From Fugu With Love: New Capabilities for the Web

Thomas Steiner
tomac@google.com
Google Germany GmbH
Hamburg, Germany

Pete LePage
petele@google.com
Google LLC
New York, NY, United States

Thomas Nattestad
Rory McClelland
nattestad@google.com
rorymcclelland@google.com
Google Germany GmbH
Munich, Germany

Alex Russell
slightlyoff@google.com
Google LLC
San Francisco, CA, United States

Dominick Ng
dominickn@google.com
Google Australia
Sydney, NSW, Australia

ABSTRACT

With this demo, we will show at the example of a greeting card web application how new and upcoming browser capabilities can progressively enhance this application so that it remains useful on all modern browsers, but delivers an advanced experience on browsers that support new web capabilities like native file system access, system clipboard access, contacts retrieval, periodic background sync, screen wake lock, web sharing features, and many more.

CCS CONCEPTS

• **Information systems** → **Web applications**; **Browsers**.

KEYWORDS

Progressive Web Apps, Web APIs, Web Incubator Community Group

ACM Reference Format:
Thomas Steiner, Pete LePage, Thomas Nattestad, Rory McClelland, Alex Russell, and Dominick Ng. 2020. From Fugu With Love: New Capabilities for the Web. In *Companion Proceedings of the Web Conference 2020 (WWW '20 Companion), April 20–24, 2020, Taipei, Taiwan.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/XXXXXX.XXXXXX

1 INTRODUCTION AND BACKGROUND

1.1 The App Gap

Modern web browsers provide a solid foundation of capabilities that allow for the creation of impressive web applications. Those capabilities have enabled new experiences on the web that were never thought possible in what initially started as a global information space of linked documents. WebAssembly (Wasm) is enabling new classes of games and productivity apps, Web Real-Time Communication (WebRTC) enables new ways to communicate, and service workers allow developers to create reliably fast web experiences almost regardless of network conditions. However, there are some capabilities, like file system access, raw clipboard access, background app refresh, and more, that are available to native platforms like Android or iOS, but that are not available to the web platform. These missing capabilities mean some types of apps cannot be delivered on the web, or that these apps are less useful. In colloquial terms, this is sometimes referred to as the *app gap*.

1.2 Using Web Technologies, But Not the Web

To cope, some developers only build native apps—that is, do not build for the web in the first place. Alternatively, they use wrappers like Apache Cordova[1] on mobile or Electron[2] on desktop to access the underlying capabilities of the device, while still building *with* the web technologies HTML, CSS, and JS. This is not without risk from a security point of view [2, 11]. Users also pay more in download size and disk space, as runtimes must be distributed with application code in this model. We argue, instead, that web developers should have access to the capabilities they need to create great web experiences and we want to support them by collaborating in the *Capabilities Project*, code-named *Project Fugu*.

1.3 The Capabilities Project, Or Project Fugu

In the context of the Project Fugu, we want to enable web apps to do anything native apps can, by exposing the capabilities of native platforms to the web platform, while maintaining user security, privacy, trust, and other core tenets of the web. Giving developers these new tools will empower the open web as a place where almost any experience can be created, and make it a first class platform for developing apps that run on any browser, with any operating system, and on any device. We design and develop these new capabilities in an open and transparent way in the W3C's Web Incubator Community Group[3] (WICG) using the existing open web platform standards processes while getting early feedback from developers and other browser vendors as we iterate on the design of these features to ensure its interoperability.

Project Fugu is a truly cross-company effort, with contributors from Google, Intel, and Microsoft. All monthly meeting notes are publicly accessible to anyone in the Chromium organization,[4] and the regularly scheduled conference calls are open for anyone to join upon invitation.

