Getting Started with Progressive Web Apps

By Addy Osmani
Eng Manager, Web Developer Relations

There's been much welcome discussion about Progressive Web Apps lately. They're still a relatively new model, but their principles can equally enhance apps built with vanilla JS, React, Polymer, Angular or any other framework. In this post, I'll summarize some options and reference apps for getting started with your own progressive web app today.

What is a Progressive Web App?

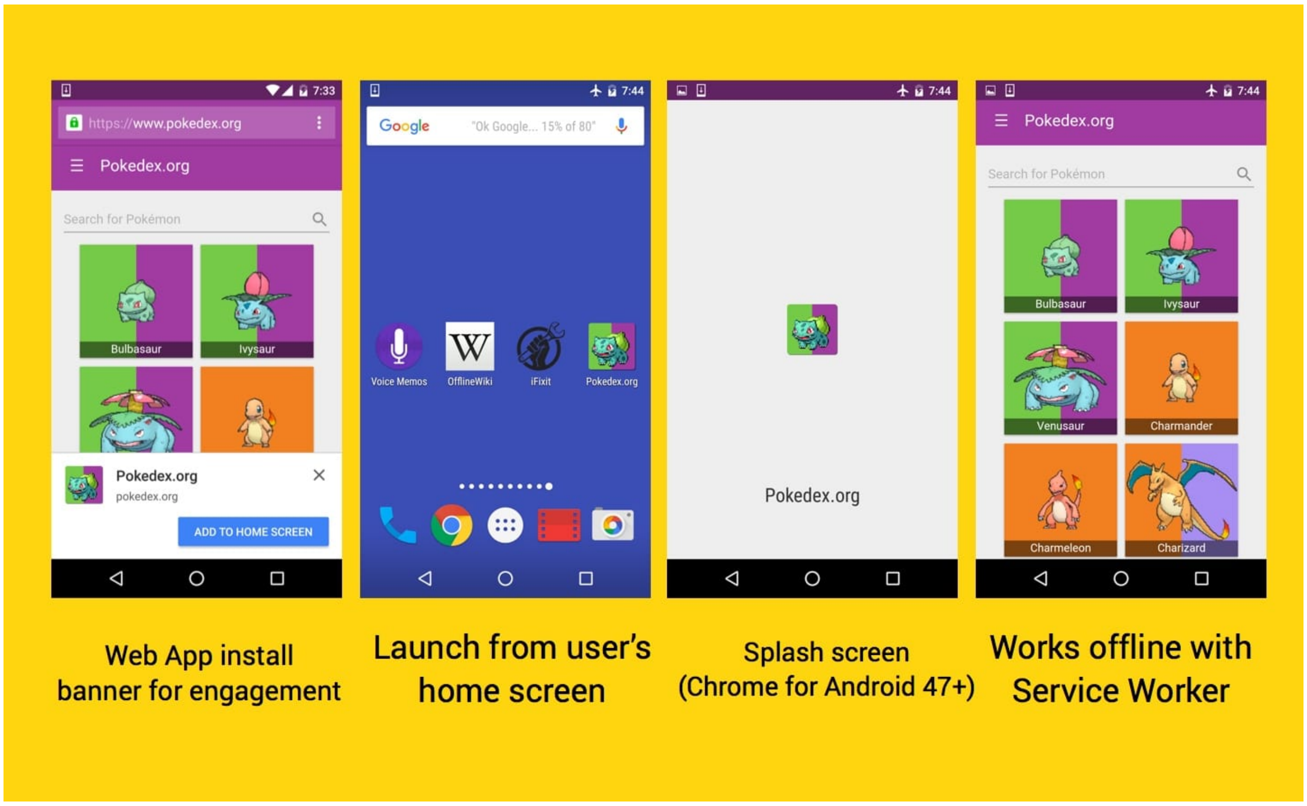
> Progressive Web Apps use modern web capabilities to deliver an app-like user experience. They evolve from pages in browser tabs to immersive, top-level apps, maintaining the web's low friction at every moment.

