would provide insights into their preferences, attitudes, and overall satisfaction with the user interface and experience.
6. Comparative Analysis: A comparative analysis could be conducted to compare the user experience and user interface design of web applications using experimental web APIs to those relying on traditional web technologies. This would help identify the specific benefits and improvements brought about by the adoption of experimental APIs.
7. Case Studies: In-depth case studies of specific web applications or services that have implemented experimental web APIs could be conducted to explore their impact on user experience and interface design in real-world scenarios. These case studies would provide practical insights and highlight best practices.

By considering these aspects, a study on the impact of experimental web APIs on user experience and user interface design would provide valuable insights into how these APIs enhance the overall usability, aesthetics, and user satisfaction of web applications. The findings would guide developers and designers in leveraging the potential of experimental web APIs to create more engaging and user-friendly web experiences.

- **Evaluation of the standardization efforts by industry organizations such as W3C to ensure compatibility and interoperability of experimental web APIs across different browsers and platforms.**

The evaluation of standardization efforts by industry organizations such as the World Wide Web Consortium (W3C) plays a crucial role in ensuring compatibility and

interoperability of experimental web APIs across different browsers and platforms. This type of analysis would involve assessing the effectiveness and impact of standardization initiatives in the following ways:

1. Specification Development: The study would examine the process of developing specifications for experimental web APIs within the W3C or other industry organizations. This includes evaluating the level of community involvement, collaboration, and transparency in the standardization process. It would also assess how well-defined and comprehensive the specifications are in addressing the functionality, behavior, and security aspects of the APIs.
2. Browser Vendor Adoption: The study would assess the level of adoption and implementation of the standardized experimental web APIs by major browser vendors. This involves analyzing the compatibility and consistency of API support across different browsers and versions. It would also consider the adherence to the standardized specifications, including any deviations or vendor-specific extensions.
3. Interoperability Testing: The study would evaluate the interoperability of experimental web APIs across different browsers and platforms. This would involve conducting tests and experiments to assess how well the APIs function in real-world scenarios across various combinations of browsers, operating systems, and hardware configurations. It would also identify any potential issues or inconsistencies that arise when using the APIs across different environments.
4. Developer Feedback and Adoption: The study would gather feedback from developers who have worked with standardized experimental web APIs. This includes their experiences, challenges, and suggestions related to the compatibility and interoperability of these APIs. It would also analyze the adoption rate of standardized APIs by the developer community, considering factors such as ease of implementation, available resources, and community support.
5. Impact on Web Ecosystem: The study would examine the broader impact of standardization efforts on the web ecosystem. This includes evaluating how standardized experimental web APIs contribute to a more stable and predictable development environment, fostering innovation, and encouraging the creation of cross-platform web applications. It would also consider the influence of

standardization on market trends, industry practices, and the overall advancement of web technologies.

By conducting an evaluation of standardization efforts, the study would provide insights into the effectiveness and benefits of standardized experimental web APIs. It should help identify areas for improvement, highlight successful standardization cases, and guide future standardization efforts to ensure the compatibility and interoperability of these APIs, thereby facilitating the widespread adoption and utilization of experimental web technologies.

- **Comparison of the experimental APIs offered by different browsers and platforms and their respective strengths and weaknesses.**

A study comparing the experimental APIs offered by different browsers and platforms and their respective strengths and weaknesses would involve several key components:

1. API Inventory: The study would start by creating an inventory of experimental APIs offered by different browsers and platforms. This includes identifying and categorizing the APIs based on their functionality, such as hardware access, multimedia, storage, or communication.
2. API Feature Analysis: The study would analyze the features and capabilities of each experimental API. This involves assessing the level of support for specific functionality, the flexibility of the API design, and the ease of use for developers. It would also consider factors such as performance, reliability, and compatibility with different browsers and platforms.
3. Compatibility and Interoperability: The study would evaluate the compatibility and interoperability of the experimental APIs across different browsers and platforms. This includes assessing how well the APIs adhere to industry standards and specifications, and if there are any vendor-specific implementations or deviations. It would also consider any compatibility challenges or limitations when using the APIs in real-world scenarios.
4. Performance and Efficiency: The study would compare the performance and efficiency of the experimental APIs in terms of resource utilization, responsiveness, and overall execution speed. This includes benchmarking and

testing the APIs in various scenarios to identify any performance bottlenecks or areas of improvement.

