What is in a Web View? An Analysis of Progressive Web App Features When the Means of Web Access is not a Web Browser

Thomas Steiner
Google Germany GmbH
20354 Hamburg, Germany
tomac@google.com

ABSTRACT

Progressive Web Apps (PWA) are a new class of Web applications, enabled for the most part by the Service Workers APIs. Service Workers allow apps to *work offline* by intercepting network requests to deliver programmatic or cached responses, Service Workers can receive *push notifications* and *synchronize* data in the background even when the app is not running, and—together with Web App Manifests—allow users to *install PWAs* to their devices' home screens. Service Workers being a Web standard, support has landed in several stand-alone Web browsers—among them (but not limited to) Chrome and its open-source foundation Chromium, Firefox, Edge, Opera, UC Browser, Samsung Internet, as well as preview versions of Safari. In this paper, we examine the PWA feature support situation in *Web Views*, that is, *in-app Web experiences* that are explicitly *not* stand-alone browsers. Such in-app browsers can commonly be encountered in chat applications like WeChat or WhatsApp, online social networks like Facebook or Twitter, but also email clients like Gmail, or simply anywhere where Web content is displayed inside native apps. We have developed an open-source application called *PWA Feature Detector* that allows for easily testing in-app browsers (and naturally stand-alone browsers as well) and have evaluated the level of support for PWA features on different devices and Web Views. On the one hand, our results show that there are big differences between the various Web View technologies and the browser engines they are based upon, but on the other hand, that the results are independent from the devices' operating systems, which is good news given the problematic update policy of many device manufacturers. These findings help developers make educated choices when it comes to determining whether a PWA is the right approach given their target users' means of Web access.

KEYWORDS

Progressive Web Apps, Service Workers, Web Views

1 INTRODUCTION

In recent years, there has been a paradigm shift from browser to native apps and back to browser again. The Web currently is undergoing a silent revolution with Web apps, more descriptively *Progressive Web Apps*, or for short just PWAs. How did we get there?

WOODSTOCK'97, July 1997, El Paso, Texas USA

ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.475/123_4

1.1 History of Progressive Web Apps

Since around 2005, Web development has moved from static multi-page *documents* to single-page *applications*, heavily enabled by the `XMLHttpRequest` API, a process that eventually led Garrett to coin the term *Ajax* (Asynchronous JavaScript and XML [19]) to describe this shift. Despite an early push for Web-based apps on devices such as the 2007 iPhone, attempts at Web apps mostly failed by comparison to native apps that are distributed through app stores rather than the Web. Native apps not only had direct hardware access to, *e.g.*, camera and microphone, to various sensors like accelerometer or geolocation, but also just in general provided a better user experience and booted faster, compared to having to load in a browser at runtime. Additionally, advanced offline support and push notifications were simply unthinkable for Web applications of the epoch, and Web app icons that already could be added to devices' home screens were mostly just bookmarks with—apart from full screen mode—no special behavior. While straightforward offline scenarios could be realized with `AppCache` [35], more complex offline scenarios were error-prone and hard to get right [7].

As the Web platform matured and more and more hardware-related APIs were implemented in browsers, in the end it was the addition of Service Workers [32] to the Chromium browser in 2014 [16] that started to unlock a new class of Web apps that finally could *work offline*, receive *push notifications* and *synchronize* data in the background even when the app was not running, and—together with Web App Manifests [12]—allowed users to actually *install PWAs* to their devices' home screens with proper operating system integration [25]. Other browsers like Mozilla Firefox, Microsoft Edge, Opera, UC Browser, Samsung Internet, Apple Safari Technology Preview, and several browsers more followed in implementing Service Workers. Now, even multinational companies like Twitter or trivago bet on PWA [18, 38], as well as giant national players like Tencent News or Sina Weibo in China [40]. Figure 1 shows the PWA of Flipkart, a shopping site popular in India, running in the Google Chrome browser on Android.

1.2 Research Question and Paper Structure

In this paper, we look at a special means for accessing PWAs, namely accessing them explicitly *not* through stand-alone browsers like the ones listed above, but through *in-app browsers* that render Web content in the context of native applications. Examples of such applications with in-app browsers are chat apps like WeChat (Wēixìn) or WhatsApp, online social networks like Facebook or Twitter, but also email clients like Gmail, or simply anywhere where Web content is displayed inside native apps. The technology that these applications leverage internally are so-called *Web Views*. In order to understand why this presents an interesting research problem, one

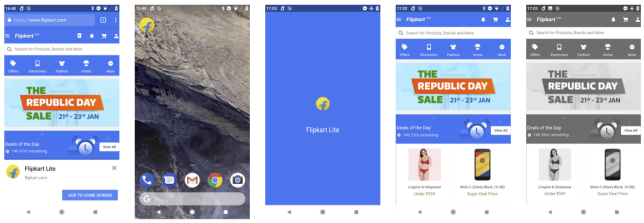

Figure 1: Screenshots showing some PWA features in action at the example of Flipkart (https://www.flipkart.com): *(i)* add to home screen prompt, *(ii)* icon on the home screen *(iii)* splash screen while launching, *(iv)* launched in full screen without URL bar, and *(v)* signaling offline state.

