From Fugu With Love: New Capabilities for the Web

Anonymous Author(s)

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 a truly stunning experience on browsers that support new web capabilities like native file system access, system clipboard access, contacts retrieval, periodic background sync, screen wake lock, sharing features, and many more.

CCS CONCEPTS

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

KEYWORDS

Progressive Web Apps, Web APIs, Web Incubator Community Group

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 on 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—or 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], and also suboptimal for users who, apart from the application code, on top need to download the wrapper code—for each application separately. We argue that instead 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 capabilities project, 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 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 to ensure its interoperability.

The capabilities project is a truly cross-company effort, with contributors from Google, Intel, and Microsoft. All monthly meeting notes are publicly available,[4] and the regularly scheduled conference calls are open for anyone to join.

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, and have already started work on a handful of them. 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] Fugu is a pufferfish that is considered a delicacy, however, if not carefully prepared, can be lethally poisonous. This analogy works quite well with many of the capabilities we deal with, hence the code name *project fugu*. 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 Fugu 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, work within the existing standards process. Developer feedback is critical to

---

[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
[5]Project Fugu bugs: https://bit.ly/fugu-bugs
[6]Project Fugu landing page: https://developers.google.com/web/updates/capabilities

help us ensure we are shipping the right features, but it is easier to change course early in the process. 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;[7] both go hand in hand together. The fugu 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. What is the developer trying to accomplish? 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 in a way that they expect. This is also an opportunity to gather public support and verify that there really is a need for this capability.

(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. Then, once the design starts to stabilize, we typically use an origin trial[8] to experiment with the implementation. Origin trials allow developers to try new features with real users, and give feedback on the implementation. This real world feedback helps shape and validate the design, ensuring we get it right, before it becomes a standard.

(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.
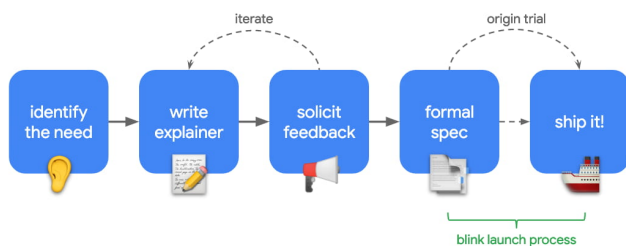

Figure 1: The Capabilities Process

[7]Blink launch process: https://www.chromium.org/blink/launching-features
[8]Origin Trial: http://googlechrome.github.io/OriginTrials/explainer.html

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 thus outlined our standpoints regarding evolving the current permission model. Briefly, we propose the following steps to be taken:

- 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 in their favor.
- 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 the introduction of 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 platform of secure contexts, with control managed exclusively by choosers, prompts, or other user consent UX 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. Below, we list three platforms that went to market with this promise. While there are more, these three can serve as representative examples of the field.

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, could connect to the cloud, 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 the state of apps to all desktop devices the user signed in to. Chrome Apps could use a set of proprietary chrome.* apis.[9] After three years, Chrome Apps were deprecated in August 2013[10] The capabilities project is a direct successor to Chrome Apps, in fact, Progressive Web Apps built using fugu apis are one of the recommended migration strategies.[11]

2.2 Palm webos

Palm webos[12] 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[13] powering smart devices, *e.g.*, tvs.

2.3 Firefox os

Firefox os, also known as Boot 2 Gecko,[14] 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 application marketplace. The applications used standards like JavaScript 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,[15] especially on feature phones in emerging markets.

3 DEMO DESCRIPTION

3.1 Core Contributions and Intended Audience

Back 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.

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 a truly stunning 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 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.

3.3 Web Share (Target) api Support

Creating the most amazing greeting cards is boring if there is no one out to appreciate them. 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 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, for example, from the native photo gallery app.

3.4 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.

3.5 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

---

[9]chrome.* apis: https://developer.chrome.com/apps/api_index
[10]Chrome Apps deprecation update note: https://blog.chromium.org/2016/08/from-chrome-apps-to-web.html
[11]Transitioning from Chrome Apps: https://developers.chrome.com/apps/migration
[12]Palm webos announcement: https://web.archive.org/web/20090117111808/http://investor.palm.com/releasedetail.cfm?ReleaseID=358392
[13]webos open-source project: https://www.webosose.org/
[14]Boot 2 Gecko: https://developer.mozilla.org/en-US/docs/Archive/B2G_OS
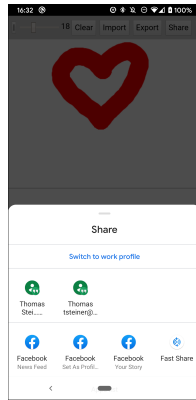[15]Kaios: https://www.kaiostech.com/
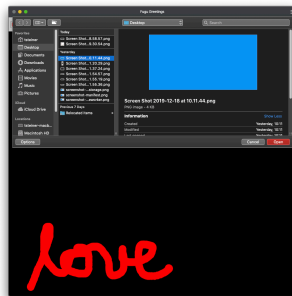
Figure 2: Web Share API

Figure 3: Native File System Access API

add their names to the greeting card message. This is a one-off operation facilitated by the Contact Picker API [1], no continuous access to the contacts is granted.

3.6 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).

3.7 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.

3.8 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 public kiosk setup), and ultimately the File Handling API, which allows the greeting card app to register as a file handler so image files can be opened with the app directly via 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.* Technical Report. 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.
[5] Matt Giuca and Jay Harris. 2019. *Badging API. Draft Community Group Report 11 November 2019.* Technical Report. WICG. https://wicg.github.io/badging/.
[6] Gary Kacmarcik and Grisha Lyukshin. 2019. *Clipboard API and events. Editor's Draft, 22 August 2019.* Technical Report. 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.* Technical Report. WICG. url://wicg.github.io/native-file-system/.
[9] Mounir Lamouri, Marcos Cáceres, and Jeffrey Yasskin. 2019. *Permissions. Editor's Draft, 30 October 2019.* Technical Report. 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://chromium.googlesource.com/chromium/src/+/lkgr/docs/security/permissions-for-powerful-web-platform-features.md.
[13] Alex Russell and Thomas Nattestad. 2018. *Permissions Workshop Position Paper.* Technical Report. https://www.w3.org/Privacy/permissions-ws-2018/papers/thomas-nattestad.pdf.
[14] Jeffrey Yasskin. 2017. *Requesting Permissions. Draft Community Group Report, 28 September 2017.* Technical Report. WICG. https://wicg.github.io/permissions-request/.

A 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[16] or Edge Canary[17] browsers. 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

---

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