5. Developer Feedback and Adoption: The study would gather feedback from developers who have used the experimental APIs in their applications. This includes their experiences, challenges, and opinions regarding the strengths and weaknesses of the APIs. It would also consider factors such as developer documentation, community support, and the availability of resources for learning and development.
6. Use Case Analysis: The study would examine different use cases where the experimental APIs are applied. This includes identifying the domains or industries that benefit the most from specific APIs and understanding the potential impact and value they bring to web applications. It would also consider the versatility and flexibility of the APIs in addressing a wide range of application scenarios.

By conducting such a study, it provides an objective and comprehensive comparison of the experimental APIs offered by different browsers and platforms. It helps developers and decision-makers in selecting the most suitable APIs for their specific requirements, considering factors such as compatibility, performance, and developer support. Additionally, the study can highlight areas for improvement and guide future development efforts to enhance the strengths and address the weaknesses of these experimental APIs.

5. Bibliography

[1] “Experimental Technologies”, MDN Web Docs, Mozilla.org, https://developer.mozilla.org/en-US/docs/MDN/Writing_guidelines/Experimental_deprecated_obsolete#experimental (accessed Jan. 2013)
[2] “Chrome Experimental APIs”, Chrome Developers, https://developer.chrome.com/docs/extensions/reference/experimental/ (accessed Jan, 2013)
[3] “Value of Creating Standards at W3C”, W3C.org, https://www.w3.org/standards/about.html (accessed Jan. 2013)
[4] “Goodbye, Desktop Apps. Modern web applications are replacing your favorite desktop apps”, Medium, https://levelup.gitconnected.com/goodbye-desktop-apps-bf4d2c0438c4 (accessed May. 2023)
[5] “Introduction to web APIs”, MDN Web Docs, Mozilla.org, https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Client-side_web_APIs/Introduction (accessed Nov. 2022)
[6] Philipp Fleck, Dieter Schmalstieg, and Clemens Arth. 2020. Creating IoT-ready XR-WebApps with Unity3D. In The 25th International Conference on 3D Web Technology (Web3D '20). Association for Computing Machinery, New York, NY, USA, Article 1, 1–7. https://doi.org/10.1145/3424616.3424691
[7] T. Leppänen, A. Heikkinen, A. Karhu, E. Harjula, J. Riekki and T. Koskela, "Augmented Reality Web Applications with Mobile Agents in the Internet of Things," *2014 Eighth International Conference on Next Generation Mobile Apps, Services and Technologies*, Oxford, UK, 2014, pp. 54-59, doi: 10.1109/NGMAST.2014.24.
[8] A. Taivalsaari, T. Mikkonen, D. Ingalls and K. Palacz, "Web Browser as an Application Platform," *2008 34th Euromicro Conference Software Engineering and Advanced Applications*, Parma, Italy, 2008, pp. 293-302, doi: 10.1109/SEAA.2008.17
[9] “A Short History of the web”, CERN, https://home.cern/science/computing/birth-web/short-history-web (accessed Nov. 2022)
[10] “The History of Web Browsers”, Mozilla.org, https://www.mozilla.org/en-US/firefox/browsers/browser-history/ (accessed Dec. 2022)