needs to first understand the role that applications like WeChat and thus Web Views play in markets like China. Chan writes in an article [14] for the venture capital firm Andreessen Horowitz: "Millions (note, not just thousands) of lightweight apps live inside WeChat, much like webpages live on the internet. *This makes WeChat more like a browser for mobile websites*, or, arguably, a mobile operating system—complete with its own proprietary app store. The lightweight apps on WeChat are called 'official accounts'. Approved by WeChat after a brief application process, there are well over 10 million of these official accounts on the platform—ranging from celebrities, banks, media outlets, and fashion brands to hospitals, drug stores, car manufacturers, internet startups, personal blogs, and more". Chan goes on: "WeChat focuses on taking care of the plumbing—overseeing the integration of such pre-existing services into its portal—by simply linking users from the wallet menu to webpages from within the app. It's yet another way in which *WeChat becomes an integrated browser for the mobile (and web) world*". It is to be noted that this development comes to the detriment of the so-called Open Web. As Yang and Yang write in the Financial Times [39]: "[WeChat's] news feed and search tools pull content only from within WeChat's walls rather than from the open web, including updates posted by individual users called moments, corporate accounts and an immense collection of WeChat accounts which are used by newspapers and independent bloggers". While personally we do not embrace this development and are advocates of the open Web, we nevertheless examine the implications of an in-app closed Web experience and its impact on PWAs.

In the remainder of this paper, we first look at the technical background of Web Views on both Android and iOS in section 2. We then describe the examined PWA features and their underlying APIs in section 3, in which we also introduce our application *PWA Feature Detector*. In continuation, we present and discuss our results in section 4. We close the paper with an outlook on future work, and draw some conclusions in section 5.

2 BACKGROUND ON WEB VIEWS

There are several different ways to integrate Web content in native applications, each having their own benefits and drawbacks. In the following, we describe the options on the two popular mobile operating system Android and iOS. At time of writing (January 2018), Safari on iOS does not support Service Workers and thus PWAs yet, albeit work on implementing them has started, and first versions of Safari Technology Preview on Apple's desktop operating system macOS have enabled Service Workers by default [17]. In the spirit of Progressive Enhancement [13], where new features will just be used when they are available, and in the hope of Service Workers landing on the iOS platform in the not too distant future, we nevertheless describe the Web View situation on iOS as well.

2.1 Web Views on Android

Android Web Views with `WebView`. In the Android operating system, a `WebView` [2] is a subclass of a `View` that displays Web pages. This class is the basis upon which developers can create their own Web browser or simply display some online content in their apps. It does not include any features of a fully developed Web browser, such as navigation controls or an address bar. All that `WebView` does, by default, is show a Web page. Therefore, it uses the system browser's rendering engine to display Web pages and includes methods to navigate forward and backward through a history, zoom in and out, perform text searches, inject custom JavaScript, and more. Looper describes [28] the development of the component as follows: "Whereas earlier versions of the Android OS relied on the WebKit rendering engine to power its `WebView`, as of Android 4.4, various versions of Chromium are implemented. Typically, with each consecutive update of Android's OS, a new version of Chromium would also be included, thereby giving access to the new rendering engine's capability. This causes issues in backward compatibility for developers who must support earlier versions of Android. To combat this particular problem, as of Android 5.0, the concept of the auto-updating `WebView` has been introduced. Instead of the `WebView` version and capabilities depending on Android OS' update cycle, the Android 5.0 `WebView` is a system-level `.apk` file available in Google Play that can update itself in the background".

Chrome Custom Tabs with `CustomTabsIntent`. While `WebViews` are completely isolated from the user's regular browsing activities, Chrome Custom Tabs [23], available since Chrome 45 (September 2015) and instantiatable as `CustomTabsIntent`, provide a way for an application to customize and interact with a Chrome Activity on Android. This makes the Web content feel like being a part of the application, while retaining the full functionality and performance of a complete Web browser through a shared cookie jar and permissions model, so users do not have to log in to sites they are already connected to, or re-grant permissions they have already granted.

Trusted Web Activity with `TrustedWebUtils`. Chrome Custom Tabs solved many issues of Android `WebViews`, however, had the drawback of not being available in a fullscreen variant without any browser user interface elements like `WebViews` were. As of October 2017, Trusted Web Activities [22] are a new way to integrate Web app content such as PWAs with Android apps. They can be instantiated with the `TrustedWebUtils` and use a communication protocol based on Chrome Custom Tabs. Content in a Trusted Web Activity is trusted—the app and the site it opens are expected to come from the same developer, this is verified using Digital Asset Links.[1] The host app does not have direct access to Web content in a Trusted Web Activity or any other kind of Web state. Transitions between

[1]Digital Asset Links: https://developers.google.com/digital-asset-links/

Web and native content are between activities. Each activity (*i.e.*, screen) of an app is either completely provided by the Web, or by an Android activity. While not enforced at time of writing, Trusted Web Activities will ultimately need to meet content requirements similar to the "improved add to home screen" flow [25], which is designed to be a baseline of interactivity and performance.