It's important to remember that Progressive Web Apps work everywhere but are supercharged in modern browsers. Progressive enhancement is a backbone of the model.

Aaron Gustafson likened progressive enhancement to a peanut M&M. The peanut is your content, the chocolate coating is your presentation layer and your JavaScript is the hard candy shell. This layer can vary in color and the experience can vary depending on the capabilities of the browser using it.

Think of the candy shell as where many Progressive Web App features can live. They are experiences that combine the best of the web and the best of apps. They are useful to users from the very first visit in a browser tab, no install required.

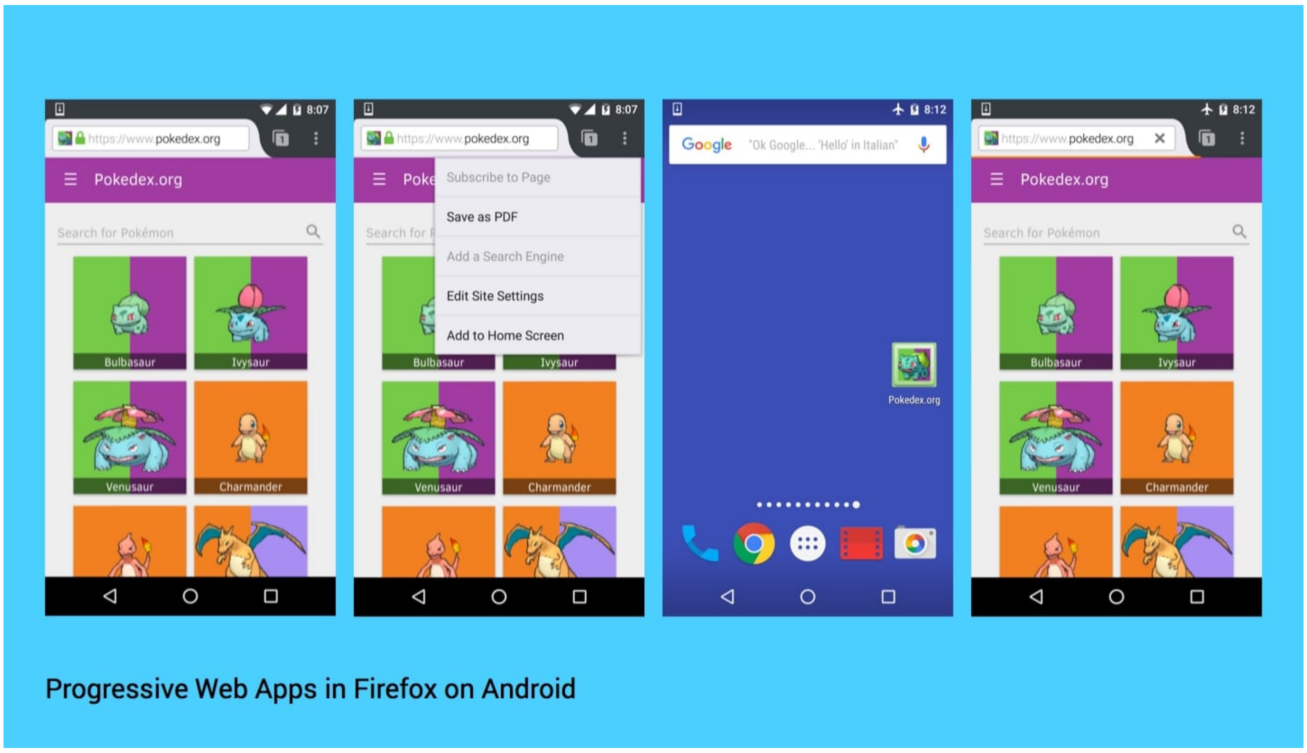
As the user builds a relationship with these apps through repeat use, they make the candy shell even sweeter - loading very fast on slow network connections (thanks to service worker), sending relevant Push Notifications and having a first-class icon on the user's home screen that can load them as fullscreen app experiences. They can also take advantage of smart web app install banners.