[11] Manuel Castells, “The Impact of the Internet on Society: A Global Perspective”, OpenMind, https://www.bbvaopenmind.com/en/articles/the-impact-of-the-internet-on-society-a-global-perspective/ (accessed Feb. 2023)
[12] “Why your browser is becoming an OS” neverinstall.com, https://blog.neverinstall.com/why-your-browser-is-becoming-an-os/ (accessed May 2023)
[13] “Browser OS could turn the browser into the new desktop”, TechTarget, https://www.techtarget.com/searchvirtualdesktop/news/252467524/Browser-OS-could-turn-the-browser-into-the-new-desktop (accessed on 05/2023)
[14] “ESG Brief: Has the Web Browser Become the New Operating System?”, Enterprise Strategy Group, https://www.esg-global.com/research/esg-brief-has-the-web-browser-become-the-new-operating-system (accessed on 05/2023)
[15] Nyrhinen, Feetu & Mikkonen, Tommi. (2009). Web Browser as a Uniform Application Platform: How Far Are We?. 578-584. 10.1109/SEAA.2009.37.
[16] Michalis Diamantaris, Francesco Marcantoni, Sotiris Ioannidis, and Jason Polakis. 2020. The Seven Deadly Sins of the HTML5 WebAPI: A Large-scale Study on the Risks of Mobile Sensor-based Attacks. ACM Trans. Priv. Secur. 23, 4, Article 19 (November 2020), 31 pages. https://doi.org/10.1145/3403947
[17] “All Standards and Drafts”, W3C, https://www.w3.org/TR/ (accessed Jan 2023)
[18] L. Wyse and S. Subramanian, "The Viability of the Web Browser as a Computer Music Platform," in *Computer Music Journal*, vol. 37, no. 4, pp. 10-23, Dec. 2013, doi: 10.1162/COMJ_a_00213.
[19] Q. H. Do, P. Hosseyni, R. Küsters, G. Schmitz, N. Wenzler and T. Würtele, "A Formal Security Analysis of the W3C Web Payment APIs: Attacks and Verification," *2022 IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, USA, 2022, pp. 215-234, doi: 10.1109/SP46214.2022.9833681.
[20] Chavan, Mr & Bhatkar, Mr & Muley, Kirti. (2022). Progressive Web Apps vs Responsive Web Apps. International Journal of Advanced Research in Science, Communication and Technology. 211-214. 10.48175/IJARSCT-5668.
[21] Ali, Sarwar & Grover, Chetna & Chaudhary, Renu. (2022). Progressive Web Apps (PWAs)—Alternate to Mobile and Web. 10.1007/978-981-19-4193-1_55.

[22] Tandel, Sayali & Jamadar, Abhishek. (2018). Impact of Progressive Web Apps on Web App Development. 10.15680/IJIRSET.2018.0709021.
[23] “XMLHttpRequest”, Wikipedia.org, https://en.wikipedia.org/wiki/XMLHttpRequest (accessed, Mar. 2023)
[24] “XMLHttpRequest Living Standard”, whatwg.org, https://xhr.spec.whatwg.org/ (accessed, May. 2023)
[25] “WebGL: 2D and 3D graphics for the web”, Mozilla.org, https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API (accessed Mar. 2023)
[26] Michel Buffa. Audio on the Web (a brief history). SMC 2022 - Sound and Music Computing, Jun 2022, Saint-Etienne, France.. hal-03871527
[27] “Web Audio API”, Mozilla.org, https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API (accessed, May. 2023)
[28] “About W3C”, W3C Consortium, https://www.w3.org/Consortium/ (accessed, Mar. 2023)
[29] N. Peters, S. Park, D. Clifford, S. Kyostila, R. Mcllroy, B. Meurer, H. Payer, and S. Chakraborty. 2018. Phase-Aware Web Browser Power Management on HMP Platforms. In Proceedings of the 2018 International Conference on Supercomputing (ICS '18). Association for Computing Machinery, New York, NY, USA, 274–283. https://doi.org/10.1145/3205289.3205293
[30] Tiberkak, A., Hentout, A. & Belkhir, A. WebRTC-based MOSR remote control of mobile manipulators. *Int J Intell Robot Appl* (2023). https://doi.org/10.1007/s41315-023-00281-3
[31] Z. Du, Z. Xu, F. Dong and D. Shen, "A Novel Solution of Cloud Operating System Based on X11 and Docker," *2017 Fifth International Conference on Advanced Cloud and Big Data (CBD)*, Shanghai, China, 2017, pp. 26-32, doi: 10.1109/CBD.2017.13.
[32] Garad, Sonali & Bendale, Shailesh. (2021). Progressive Web Apps: A Seamless User Interface. 2582-5208.
[33] Vesic, Slavimir. (2016). Progressive Web Apps: between native and mobile Web applications. Info M.
[34] “Understanding the Role of Rendering Engine in Browsers”, BrowserStack, https://www.browserstack.com/guide/browser-rendering-engine (accessed, Feb, 2023)