2.2 Web Views on ios

ios Web Views with `UIWebView`. Similar to Android, on ios as well Web content could be embedded with a simple system-level Web View called `UIWebView` [5]. With the release of ios 4.3 in early 2011, Apple introduced Nitro, a faster, just-in-time (JIT) JavaScript engine for Safari that considerably sped up the browser's performance in loading complex Web pages. Nitro was exclusive to Safari: third-party developers could not benefit from the faster performance in their Web Views based on `UIWebView`, which was widely considered a calculated move to encourage usage of Safari over Web Views and Web apps saved to the iPhone's home screen [36].

ios Web Views with `WKWebView`. In June 2014, Apple announced `WKWebView` [6], a new API that would allow developers to display Web content in custom Web Views with the same performance benefits of Safari. Designed with security in mind, `WKWebView` featured the same Nitro engine of Safari, while still allowing developers to customize the experience with their own user interface and features. Due to Apple's App Store restrictions, third-party browsers on ios internally need to depend on `WKWebView`, documented, *e.g.*, for Edge for ios [29] or Chrome for ios [15].

ios Web Views with `SFSafariViewController`. In September 2015 with the release of ios 9, Apple introduced a new Web View called `SFSafariViewController` [4], which enables apps to delegate the responsibility of showing Web content to Safari itself, avoiding the need to write custom code for built-in browsers. Up until ios 10, `SFSafariViewController` shared cookies and website data with Safari, which means that if a user was already logged in to a specific website in Safari and a link to that website was opened in `SFSafariViewController`, the user was already logged in. As of ios 11, cookie and website data is no longer shared automatically, but developers can on an as-needed basis leverage an `SFAuthenticationSession` [3] that shares data upon user consent.

2.3 Parallelisms on the Two Operating Systems

The development on the two operating systems has clear parallelisms that can be summarized as follows. From the initially slow and gradually improved simple Web Views `WebView` (with the transparent internal switch from WebKit to Chromium) on Android and `UIWebView` and `WKWebView` on ios, there was an evolution to more powerful and better integrated browser tab experiences, namely `CustomTabsIntent` on Android and `SFSafariViewController` on ios, which both (only upon user consent since ios 11) share cookies, permissions, *etc.* with the particular system's main browser. Solely Android's Trusted Web Activity so far has no ios equivalent yet. If (or rather as soon as) Apple catches up with implementing Service Workers, our findings will be applicable on ios as well, according to the parallelisms described in this section.

3 DETECTING PWA FEATURES

What exactly makes a Web app a *Progressive* Web App is not clearly defined. One of the most open definitions comes from Samsung [33], maker of the Samsung Internet browser: "Progressive Web Apps (PWAs) are regular mobile and desktop web applications that are accessible in any web browser. In browsers that support new open web standards [...] they *can* provide additional capabilities including offline support and push notifications" (emphasis ours). However, just like with Ajax [19], the term PWA became a catch-all umbrella brand for Web apps that in some way or the other use Service Worker APIs, feel (native) "app-like," use latest browser features if they are available (Progressive Enhancement [13]), or that can be installed to the home screen. Russell [31] lists a number of requirements for what he calls "baseline appyness": "A Progressive Web App is functionally defined by the technical properties that allow the browser to detect that the site meets certain criteria and is worthy of being added to the homescreen. [...] Apps on the homescreen:

- Should load instantly, regardless of network state. [T]hey [don't] need to function fully offline, but they must put their own UI on screen without requiring a network round trip.
- Should be tied in the user's mind to where they came from. The brand or site behind the app shouldn't be a mystery.
- Can run without extra browser chrome (*e.g.*, the URL bar). [...] To prevent hijacking by captive portals (and worse), apps must be loaded over TLS connections."

In continuation, Russell translates these requirements into more technical terms, writing that PWAs must (emphasis ours):

- "Originate from a Secure Origin. Served over TLS and green padlock displays (no active mixed content).
- Load while offline (even if only a custom offline page). By implication, this means that *Progressive Web Apps require Service Workers.*
- Reference a Web App Manifest [...]"

In consequence, we consider a "PWA feature" any feature that requires one or more of the Service Worker APIs. Additionally, *iff* (if and only if) the Web View implements Service Workers, we further consider additional recent browser APIs, detailed in the following.

3.1 Detecting Service Worker Support

A `ServiceWorker` is installed by calling the `register` method on the `navigator` object, whose first parameter is obligatory and contains a URL that points to a JavaScript file with the Service Worker code. The result of this promise-based API in the success case is then a `ServiceWorkerRegistration` object, which is either newly created if there was no previous `ServiceWorker`, or updated in the alternative case where a previous `ServiceWorker` existed [32]. In order to detect if a given Web View supports PWA features at all, we can thus make a simple existence check for the API, and then try to register a Service Worker, as outlined in Listing 1.

3.2 Considered Progressive Web App Features

Offline Capabilities The ability to still load and work at least to some extent, even when the device is offline [32], for example, when airplane mode is on or when the device temporarily has no network coverage.

Push Notifications The capability to display push notifications as defined in the Push API [11], for example, to point users to fresh content, even when the app is not running.

Add to Home Screen The capability to be installed (added) to a device's home screen for easy access as outlined in [25].

Background Sync The capability to synchronize data in the background [32], for example, to send messages in a deferred way after a temporary offline situation in a chat app.

Navigation Preload The capability to start network navigation requests even while the Service Worker has not booted yet [8], which would else be a blocking operation.

Silent Push The capability to use the Web Budget API [10] to determine if potentially expensive operations like background refresh may be started upon a silent push notification.

Storage Estimation The capability to estimate the available storage that an application already uses and to know the available quota enforced by the browser [34].

Persistent Storage The capability to persistently store data that is guaranteed not to be purged by the browser without user consent, even if memory is running out [34].

Web Share The capability to invoke the native sharing widget of the operating system, as defined in the Web Share API [20].