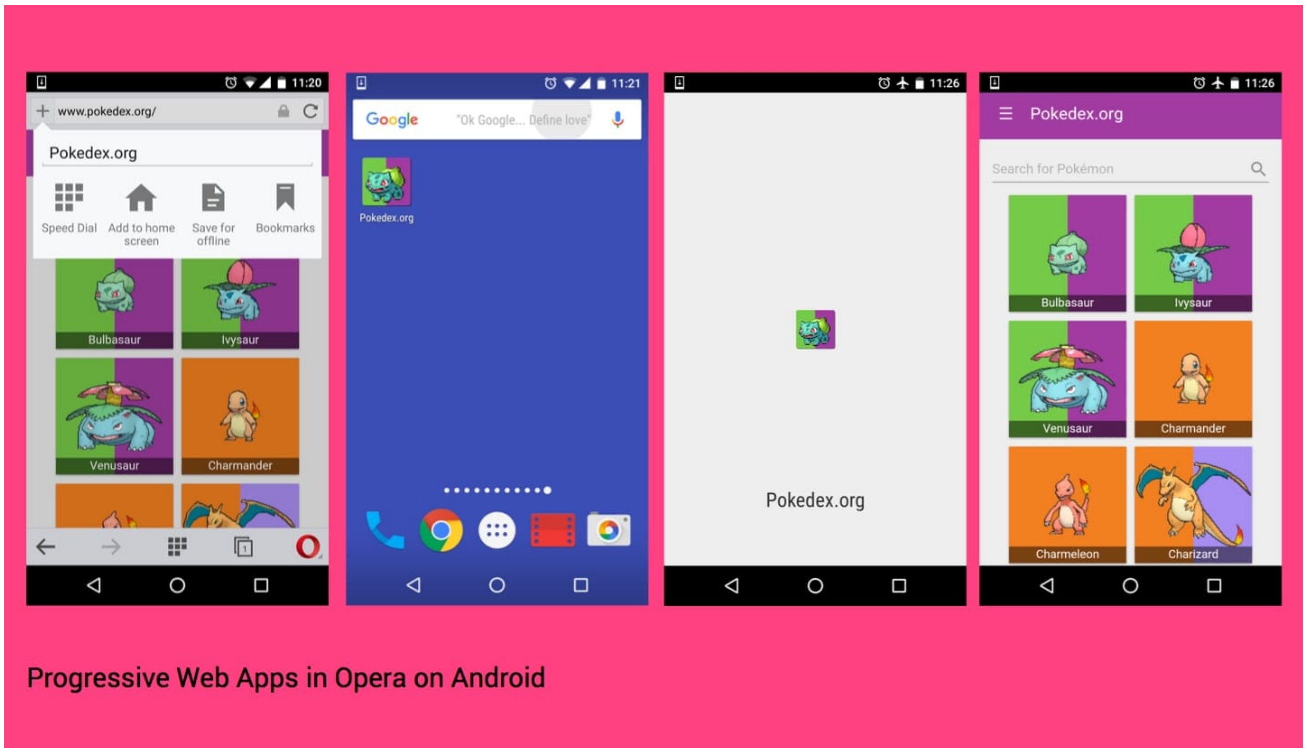
Progressive Web Apps are:

- **Progressive** - Work for every user, regardless of browser choice because they're built with progressive enhancement as a core tenant.
- **Responsive** - Fit any form factor, desktop, mobile, tablet, or whatever is next.
- **Connectivity independent** - Enhanced with service workers to work offline or on low quality networks.
- **App-like** - Use the app shell model to provide app-style navigations and interactions.
- **Fresh** - Always up-to-date thanks to the service worker update process.
- **Safe** - Served via TLS to prevent snooping and ensure content hasn't been tampered with.
- **Discoverable** - Are identifiable as "applications" thanks to W3C manifests and service worker registration scope allowing search engines to find them.
- **Re-engageable** - Make re-engagement easy through features like push notifications.
- **Installable** - Allow users to "keep" apps they find most useful on their home screen without the hassle of an app store.
- **Linkable** - Easily share via URL and not require complex installation.