[35] “Web Browser Market Share”, StatCounter, https://gs.statcounter.com/browser-market-share#monthly-200901-202303 (accessed May. 2023)
[36] “The browser is the operating system 10 years on … on mobile”, The Guardian, https://www.theguardian.com/technology/2014/apr/09/software-internet (accessed May 2023)
[37] L. Kumar, R. Kushwaha and R. Prakash, "Design & Development of Small Linux Operating System for Browser Based Digital Set Top Box," *2009 First International Conference on Computational Intelligence, Communication Systems and Networks*, Indore, India, 2009, pp. 277-281, doi: 10.1109/CICSYN.2009.81.
[38] Roumeliotis, Konstantinos & Tselikas, Nikolaos. (2022). Evaluating Progressive Web App Accessibility for People with Disabilities. Network. 2. 350-369. 10.3390/network2020022.
[39] Karavashkin, Lev & Molodyakov, Sergey & Medvedev, Boris. (2023). Caching Data in a Web Audio Service Using Progressive Web Apps Technologies. 10.1007/978-3-031-20875-1_34.
[40] R. Selby, "Platforms for Software Execution: Databases vs. Operating Systems vs. Browsers," *Proceedings of the (19th) International Conference on Software Engineering*, Boston, MA, USA, 1997, pp. 578-578.
[41] “Back to Basics: Browser Versus Operating System”, Dr. Networking, http://www.drnetworking.net/july-2017/back-basics-browser-versus-operating-system/ (accessed on 05/2023)
[42] Chijin Zhou, Quan Zhang, Mingzhe Wang, Lihua Guo, Jie Liang, Zhe Liu, Mathias Payer, and Yu Jiang. 2022. Minerva: browser API fuzzing with dynamic mod-ref analysis. Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022). Association for Computing Machinery, New York, NY, USA, 1135–1147. https://doi.org/10.1145/3540250.3549107
[43] “Web API Specifications”, Mozilla Developer Network, https://developer.mozilla.org/en-US/docs/Web/API (accessed May 2023)
[44] “Chromium Developer Portal”, The Chromium Projects, https://www.chromium.org/developers/ (accessed, Apr. 2023)
[45] “Documentation for Web Platform APIs”, Chrome Developers, https://developer.chrome.com/docs/web-platform/ (accessed Apr. 2023)