Media Session The capability to show customized media metadata on the platform user interface, customize available platform media controls, and access platform media keys found in notification areas and on lock screens of mobile devices as defined in the Media Session standard [27].

Media Capabilities The ability to make an optimal decision when picking media content for the user by exposing information about the decoding and encoding capabilities for a given format, but also output capabilities to find the best match based on the device's display as defined in the Media Capabilities standard [26].

Device Memory The capability to read the amount of available Random Access Memory (RAM) in Gigabyte of a device in order to allow servers to customize the app experience based on the built-in memory [30].

Getting Installed Related Apps The capability to detect if a corresponding native application is installed alongside the PWA in order to, for example, not show push notifications twice on both apps [24].

Payment Request The capability to act as intermediary among merchants, users, and payment methods by means of a standardized payment communication flow that supports different secure payment methods [9].

Credential Management The capability to request a user's credentials from the browser, and to help the browser correctly store credentials for future use to facilitate logins [37].

3.3 Feature-Detecting Various PWA Features

A core principle of Progressive Enhancement [13] is *feature detection*. The idea behind feature detection is to run a test to determine whether a certain feature is supported in the current browser, and then conditionally run code to provide an acceptable experience both in browsers that do support the feature, and browsers that do not. It is distinct from *browser sniffing*, where based on the user agent string assumptions are being made regarding feature support, which is generally considered problematic and bad practice [1]. Listing 2 shows the feature detection tests we run in order to detect the features listed in the previous subsection. As outlined before, a `ServiceWorkerRegistration`, *i.e.*, an active Service Worker is a prerequisite for all tests. The variables `nav` for `navigator`, `win` for `window`, and `doc` for `document` purely serve for code minification.

```
// This commonly should happen after `window.onload`
// has fired in order to prioritize content display
if ('serviceWorker' in navigator) {
  navigator.serviceWorker.register(scriptURL, options)
  .then(registration => {
    console.log(registration);
  })
  .catch(error => {
    console.log(error);
  });
} else {
  console.log('Service Workers not supported');
}
```

Listing 1: Checking for Service Worker support.

```
// nav ==> navigator
// win ==> window
// doc ==> document
// reg ==> ServiceWorkerRegistration
const detectFeatures = (reg) => {
  return {
    'Offline Capabilities': 'caches' in win,
    'Push Notifications': 'pushManager' in reg,
    'Add to Home Screen': doc.createElement('link')
        .relList.supports('manifest'),
    'Background Sync': 'sync' in reg,
    'Navigation Preload': 'navigationPreload' in reg,
    'Silent Push': 'budget' in nav &&
        'reserve' in nav.budget,
    'Storage Estimation': 'storage' in nav &&
        'estimate' in nav.storage,
    'Persistent Storage': 'storage' in nav &&
        'persist' in nav.storage,
    'Web Share': 'share' in nav,
    'Media Session': 'mediaSession' in nav,
    'Media Capabilities': 'mediaCapabilities' in nav,
    'Device Memory': 'deviceMemory' in nav,
    'Getting Installed Related Apps':
        'getInstalledRelatedApps' in nav,
    'Payment Request': 'PaymentRequest' in win,
    'Credential Management': 'credentials' in nav,
  };
};
```

Listing 2: Feature detection of various PWA features.

3.4 Implementation Details

We have developed an open-source application called *PWA Feature Detector* that allows for easily testing in-app browsers (and naturally stand-alone browsers as well) and check for the available PWA features. The code of the application can be found at https://github.com/tomayac/pwa-feature-detector, the app itself is deployed at https://tomayac.github.io/pwa-feature-detector/. When the `window.load` event fires, it tries to register a no-op Service Worker that—in the success case—immediately claims its clients in order to obtain a `ServiceWorkerRegistration`, which is required for the then executed feature detection tests in Listing 2. Finally, it

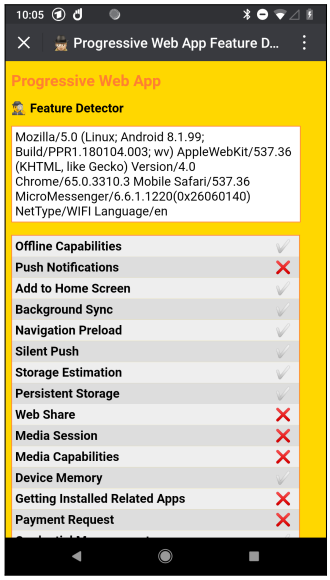

Figure 2: PWA Feature Detector running on Android 8.1.99 in the app WeChat in a `WebView` based on Chrome 65.

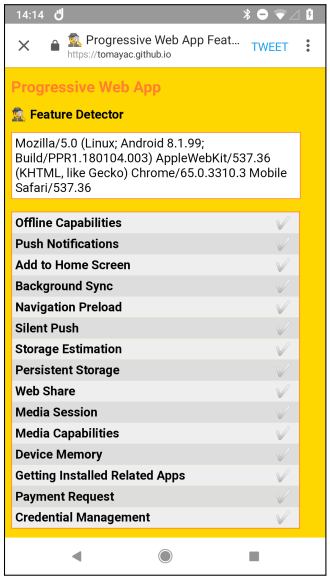

Figure 3: PWA Feature Detector running on Android 8.1.99 in the app Twitter in a `CustomTabsIntent` based on Chrome 65.

displays the results in tabular form, and also prints the browser's user agent string. Figure 2 shows a screenshot of PWA Feature Detector running on Android 8.1.99 in the chat app WeChat in a `WebView` based on Chrome 65. In contrast, Figure 3 shows a screenshot of PWA Feature Detector running on the *same device*, but this time in the social networking app Twitter, which, rather than using a `WebView`, displays Web content in a `CustomTabsIntent`. Despite the exact same underlying browser engine (Chrome/65.0.3310.3), the more powerful `CustomTabsIntent`-based in-app browser clearly wins the PWA feature competition. For non-domain experts we note that the `WebView` can be easily spotted by looking for the string "wv" in the displayed user agent [21].