Progressive Web Apps also aren't unique to Chrome for Android. Below we can see the Pokedex Progressive Web App working in Firefox for Android (Beta) with early Add to home screen and service worker caching features running just fine.



One of the nice aspects of the "progressive" nature to this model is that features can be gradually unlocked as browser vendors ship better support for them. Progressive Web Apps such as Pokedex also of course work great in Opera on Android too with a few notable differences in implementation:



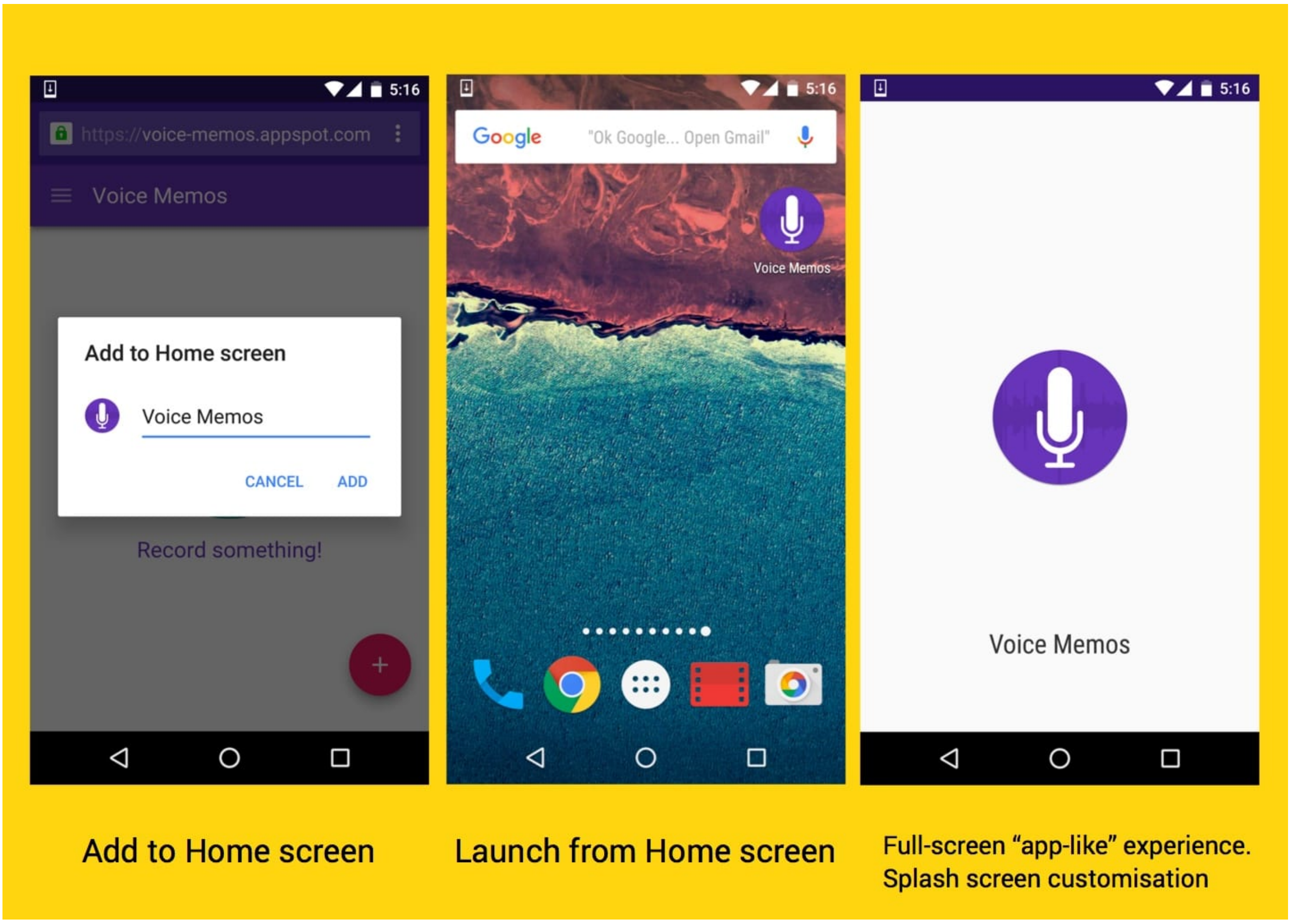
For diving deeper into Progressive Web Apps, read Alex Russell's original blog post introducing them. Paul Kinlan also started a very useful Stack Overflow tag for Progressive Web Apps worth checking out.