[46] “Experimental features in Firefox”, MDN Web Docs, https://developer.mozilla.org/en-US/docs/Mozilla/Firefox/Experimental_features (accessed Apr. 2023)
[47] “Safari Developer Resources”, Apple Developer Portal, https://developer.apple.com/safari/resources/ (accessed Apr. 2023)
[48]“Android Open Source Project”, AOSP, https://source.android.com/ (accessed May 2023)
[49] “Phone, tablet users spend more time with apps than Web”, CNN Business, https://edition.cnn.com/2012/01/20/tech/mobile/apps-web-gahran/index.html (accessed May 2023)
[50] “The Majority of Americans’ Mobile Time Spent Takes Place in Apps”, Insider Intelligence, https://www.insiderintelligence.com/content/the-majority-of-americans-mobile-time-spent-takes-place-in-apps (accessed May 2023)
[51] “Meet Chrome OS”, Google.com, https://www.google.com/chromebook/chrome-os/ (accessed Jan 2023)
[52] “Introducing Mozilla WebThings - Mozilla Hacks”, Mozilla.org, https://hacks.mozilla.org/2019/04/introducing-mozilla-webthings/ (accessed Feb 2023)
[53] “webianShell - A graphical shell for the web”, Webian.org, https://www.webian.org/shell/ (accessed Feb 2023)
[54] “W3C: All standards and drafts”, W3C.org, https://www.w3.org/TR/?tag=webapi (accessed May 2023)
[55] “MDN Web Docs: Web APIs”, Mozilla.org, https://developer.mozilla.org/en-US/docs/Web/API, (accessed May 2023)
[56] “Chrome Developers”. Google.com, https://developer.chrome.com/ (accessed May 2023)
[57] “Can I use...Browser support tables for modern web technologies”, caniuse.com, https://caniuse.com/ (accessed May 2023)
[58] “MDN Web Docs: Web APIs”, Mozilla.org, https://developer.mozilla.org/en-US/docs/Web/API (accessed Apr 2023)
[59] “HTML & JavaScript usage metrics”, chromestatus.com, https://chromestatus.com/metrics/feature/popularity (accessed May 2023)
[60] “ State of JavaScript - The annual developer survey of the JavaScript ecosystem”, stateofjs.com, https://stateofjs.com/en-us/ (accessed Mar 2023)

[61] “Audio Output Devices API”, Mozilla.org, https://developer.mozilla.org/en-US/docs/Web/API/Audio_Output_Devices_API (accessed Mar 2023)
[62] “Audio Output Devices API”, W3C.org, https://w3c.github.io/mediacapture-output/ (accessed Mar 2023)
[63] “Issue 648286: Unable to send audio output to earpiece for WebRTC audio-only use case on Android phones” Chromium.org, https://bugs.chromium.org/p/chromium/issues/detail?id=648286 (accessed Apr 2023)
[64] “Can I use audioout?”, caniuse.com, https://caniuse.com/?search=audioout (accessed Mar 2023)
[65] “Can I use sinkid?”, caniuse.com, https://caniuse.com/?search=html%20sinkid (accessed Mar 2023)
[66] “HTMLMediaElementSetSinkId Feature Popularity Timeline”,Chrome Platform Status,, chromestatus.com, https://chromestatus.com/metrics/feature/timeline/popularity/3877 (accessed Mar 2023)
[67] “Web Background Synchronization”, W3C.org, https://wicg.github.io/background-sync/spec (accessed Mar 2023)
[68] “Background Synchronization API”, Mozilla.org, https://developer.mozilla.org/en-US/docs/Web/API/Background_Synchronization_API, (accessed Mar 2023)
[69] “Web BackgroundSync API SyncManager implementation”, Mozilla.org, https://bugzilla.mozilla.org/show_bug.cgi?id=1217544 (accessed Apr 2023)
[70] “Introducing Background Sync”, chrome.com, https://developer.chrome.com/blog/background-sync (accessed Apr 2023)
[71] “Can I use syncmanager?”, caniuse.com, https://caniuse.com/?search=syncmanager (accessed May 2023)
[72] “BackgroundSync Feature Popularity Timeline”,Chrome Platform Status, chromestatus.com, https://chromestatus.com/metrics/feature/timeline/popularity/745 (accessed May 2023)
[73] “PeriodicBackgroundSync Feature Popularity Timeline”,Chrome Platform Status, chromestatus,com, https://chromestatus.com/metrics/feature/timeline/popularity/2930 (accessed May 2023)
[74] “Barcode Detection API”, W3C.org, https://wicg.github.io/shape-detection-api/#barcode-detection-api (accessed Feb 2023)