4 RESULTS AND DISCUSSION

We have run PWA Feature Detector on a representative range of devices with different Android operating system versions, browser engines, and several applications with in-app browsers of users in China and Germany. We covered everything from Android 6 ("Marshmallow"), to Android 7 ("Nougat"), to the (at time of writing) most up-to-date Android 8 ("Oreo"). For the browser engines, we had devices running Chrome 53, 55, 57, 59, 61, 63, and 65. The popularity of WeChat [14] in China clearly was reflected in the applications with in-app browsers that we covered. We had WeChat (that identifies itself as "MicroMessenger" in user agent strings), Sina Weibo, Facebook, Facebook Messenger, and Twitter. Table 1 shows our results for in-app browsers based on `WebView`, Table 2 shows the results for `CustomTabsIntent`. Figure 3 shows screenshots of all tested devices. The results in both tables are primarily ordered by browser engine version and secondarily ordered by Android version. As can be seen, the Android version has no impact on the set of supported PWA features, which makes sense given the technical background information in section 2 regarding the decoupling of operating system version and `WebView` version.

4.1 Discussion of the `WebView` Results

Table 1 unsurprisingly shows that the more mature the underlying browser engine gets, the more PWA features become available. We can see that the PWA features which—according to the feature detection tests from Listing 2—were reported to be working from the earliest examined browser engine on are *Offline Capabilities*, *Background Sync*, *Credential Management*, and *Add to Home Screen*. However, we need to have a closer look at the results.

4.1.1 *PWA Features Reported to Be Working.*

Offline Capabilities Supported from the start, the feature *Offline Capabilities* is working as expected.

Credential Management The feature *Credential Management* is presumably erroneously exposed, which is tracked in Chromium bug https://crbug.com/589829.

Add to Home Screen While in theory supported from the start, the situation with *Add to Home Screen* is blurry. First, the criteria for *when exactly* the prompt actually triggers are not exposed publicly. The conditions listed in [25] are necessary, but not sufficient. What the feature test does is check if the browser knows how to deal with a Web App Manifest, however, if it does is left to browser implementers [12].

Background Sync While in theory supported from the start, *Background Sync* exists, but fails upon trying to use it. This is tracked in Chromium bug https://crbug.com/570713.

Persistent Storage Chrome 55 added support for *Persistent Storage* [34], but while the `navigator.storage.persist` method exists and can be called, the result is always negative, so data is actually never persisted.

Feature	Examples																		
Offline Capabilities	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Credential Management	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Add to Home Screen	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Background Sync	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Persistent Storage	X	X	X	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Navigation Preload	X	X	X	X	X	X	X	X	X	X	✓	✓	✓	✓	✓	✓	✓	✓	✓
Silent Push	X	X	X	X	X	X	X	X	X	X	X	X	✓	✓	✓	✓	✓	✓	✓
Storage Estimation	X	X	X	X	X	X	X	X	X	X	X	X	✓	✓	✓	✓	✓	✓	✓
Device Memory	X	X	X	X	X	X	X	X	X	X	X	X	X	✓	✓	✓	✓	✓	✓
Media Capabilities	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Getting Installed Related Apps	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Web Share	X	X	X	X	X	X	X	X	X	X	X	X	✓	X	X	X	X	X	X
Media Session	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Payment Request	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Push Notifications	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Android Version	7.0	7.1.1	7.1.2	7.0	6.0.1	7.0	7.0	7.1.1	7.1.1	7.0	7.0	8.1.0	7.0	8.1.0	8.1.0	8.1.0	8.1.99	8.1.99	8.1.99
Browser Engine	Chrome 53	Chrome 53	Chrome 53	Chrome 55	Chrome 57	Chrome 57	Chrome 57	Chrome 57	Chrome 57	Chrome 59	Chrome 59	Chrome 61	Chrome 63	Chrome 63	Chrome 63	Chrome 63	Chrome 65	Chrome 65	Chrome 65
Application	MicroMessenger 6.6.1	MicroMessenger 6.6.1	MicroMessenger 6.6.1	Weibo 7.12.0	MicroMessenger 6.6.1	MicroMessenger 6.6.1	MicroMessenger 6.6.1	MicroMessenger 6.6.1	MicroMessenger 6.6.1	Weibo 8.0.2	Weibo 8.0.2	Facebook 154.0.0.33.385	Weibo 8.0.2	MicroMessenger 6.6.1	Facebook 154.0.0.33.385	Facebook 148.0.0.20.381	MicroMessenger 6.6.1	Facebook 155.0.0.22.96	WebView

Table 1: Increasingly improving PWA feature support situation on various Android `WebViews`, ordered by browser engine and Android version. The sole *seemingly* supported Web Share feature in Chrome 61 was actually a bug (https://crbug.com/765923).