WWW '20 Companion, April 20–24, 2020, Taipei, Taiwan

ACM ISBN 978-1-4503-7024-0/20/04.
https://doi.org/10.1145/XXXXXX.XXXXXX

[1] Apache Cordova: https://cordova.apache.org/
[2] Electron: https://electronjs.org/
[3] WICG: https://wicg.io/
[4] Project Fugu notes: https://bit.ly/fugu-sync

We have identified and prioritized an initial set of capabilities we heard partner demand for and that we see as critical to closing the gap between web and native. People interested in the list can review it by searching the Chromium bug database for bugs that are tagged with the label `proj-fugu`.[5] Regarding the project's code name: fugu is a pufferfish that is considered a delicacy, however, if not carefully prepared, it can be lethally poisonous. This analogy works quite well with many of the capabilities we deal with. We have outlined our vision for a more capable web in a blog post [10] on the Chromium blog and invite interested parties to follow along the project's progress on the dedicated project landing page.[6]

1.4 The Capabilities Process

We have developed a process to make it possible to design and develop new web platform capabilities that meet the needs of developers quickly, in the open, and most importantly, without moving feature development outside the standards process. The capabilities process, depicted in Figure 1, consists of the following steps:

(1) **Identify the developer need:**
The first step is to identify and understand the developer need. How are they doing it today? And what and whose problems or frustrations are fixed by this new capability? Typically, these come in as feature request from developers, frequently via bugs filed on bugs.chromium.org.
(2) **Create an explainer:**
After identifying the need for a new capability, create an explainer, essentially a design document that is meant to explain the problem, along with sample code showing how the API might work. The explainer is a living document that will go through heavy iteration as the new capability evolves.
(3) **Get feedback and iterate on the explainer:**
Once the explainer has a reasonable level of clarity, it is time to publicize it, to solicit feedback, and iterate on the design. This is an opportunity to verify the new capability meets the needs of developers and works as expected and to gather public support and verify that there really is a need for this.
(4) **Move the design to a specification and iterate:**
Once the explainer is in a good state, the design work transitions into a formal specification, working with developers and other browser vendors to iterate and improve on the design. Once the design starts to stabilize, we typically use an origin trial[7] to experiment with the implementation. Origin trials allow developers to try new features with real users, and give feedback on the implementation.
(5) **Ship it:**
Finally, once the origin trial is complete, the spec has been finalized, and all of the other launch steps have been completed, it is time to ship it to stable.

It is worth noting that many ideas never make it past an explainer or origin trial stage. *Not* shipping a feature because it does not solve the developer need is fine. We want to highlight that this process does *not* replace the Blink launch process;[8] both go hand in hand.

[5]Project Fugu bugs: https://bit.ly/fugu-bugs
[6]Project Fugu landing page: https://developers.google.com/web/updates/capabilities
[7]Origin Trial: http://googlechrome.github.io/OriginTrials/explainer.html
[8]Blink launch process: https://www.chromium.org/blink/launching-features

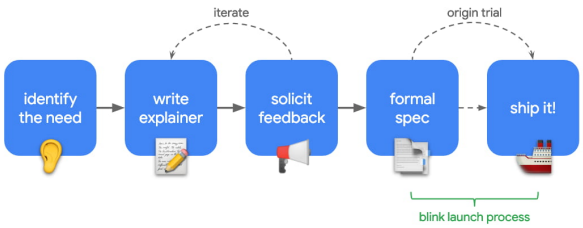

Figure 1: The Capabilities Process

1.5 Permissions

Some of the capabilities we work on are potentially harmful for users if not handled appropriately (we recall the code name of this project). In a position paper [13] presented at the W3C Permissions Workshop, we have outlined our standpoints regarding evolving the current permission model. We propose the following steps:

- Permission-requesting APIs need to introduce the ability for developers to register interest in a capability before ever being allowed to prompt users and to be called back when the situation changes, *e.g.*, if increased site engagement is noted, when before the threshold would not have been reached.
- Since different web APIs currently have disparate ways to signal a developer's intent to use them, permissions requests should be centralized on a single method [14] to enable better user controls and bring much needed developer consistency.
- The Permissions API [9] should be extended with a revoke method, so developers are able to ensure their applications operate with least-privilege.
- The Web Application Manifest should be extended to include fields which allow sites to identify to the runtime a maximum set of permissions. Requests for permissions not included in this list should fail.