[75] “Barcode Detection API”, MDN Web Docs, Mozilla.org,,
https://developer.mozilla.org/en-US/docs/Web/API/Barcode_Detection_API (accessed Feb 2023)
[76] “Working with the BarcodeDetector”, Chrome Developers Article, chrome.com, https://developer.chrome.com/articles/shape-detection/#barcodedetector (accessed Feb 2023)
[77] “Can I use barcodedetector?, caniuse.com,
https://caniuse.com/?search=barcodedetector (accessed Feb 2023)
[78] “BarcodeDetectorDetect Feature Popularity Timeline”,Chrome Platform Status, chromestatus.com, https://chromestatus.com/metrics/feature/timeline/popularity/3711 (accessed Feb 2023)
[79] “ShapeDetection_BarcodeDetectorConstructor Feature Popularity Timeline”,Chrome Platform Status, chromestatus.com,
https://chromestatus.com/metrics/feature/timeline/popularity/1991 (accessed Feb 2023)
[80] “Same Origin- HTML Standard”, W3C.org,
https://html.spec.whatwg.org/multipage/browsers.html#same-origin (accessed June 2023)
[81] “Compute Pressure Level 1”, W3C.org,
https://www.w3.org/TR/2023/WD-compute-pressure-20230613/ (accessed June 2023)
[82] “Compute Pressure API”, Origin Trial, Chrome Developers Documentation,
https://developer.chrome.com/docs/web-platform/compute-pressure (accessed June 2023)
[83] “Compute Pressure Levels”, Web Platform Tests, github.org,
https://github.com/web-platform-tests/wpt/labels/compute-pressure (accessed May 2023)
[84] “Web Platform Design Principles”, W3C.org,
https://w3ctag.github.io/design-principles/#device-ids, (accessed Mar 2023)
[85] “Web Bluetooth”, Web Bluetooth Community Group, W3C.org,
https://webbluetoothcg.github.io/web-bluetooth (accessed May 2023)
[86] “WebBluetoothCG Demos, Web Bluetooth Community Group,
https://github.com/WebBluetoothCG/demos (accessed May 2023)
[87] “Communicating with Bluetooth devices over JavaScript”, Chrome Developers Article, Google.com, https://developer.chrome.com/articles/bluetooth (accessed May 2023)
[88] “Web Bluetooth Implementation Status”, Web Bluetooth Community Group, github.com,

https://github.com/WebBluetoothCG/web-bluetooth/blob/main/implementation-status.md (accessed May 2023)
[89] “Web Bluetooth Position”, Mozilla Standards Positions, Mozilla.org, https://mozilla.github.io/standards-positions/#web-bluetooth (accessed may 2023)
[90] “Can I use Bluetooth?”, caniuse.com, https://caniuse.com/?search=bluetooth (accessed May 2023)
[91] “WebBluetoothGetAvailability Feature Popularity Timeline”, Chrome Platform Status, chromestatus.com, https://chromestatus.com/metrics/feature/timeline/popularity/3708 (accessed May 2023)
[92] “WebBluetoothGetDevices Feature Popularity Timeline”,Chrome Platform Status, chromestatus.com, https://chromestatus.com/metrics/feature/timeline/popularity/3328 (accessed May 2023)
[93] “WebBluetoothRequestDevice Feature Popularity Timeline”, Chrome Platform Status, chromestatus.com, https://chromestatus.com/metrics/feature/timeline/popularity/1670 (accessed May 2023)
[94] “Keyboard - Web APIs”, MDN Web Docs, Mozilla.org, https://developer.mozilla.org/en-US/docs/Web/API/Keyboard (accessed Mar. 2023)
[95] “Keyboard Lock”, W3C.org, https://wicg.github.io/keyboard-lock (accessed Mar. 2023)
[96] “Keyboard Map”, W3C.org, https://wicg.github.io/keyboard-map/ (accessed Mar. 2023)
[97] “Can I use Keyboard API?”, caniuse.com, https://caniuse.com/?search=keyboard%20api (accessed Mar. 2023)
[98] “KeyboardApiGetLayoutMap Feature Popularity Timeline”, Chrome Platform Status, https://chromestatus.com/metrics/feature/timeline/popularity/2432 (accessed Mar. 2023)
[99] “KeyboardApiLock Feature Popularity Timeline, Chrome Platform Status, https://chromestatus.com/metrics/feature/timeline/popularity/2394 (accessed Mar. 2023)
[100] “KeyboardApiUnlock Feature Popularity Timeline, Chrome Platform Status, https://chromestatus.com/metrics/feature/timeline/popularity/2395 (accessed Mar. 2023)
[101] “Presentation API”, MDN Web Docs, Mozilla.org, https://developer.mozilla.org/en-US/docs/Web/API/Presentation_API (accessed Apr. 2023)