Principles

Web app manifest

> The Manifest for Web applications is a simple JSON file that gives you, the developer, the ability to control how your app appears to the user in the areas that they would expect to see apps (for example the device home screen), direct what the user can launch and more importantly how they can launch it

The manifest enables your web app to have a more native-like presence on the user's home screen. It allows the app to be launched in full-screen mode (without a URL bar being present), provides control over the screen orientation and in recent versions of Chrome on Android supports defining a Splash Screen and theme color for the address bar. It is also used to define a set of icons by size and density used for the aforementioned Splash screen and home screen icon.



A sample manifest file can be found in Web Starter Kit and over in the Google Chrome samples. Bruce Lawson wrote a Manifest Generator and Mounir Lamouri has also written a handy Web Manifest validator worth checking out.

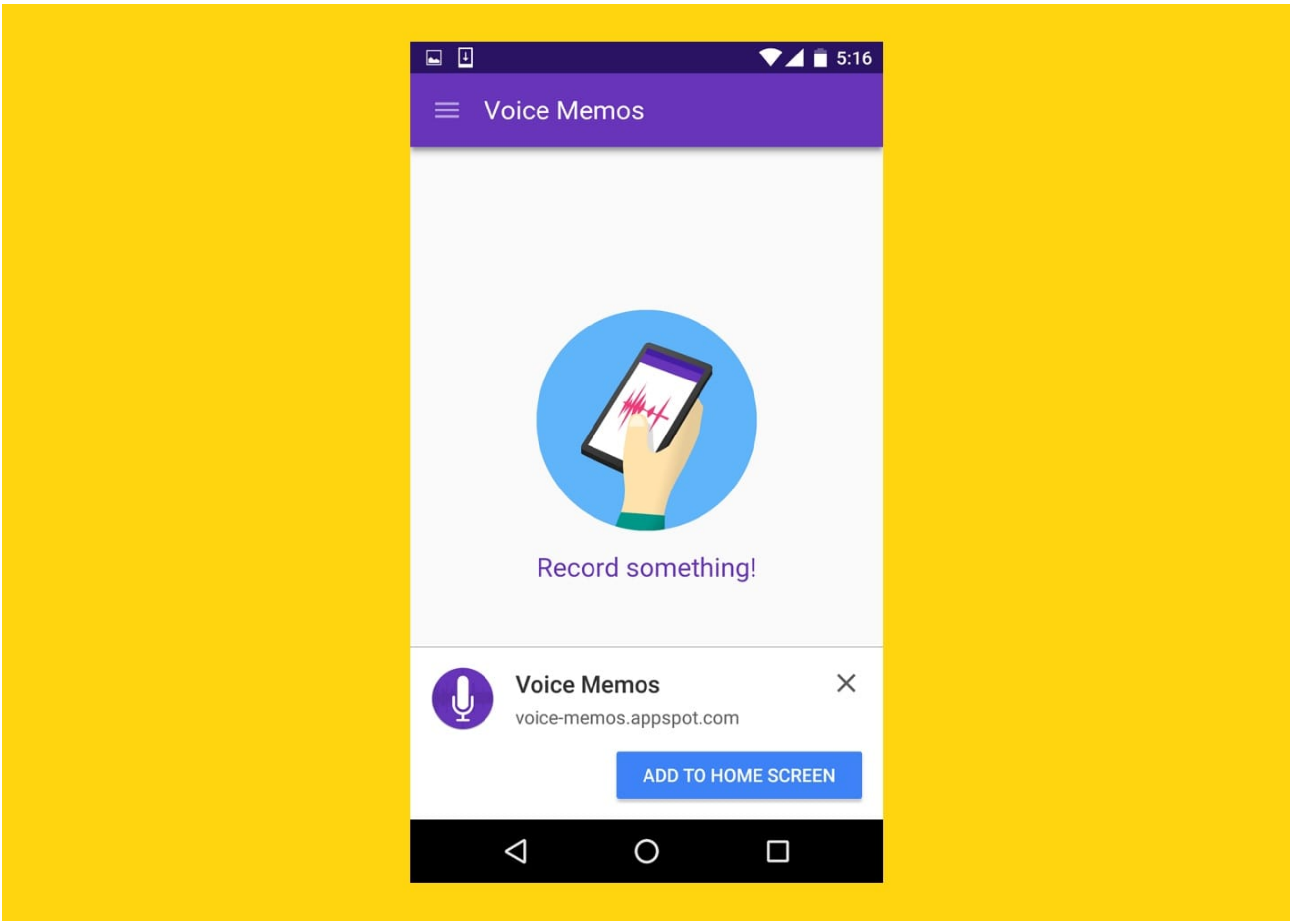
In my personal projects, I rely on realfavicongenerator to generate the correctly sized icons for both the web app manifest and for use across iOS, desktop and so on. The favicons Node module is also able to achieve a similar output as part of your build process.