1.6 Access to Powerful Web Platform Features

Allowing users to control which sites are able to access powerful APIs is crucial for maintaining the security and privacy properties of the web. The impact of restrictions on the developer ergonomics and user utility of the API and the web platform overall must also be considered. The following general principles [12] summarize the overall approach of the Chromium project to evaluating how powerful new features should be controlled on the web:

- Access to powerful APIs should be available to the entire web of secure contexts, with control managed exclusively by user consent UX like prompts or pickers at time-of-use.
- API-specific restrictions on the scope of access may also be used to guard against potential abuse.
- Usage of powerful APIs should be clearly disclosed to users, ideally using a central hub that offers users control over what sites can use which capabilities.
- Installing a web app is associated with persistence, and thus persistent and/or background access to powerful APIs may only be granted (possibly subject to additional requirements) to installed web apps. Non-installed sites may still request and be granted permission to use powerful APIs, but should not have their access persisted.

- Installation or engagement alone should not act as a vote of trust for either granting access or enabling the ability to ask for access to powerful APIs. Separately, efforts should be made to curtail the existing persistency on the web platform outside of installed web apps, *e.g.*, time-limiting permission grants, more aggressively expiring cookies, and restricting background task execution.

2 PRIOR ART

This is not the first attempt at making the web a powerful application platform. We list three examples that went to market with this promise. While there are more,[9] these three are very representative.

2.1 Chrome Apps

In 2013, Chrome Apps [7] promised to bring together the speed, security, and flexibility of the modern web with the powerful functionality previously only available with software installed on devices. Chrome Apps were designed to work offline, were capable of running in stand-alone windows, could be launched directly from the desktop, supported desktop notifications, and could interact with USB, Bluetooth and other devices connected to a desktop, including digital cameras. They were kept updated automatically and synced their state to the cloud. Chrome Apps could use a set of proprietary `chrome.*` APIs.[10] After three years, Chrome Apps were deprecated in August 2016.[11] Project Fugu is a direct successor to Chrome Apps. In fact, Progressive Web Apps built using Fugu APIs are one of the recommended migration strategies.[12]

2.2 Palm webOS

Palm webOS[13] was invented exclusively for mobile use. It recognized that users wanted their people, calendars, and information to move with them, wherever they were, wirelessly, as opposed to being bound to a personal computer. At its core, webOS leverages industry-standard technologies, including web technologies such as CSS, XHTML, and JS. It changed owners several times and lives on as an open-source project[14] powering smart devices, *e.g.*, TVs.

2.3 Firefox OS

Firefox OS, also known as Boot 2 Gecko[15] and first commercially released in 2013, is a discontinued open-source operating system designed by Mozilla and external contributors. It was based on the rendering engine of the Firefox web browser, Gecko, and on the Linux kernel. The operating system was capable of running web applications directly or those installed from an app marketplace. The applications used standards like JS and HTML5, and web APIs that could communicate directly with the underlying hardware. A fork of Firefox OS now gains traction under the name of KaiOS,[16] especially on feature phones in emerging markets.

[9] Others: Adobe AIR, Windows Hosted Apps, Blackberry WebWorks apps, W3C Widgets
[10] `chrome.*` APIs: https://developer.chrome.com/apps/api_index
[11] Chrome Apps deprecation: http://bit.ly/chrome-apps-deprecation
[12] Transitioning from Chrome Apps: https://developers.chrome.com/apps/migration
[13] Palm webOS announcement: http://bit.ly/palm-webos-announcement
[14] webOS open-source project: https://www.webosose.org/
[15] Boot to Gecko: https://developer.mozilla.org/en-US/docs/Archive/B2G_OS
[16] KaiOS: https://www.kaiostech.com/

3 DEMO DESCRIPTION

3.1 Core Contributions and Intended Audience

In March 2003, Nick Finck and Steve Champeon stunned the web design world with the concept of progressive enhancement [3], a strategy for web design that emphasizes core webpage content first, and that then progressively adds more nuanced and technically rigorous layers of presentation and features on top of the content. While in 2003, progressive enhancement was about using at the time modern CSS features, unobtrusive JavaScript, and even Scalable Vector Graphics, progressive enhancement in 2020 is about using modern browser capabilities.