[102] “Presentation API”, W3C.org, https://w3c.github.io/presentation-api (accessed Apr. 2023)
[103] “Can I use presentation?”, caniuse.com, https://caniuse.com/?search=presentation (accessed Apr. 2023)
[104] “PresentationDefaultRequest Feature Popularity Timeline, Chrome Platform Status, https://chromestatus.com/metrics/feature/timeline/popularity/926 (accessed Apr. 2023)
[105] “PresentationAvailabilityChangeEventListener Feature Popularity Timeline, Chrome Platform Status, https://chromestatus.com/metrics/feature/timeline/popularity/927 (accessed Apr. 2023)
[106] “Web NFC API”, MDN Web Docs, Mozilla.org, https://developer.mozilla.org/en-US/docs/Web/API/Web_NFC_API (accessed May 2023)
[107] “Web NFC”, W3C.org, https://w3c.github.io/web-nfc (accessed May 2023)
[108] “[webkit-dev] WebKit position on Web NFC”, webkit.org, https://lists.webkit.org/pipermail/webkit-dev/2020-January/031007.html (accessed May 2023)
[109] “Web NFC Position”, Mozilla Specification Positions, Mozilla.org, https://mozilla.github.io/standards-positions/#web-nfc (accessed May 2023)
[110] “Can I use ndef?, caniuse.com, https://caniuse.com/?search=ndef (accessed May 2023)
[111] “WebNfcAPI Feature Popularity Timeline”, Chrome Platform Status,https://chromestatus.com/metrics/feature/timeline/popularity/3127 (accessed May 2023)
[112] “WebNfcNdefReaderScan Feature Popularity Timeline”, Chrome Platform Status, https://chromestatus.com/metrics/feature/timeline/popularity/3094 (accessed May 2023)
[113] “WebGPU API” MDN Web Docs, Mozilla.org, https://developer.mozilla.org/en-US/docs/Web/API/WebGPU_API (accessed May 2023)
[114] “WebGPU”, W3C.org, https://www.w3.org/TR/webgpu (accessed May 2023)
[115] “WebGPU Explainer”, W3C.org, https://gpuweb.github.io/gpuweb/explainer (accessed May 2023)
[116] “WeGPU Samples”, W3C.org, https://webgpu.github.io/webgpu-samples (accessed May 2023)
[117] “Get started with GPU Compute on the web”, Chrome Developers Article, Google.com, https://developer.chrome.com/articles/gpu-compute (accessed May 2023)