Chromium-based browsers (Chrome, Opera etc.) support web app manifests today with Firefox actively developing support and Edge listing them as under consideration. WebKit/Safari have not yet posted public signals about their intents to implement the feature just yet.

For more details, read Installable Web Apps with the Web App Manifest in Chrome for Android on Web Fundamentals.

"Add to Home Screen" banner

Chrome on Android has support adding in your site to the home screen for a while now, but recent versions also support proactively suggesting sites be added using native Web App install banners.



In order for the app install prompts to display your app must:

- Have a valid web app manifest
- Be served over HTTPS (see letsencrypt for a free certificate)
- Have a valid service worker registered
- Be visited twice, with at least 5 minutes between visits

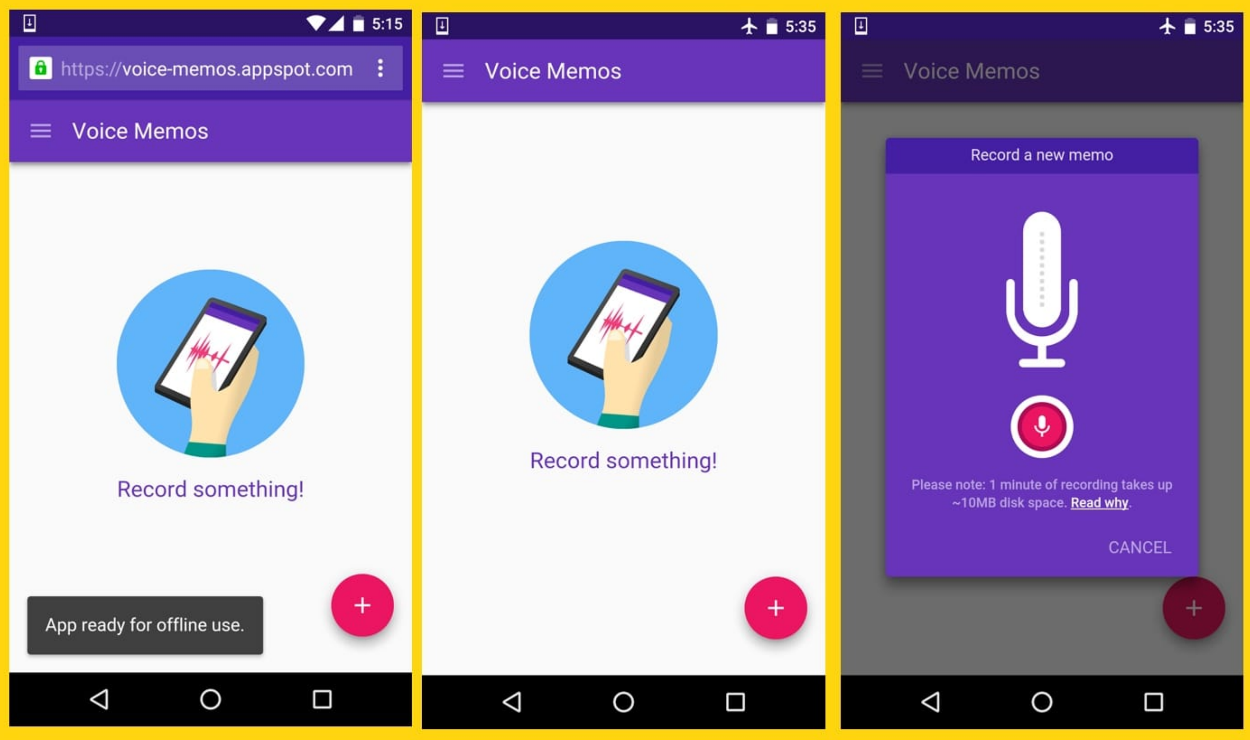
A number of App Install banner samples are available, covering basic banners through to more complex use-cases like displaying related applications.