Feature	Examples			
Offline Capabilities	✓	✓	✓	✓
Credential Management	✓	✓	✓	✓
Add to Home Screen	✓	✓	✓	✓
Background Sync	✓	✓	✓	✓
Persistent Storage	✓	✓	✓	✓
Navigation Preload	✓	✓	✓	✓
Silent Push	✓	✓	✓	✓
Storage Estimation	✓	✓	✓	✓
Device Memory	X	✓	✓	✓
Media Capabilities	X	X	✓	✓
Getting Installed Related Apps	X	✓	✓	✓
Web Share	✓	✓	✓	✓
Media Session	✓	✓	✓	✓
Payment Request	✓	✓	✓	✓
Push Notifications	✓	✓	✓	✓
Android Version	8.1.0	8.1.0	8.1.99	8.1.99
Browser Engine	Chrome 61	Chrome 63	Chrome 65	Chrome 65
Application	Twitter 7.28.0	Twitter 7.28.0	Twitter 7.27.0	Chrome Custom Tab

Table 2: Increasingly improving PWA feature support situation on various Android `CustomTabsIntents`, ordered by browser engine and Android version.

Navigation Preload The feature *Navigation Preload*, added in Chrome 59 is working as expected.

Silent Push *Silent Push*, added in Chrome 61 has a method `navigator.budget.reserve` that returns nothing, where a boolean value is expected. Like *Push Notifications* it is not supposed to work.

Storage Estimation The feature *Storage Estimation*, added in Chrome 61, is working as expected.

Device Memory The feature *Device Memory*, added in Chrome 63, is working as intended.

Web Share The *Web Share* feature, introduced in Chrome 61, was for this one version of Chrome reported to be working, but fixed to be no longer exposed in the browser via Chromium bug https://crbug.com/765923.

4.1.2 PWA Features Reported Not to Be Working.

Push Notifications The feature *Push Notifications* is supposed and confirmed not to be working.

Payment Request The feature *Payment Request* is supposed and confirmed not to be working.

Media Session The feature *Media Session* is confirmed not to be working, it might be enabled in the future, though, as discussed in Chromium bug https://crbug.com/678979.

Media Capabilities The feature *Media Capabilities* is confirmed not to be working, it might be enabled in the future, though, as discussed in Chromium bug https://crbug.com/690364.

Getting Installed Related Apps The feature *Getting Installed Related Apps* is supposed and confirmed not to be working.

4.2 Discussion of the `CustomTabsIntent` Results

The results in Table 2 present no big surprises. We can see *Device Memory* support and *Getting Installed Related Apps* support be added in Chrome 63, and *Media Capabilities* in Chrome 65. Everything else is confirmed to be working as intended, as `CustomTabsIntent` is as close to the system browser as it gets, as outlined in section 2.

4.3 Why No Results for `TrustedWebUtils`

The results are exactly the same as with `CustomTabsIntent`, as `TrustedWebUtils` internally just launches a `CustomTabsIntent` with a special flag `EXTRA_LAUNCH_AS_TRUSTED_WEB_ACTIVITY`.

5 FUTURE WORK AND CONCLUSIONS

Future work will cover two aspects: we look at it from the point of view of app developers programming PWAs for Web Views, as well as from the angle of a browser vendor.

5.1 Improving Feature Tests

First, we will look into improving the feature detection tests in Listing 2. Such tests always need to be side-effect-free, so in the majority of cases naively trying to execute an exposed method rather than just testing for its existence is prohibitive. We have seen that no-op or dummy methods like with *Silent Push*, where

navigator.budget.reserve just did nothing, or with *Persistent Storage*, where the method navigator.storage.persist always returned a negative response can be the reason for unexpected results. In an ideal world, existence tests should just be good enough, which requires catching involuntarily exposed interfaces. Chromium bug https://crbug.com/787868 has as an objective to automatically identify such cases.

5.2 Polyfilling Missing APIs in WebViews

While generally CustomTabsIntent is the Web View of choice, there may be reasons where app developers are stuck with WebView. A *polyfill* is code that implements a feature on Web browsers that do not support the feature. The WebView.evaluateJavascript method allows JavaScript code to be asynchronously evaluated in the context of the currently displayed page. For app developers, this can help improve the PWA support situation for some APIs that can, at least to some extent, be polyfilled. One example might be *Push Notifications*, albeit polyfills like notification.js[2] can acknowledgedly only partially work due to technical constraints.

5.3 Conclusions

In this paper, we have first provided the technical background on Web Views on both Android and iOS. Second, we have defined a number of PWA features and documented and implemented tests for detecting them in the open source app PWA Feature Detector. In continuation, we have evaluated the approach on a great variety of devices with different versions of operating systems, Web Views, and applications with in-app browser. We have identified a number of issues with these feature tests, and have collected various interfaces that are exposed erroneously in browsers. The conclusion is that we need to find out why not more developers use the superior CustomTabsIntent. One theory is a perceived fear of the Open Web, *e.g.*, when through open APIs like *Web Share* users could too easily escape the walled garden of a social network.

ACKNOWLEDGEMENTS

We would like to thank the participants of the Google Developer Days 2017 in Shanghai, China, who with their phones helped gather the statistics for this paper and sent us the screenshots on WeChat. Further, we would like to thank Adam Bar of https://whatwebcando.today fame for the helpful discussions around feature detection.