We will show at the example of a greeting card demo web application how new and upcoming browser capabilities can progressively enhance this application so that it remains useful on all modern browsers, but delivers an advanced experience on browsers that support capabilities like native file system access, system clipboard access, contacts retrieval, periodic background sync, screen wake lock, sharing features, and many more.

The target audience are academia and industry web developers, browser engineers, and people involved in the W3C standards process, as well as web privacy researchers.

3.2 Demo Link and Demo Requirements

The demo is publicly available at https://glitch.com/~fugu-greetings, where it can either be explored directly, or where its source code can be inspected and cloned. We recommend running the demo in the latest Chrome Canary[17] or Edge Canary[18] browsers (both at version 82 as of February 2020). Since this demo is about bleeding edge web APIs, some browser runtime flags need to be set (below, replace `chrome://` with `edge://` on Edge):

- chrome://flags/#native-file-system-api
- chrome://flags/#enable-experimental-web-platform-features
- chrome://flags/#periodic-background-sync

3.3 Baseline Application

Progressive Web Applications (PWAs) are a type of application software delivered through the web, built using common web technologies including HTML, CSS, and JS. They are intended to work on any platform that uses a standards-compliant browser. As such, we will start with a simple drawing web application that serves as the baseline greeting card app that is offline-enabled, can be added to the user's home screen, *etc.*, and step-by-step add new browser capabilities as progressive enhancements that are dynamically offered on supporting browsers. We note that the currently rudimentary visual design of this application is *not* focus of the demo.

3.4 Web Share (Target) API Support

Creating an advanced greeting cards app is nonsensical if there is no one out to appreciate the cards. We thus add a feature that allows the user to share their drawings with the world. The Web Share API [4] allows for the sharing of files using the native device's share mechanism. Figure 2 shows the user initiating a share of a drawing on an Android device. Both the native Google Hangouts app as

[17] Chrome Canary: https://www.google.com/chrome/canary/
[18] Edge Canary: https://www.microsoftedgeinsider.com/en-us/download

well as the native Facebook app offer itself as share targets. The other way round, through the Web Share Target API, the greeting card application itself can become a share target that one can share images to as card sources, for example, from the photo gallery app.

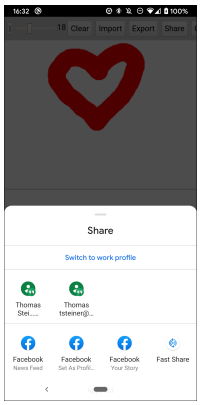

Figure 2: Web Share API

3.5 Native File System API Support

Drawing everything from scratch is hard. We thus add a feature that allows the user to import a local image into the application. Figure 3 shows the file import dialog where, after granting access, the user can open a local image file and add it to their drawing. Later, they can also save their creation to disk. Both operations are enabled through the Native File System Access API [8] and not to be confused with legacy workarounds that required uploading a local file to a server and downloading a copy.

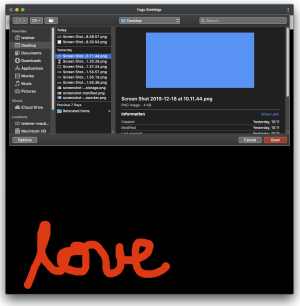

Figure 3: Native File System Access API

3.6 Contact Access API Support

At times it can be hard to correctly type a greeting card recipient's name, for example, when it is written in a different script that the present keyboard layout does not support. We add a feature that allows users to pick one (or multiple) of their local contacts and add their names to the greeting card message. This is a one-off operation facilitated by the Contact Picker API [1]; no continuous contacts access is granted and the user can choose the contact details.