[118] “Can I use webgpu?”, caniuse.com, https://caniuse.com/?search=webgpu (accessed May 2023)
[119] “WebGPU Feature Popularity Timeline”, Chrome Platform Status, https://chromestatus.com/metrics/feature/timeline/popularity/3888 (accessed May 2023)
[120] “HTMLCanvasElement_WebGPU Feature Popularity Timeline”, Chrome Platform Status, https://chromestatus.com/metrics/feature/timeline/popularity/4029 (accessed May 2023)
[121] “WebHID API”, MDN Web Docs, Mozilla.org, https://developer.mozilla.org/en-US/docs/Web/API/WebHID_API (accessed April 2023)
[122] “WebHID API”, W3C.org, https://wicg.github.io/webhid (accessed April 2023)
[123] “Human interface devices on the web: a few quick examples”, Chrome Developers Article, Google.com, https://developer.chrome.com/articles/hid-examples (accessed April 2023)
[124] “Can I use webhid?”, caniuse.com, https://caniuse.com/?search=webhid (accessed April 2023)
[125] “WebHID (Human Interface Device)”, Chrome Platform Status, chromestatus.com, https://chromestatus.com/feature/5172464636133376 (accessed April 2023)
[126] “HidGetDevices Feature Popularity Timeline”, Chrome Platform Status, https://chromestatus.com/metrics/feature/timeline/popularity/2865 (accessed April 2023)
[127] “HidRequestDevice Feature Popularity Timeline”, Chrome Platform Status, https://chromestatus.com/metrics/feature/timeline/popularity/2866 (accessed April 2023)
[128] “WebUSB API”, MDN Web Docs, Mozilla.org, https://developer.mozilla.org/en-US/docs/Web/API/WebUSB_API (accessed March 2023)
[129] “WebUSB API”, W3C.org, https://wicg.github.io/webusb (accessed March 2023)
[130] “ Access USB Devices on the Web”, Chrome Developers Article, Google.com, https://developer.chrome.com/articles/usb (accessed March 2023)
[131] WebUSB API Position”, Mozilla Specification Positions, Mozilla.org, https://mozilla.github.io/standards-positions/#webusb (accessed March 2023)
[132] “Can I use webusb?”, caniuse.com, https://caniuse.com/?search=webusb (accessed March 2023)
[133] “WebUSB API”, Chrome Platform Status, chromestatus.com, https://chromestatus.com/feature/5651917954875392 (accessed March 2023)

[134] “UsbGetDevices Feature Popularity Timeline”, Chrome Platform Status, https://chromestatus.com/metrics/feature/timeline/popularity/1519 (accessed march 2023)
[135] “UsbRequestDevice Feature Popularity Timeline”, Chrome Platform Status, https://chromestatus.com/metrics/feature/timeline/popularity/1520 (accessed March 2023)
[136] “WebXR Device API”, W3C.org, https://immersive-web.github.io/webxr (accessed February 2023)
[137] “WebXR Device API”, MDN Web Docs, Mozilla.org, https://developer.mozilla.org/en-US/docs/Web/API/WebXR_Device_API (accessed February 2023)
[138] “WebXR Samples”, W3C.org, https://immersive-web.github.io/webxr-samples (accessed February 2023)
[139] “Can I use webxr?”, caniuse.com, https://caniuse.com/?search=webxr (accessed February 2023)
[140] “WebXR Device API”, Chrome Platform Status, chromestatus.com, https://chromestatus.com/feature/5680169905815552 (accessed February 2023)
[141] “NavigatorXR Feature Popularity Timeline”, Chrome Platform Status, https://chromestatus.com/metrics/feature/timeline/popularity/2413 (accessed February 2023)
[142] “News from WWDC23:
WebKit Features in Safari 17 beta”, webkit.org, https://webkit.org/blog/14205/news-from-wwdc23-webkit-features-in-safari-17-beta/ (accessed February 2023)
[143] “Bug 208988 - [WebXR] Implement WebXR device API“, WebKit Bugzilla, https://bugs.webkit.org/show_bug.cgi?id=208988 (accessed February 2023)
[144] “WebXR Device API”, State of JavaScript 2022, https://2022.stateofjs.com/en-US/features/browser-apis/#webxr (accessed February 2023)
[145] “WebXR Device API”, State of JavaScript 2021, https://2021.stateofjs.com/en-US/features/browser-apis/#webxr (accessed February 2023)