REFERENCES

[1] Aaron Andersen. 2008. History of the browser user-agent string. https://webaim.org/blog/user-agent-string-history/. (2008).
[2] Android Developers. 2018. Building Web Apps in WebView. https://developer.android.com/guide/webapps/webview.html. (2018).
[3] Apple Developer Documentation. 2018. SFSafariAuthenticationSession. https://developer.apple.com/documentation/safariservices/sfauthenticationsession. (2018).
[4] Apple Developer Documentation. 2018. SFSafariViewController. https://developer.apple.com/documentation/safariservices/sfsafariviewcontroller. (2018).
[5] Apple Developer Documentation. 2018. UIWebView. https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIWebView_Class/. (2018).
[6] Apple Developer Documentation. 2018. WKWebView. https://developer.apple.com/library/ios/documentation/WebKit/Reference/WKWebView_Ref/. (2018).
[7] Jake Archibald. 2012. Application Cache is a Douchebag. http://alistapart.com/article/application-cache-is-a-douchebag. (2012).
[8] Jake Archibald. 2017. Speed up Service Worker with Navigation Preloads. https://developers.google.com/web/updates/2017/02/navigation-preload. (2017).
[9] Adrian Bateman, Zach Koch, Roy McElmurry, Domenic Denicola, and Marcos Cáceres. 2017. *Payment Request API.* W3C Candidate Recommendation 21 September 2017. W3C.
[10] Peter Beverloo. 2017. *Web Budget API.* WICG Editor's Draft, 24 May 2017. WICG.
[11] Peter Beverloo, Martin Thomson, Michaël van Ouwerkerk, Bryan Sullivan, and Eduardo Fullea. 2017. *Push API.* W3C Editor's Draft 15 December 2017. W3C.
[12] Marcos Cáceres, Kenneth Rohde Christiansen, Mounir Lamouri, Anssi Kostiainen, and Rob Dolin. 2017. *Web App Manifest—Living Document.* W3C Working Draft 29 November 2017. W3C.
[13] Steve Champeon. 2003. Progressive Enhancement and the Future of Web Design. http://hesketh.com/publications/progressive_enhancement_and_the_future_of_web_design.html. (2003).
[14] Connie Chan. 2015. When One App Rules Them All: The Case of WeChat and Mobile in China. https://a16z.com/2015/08/06/wechat-china-mobile-first/. (2015).
[15] Chromium Blog. 2016. A faster, more stable Chrome on iOS. https://blog.chromium.org/2016/01/a-faster-more-stable-chrome-on-ios.html. (2016).
[16] Dominic Cooney and Joshua Bell. 2014. Chrome 40 Beta: Powerful Offline and Lightspeed Loading with Service Workers. https://blog.chromium.org/2014/12/chrome-40-beta-powerful-offline-and.html. (2014).
[17] Jon Davis. 2017. Release Notes for Safari Technology Preview 46. https://webkit.org/blog/8042/release-notes-for-safari-technology-preview-46/. (2017).
[18] Nicolas Gallagher. 2017. How we built Twitter Lite. https://blog.twitter.com/engineering/en_us/topics/open-source/2017/how-we-built-twitter-lite.html. (2017).
[19] Jesse James Garrett. 2005. Ajax: A New Approach to Web Applications. http://adaptivepath.org/ideas/ajax-new-approach-web-applications/. (2005).
[20] Matt Giuca. 2017. *Web Share API.* Draft Community Group Report 30 November 2017. W3C.
[21] Google Chrome. 2018. User Agent Strings. https://developer.chrome.com/multidevice/user-agent. (2018).
[22] Google Developers. 2017. Using Trusted Web Activity. https://developers.google.com/web/updates/2017/10/using-twa. (2017).
[23] Paul Kinlan. 2016. Chrome Custom Tabs. https://developer.chrome.com/multidevice/android/customtabs. (2016).
[24] Paul Kinlan. 2017. Detect if your Native app is installed from your web site. https://developers.google.com/web/updates/2017/04/getinstalledrelatedapps. (2017).
[25] Paul Kinlan. 2017. The New and Improved Add to Home Screen. https://developers.google.com/web/updates/2017/02/improved-add-to-home-screen. (2017).
[26] Mounir Lamouri. 2017. *Media Capabilities.* Draft Community Group Report, 12 December 2017. WICG.
[27] Mounir Lamouri. 2017. *Media Session Standard.* Editor's Draft, 17 August 2017. WICG.
[28] Jen Looper. 2015. What is a WebView? https://developer.telerik.com/featured/what-is-a-webview/. (2015).
[29] Sean Lyndersay. 2017. Microsoft Edge for iOS and Android: What developers need to know. https://blogs.windows.com/msedgedev/2017/10/05/microsoft-edge-ios-android-developer/. (2017).
[30] Shubhie Panicker. 2017. *Device Memory 1.* Editor's Draft, 11 December 2017. WICG.
[31] Alex Russell. 2016. What, Exactly, Makes Something A Progressive Web App? https://infrequently.org/2016/09/what-exactly-makes-something-a-progressive-web-app/. (2016).
[32] Alex Russell, Jungkee Song, Jake Archibald, and Marijn Kruisselbrink. 2017. *Service Workers 1.* Editor's Draft, 22 December 2017. W3C.
[33] Samsung. 2017. Progressive Web Apps. https://samsunginter.net/docs/progressive-web-apps. (2017).
[34] Anne van Kesteren. 2018. *Storage.* Living Standard—Last Updated 9 January 2018. WHATWG.
[35] Anne van Kesteren and Ian Hickson. 2008. *Offline Web Applications.* W3C Working Group Note 30 May 2008. W3C.
[36] Federico Viticci. 2015. iOS 9 and Safari View Controller: The Future of Web Views. https://www.macstories.net/stories/ios-9-and-safari-view-controller-the-future-of-web-views/. (2015).
[37] Mike West. 2017. *Credential Management Level 1.* W3C Working Draft, 4 August 2017. W3C.
[38] Think with Google. 2017. The Next Billion Users: trivago Embrace Progressive Web Apps as the Future of Mobile. https://www.thinkwithgoogle.com/intl/en-gb/consumer-insights/trivago-embrace-progressive-web-apps-as-the-future-of-mobile/. (2017).
[39] Yuan Yang and Yingzhi Yang. 2017. Tencent pushes into news feed and search in challenge to Baidu. https://www.ft.com/content/59ca05e8-3ba6-11e7-821a-6027b8a20f23. (2017).
[40] Shunhao Zhu and Michael Yeung. 2017. PWA and AMP ♡ China (GDD China '17). https://www.youtube.com/watch?v=JCTjQx56-NY. (2017).