3.7 Clipboard API Support

Occasionally users might want to paste a picture from another app into the greeting card app, or copy a drawing from the greeting card app into another app. We add a feature that allows users to copy and paste images from and to the app. The Clipboard API [6] allows for the asynchronous copying and pasting of image data (currently limited to Portable Network Graphics images).

3.8 Badging API Support

If the greeting cards app is installed on a user's device, it will have an icon on their home screen. This icon can be used to convey fun information on the icon badge like the number of brushstrokes a given drawing has taken. The Badging API [5] enables apps to set a numeric badge on the app icon that increments with each stroke.

3.9 Other APIs Support

The demo will showcase a number of other APIs, namely the Shape Detection API to detect shapes like faces, the Wake Lock API to keep the screen awake while the user waits for drawing inspiration, the Periodic Background Sync API to surprise the user with a new greeting card template each day, the Idle Detection API to clear the greeting card when the user is no longer interacting with the application (for example, when it is running in a kiosk setup), and the File Handling API, which allows the app to register as a file handler to integrate with the operating system's file explorer.

4 CONCLUSIONS

We hope to gather feedback on these proposed features from the conference attendees, as well as trigger conversations about future challenges around permissions, security, and browser compatibility, and invite interested parties to learn more about Project Fugu.

REFERENCES

[1] Peter Beverloo and Rayan Kanso. 2019. *Contact Picker API.* Unofficial Proposal Draft, 4 November 2019. WICG. https://wicg.github.io/contact-api/spec/.
[2] Luca Carettoni. 2017. *Electron Security Checklist. A guide for developers and auditors.* Technical Report. Doyensec LLC. https://doyensec.com/resources/us-17-Carettoni-Electronegativity-A-Study-Of-Electron-Security-wp.pdf.
[3] Steve Champeon. 2003. *Progressive Enhancement and the Future of Web Design.* Technical Report. http://hesketh.com/publications/progressive_enhancement_and_the_future_of_web_design.html.
[4] Matt Giuca. 2017. *Web Share API.* Draft Community Group Report, 30 November 2017. w3c. https://wicg.github.io/web-share/.
[5] Matt Giuca and Jay Harris. 2019. *Badging API.* Draft Community Group Report, 11 November 2019. WICG. https://wicg.github.io/badging/.
[6] Gary Kacmarcik and Grisha Lyukshin. 2019. *Clipboard API and events.* Editor's Draft, 22 August 2019. w3c. https://w3c.github.io/clipboard-apis/.
[7] Erik Kay. 2013. *A new breed of Chrome Apps.* Technical Report. Google. https://chrome.googleblog.com/2013/09/a-new-breed-of-chrome-apps.html.
[8] Marijn Kruisselbrink. 2019. *Native File System.* Draft Community Group Report, 18 November 2019. WICG. https://wicg.github.io/native-file-system/.
[9] Mounir Lamouri, Marcos Cáceres, and Jeffrey Yasskin. 2019. *Permissions.* Editor's Draft, 30 October 2019. w3c. https://w3c.github.io/permissions/.
[10] Pete LePage. 2018. *Our commitment to a more capable web.* Technical Report. https://blog.chromium.org/2018/11/our-commitment-to-more-capable-web.html.
[11] Tongbo Luo, Hao Hao, Wenliang Du, Yifei Wang, and Heng Yin. 2011. Attacks on WebView in the Android system. In *Proceedings of the 27th Annual Computer Security Applications Conference.* ACM, 343–352.
[12] Dominick Ng and Rory McClelland. 2019. *Controlling Access to Powerful Web Platform Features.* Technical Report. https://goo.gle/access-to-powerful-features.
[13] Alex Russell and Thomas Nattestad. 2018. *Permissions Workshop Position Paper.* Technical Report. https://goo.gle/permissions-workshop-position-paper.
[14] Jeffrey Yasskin. 2017. *Requesting Permissions.* Draft Community Group Report, 28 September 2017. WICG. https://wicg.github.io/permissions-request/.