[2] https://adodson.com/notification.js/

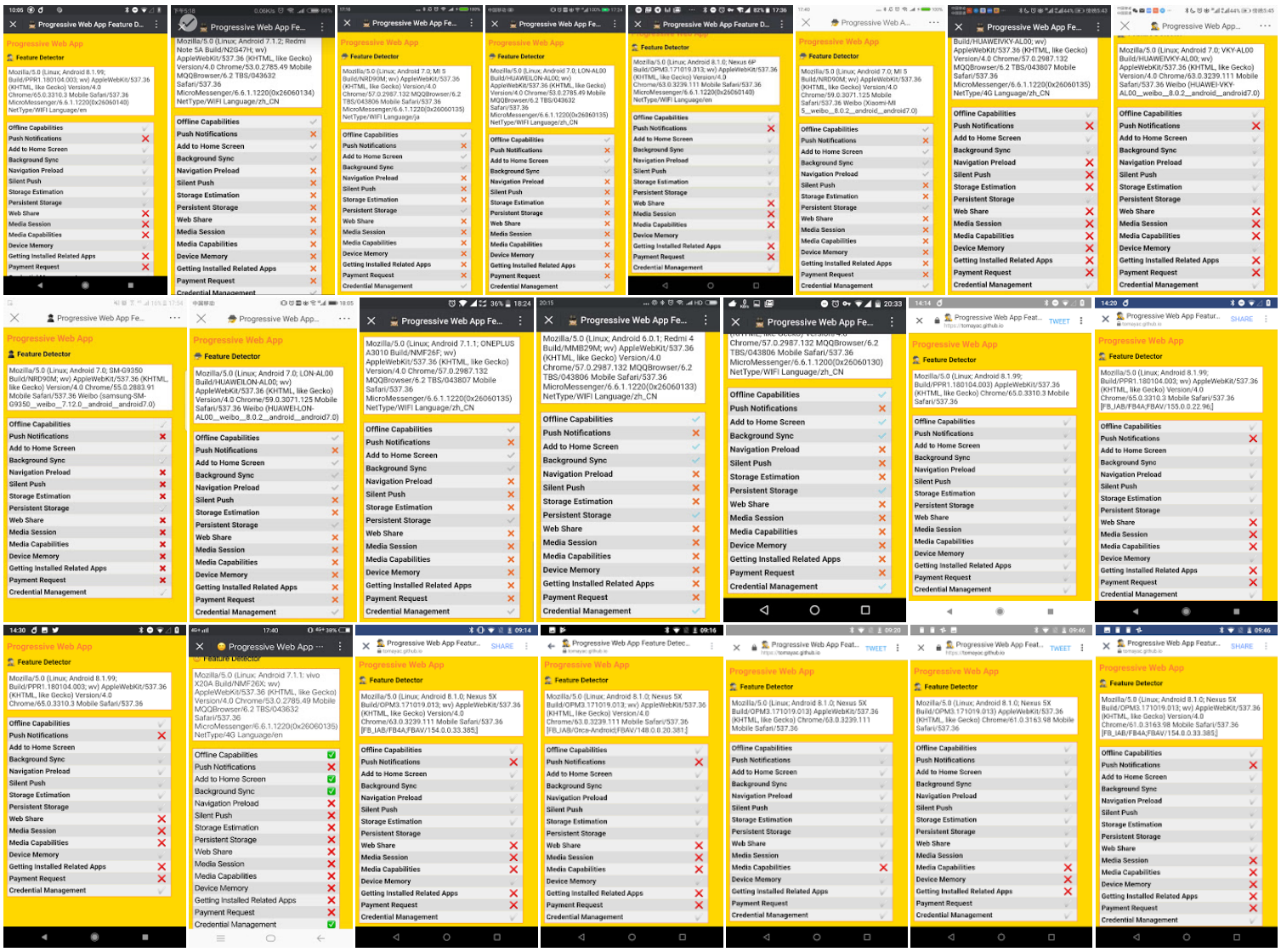

Figure 3: Index sheet with all tested device screenshots showing both `WebViews` and `CustomTabsIntents` running in different applications. High-resolution screenshots available at https://photos.app.goo.gl/FdRsGOgtgNfvRIKB2.