Service worker for offline caching

A service worker is a script that runs in the background, separate from your web page. It responds to events, including network requests made from pages it serves. A service worker has an intentionally short lifetime.

It wakes up when it gets an event and runs only as long as it needs to process it. Service worker allows you to use the Cache API to cache resources and can be used to provide users with an offline experience.

Service workers are powerful for offline caching but they also offer significant performance wins in the form of instant loading for repeat visits to your site or web app. You can cache your application shell so it works offline and populate its content using JavaScript.


5:15
https://voice-memos.appspot.com
Voice Memos
Record something!
App ready for offline use.
5:35
Voice Memos
Record something!
5:35
Voice Memos
Record a new memo
Please note: 1 minute of recording takes up ~10MB disk space. Read why.
CANCEL

Service Worker caching of the app shell, allowing it to load without the network

A comprehensive set of service worker samples are available over on Google Chrome samples. Jake Archibald's offline cookbook is a must-read and I highly recommend trying out Paul Kinlan's your first offline web app walkthrough if new to service worker.

Our team also maintains a number of service worker helper utilities and build tools that we find useful for reducing the overhead in getting service worker setup. They're listed over on Service Worker Libraries. The two main ones are:

- sw-precache: a build-time tool that generates a service worker script useful for precaching your web app shell
- sw-toolbox: a library providing runtime caching for infrequently used resources

Jeff Posnick wrote a quick primer on sw-precache called Offline-first, fast, with the sw-precache module and a codelab on the same tool that you might find useful.

Chrome, Opera and Firefox have all implemented support for service worker with Edge having positive public signals about interest in the feature. Safari briefly mentioned interest in it via one engineer's proposed five year plan.

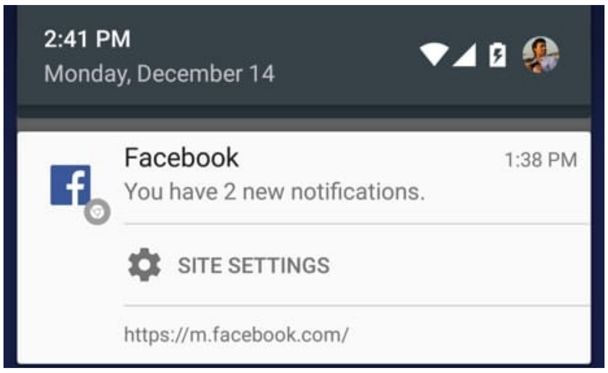
Push notifications for re-engagement

> Push notifications allow your users to opt-in to timely updates from sites they love and allow you to effectively re-engage them with customized, engaging content.

Effectively, you can build web apps that users can engage with outside of a tab. The browser can be closed and they don't even need to be actively using your web app to engage with your experience. The feature requires both service worker and a web app manifest, building on some of the features summarized earlier.

The Push API is implemented in Chrome, in development in Firefox and under consideration in Edge. There are no public signals from Safari about their intent to implement this feature just yet.

Push Notifications on the Open Web is a comprehensive intro to getting Push setup by Matt Gaunt and a Push Notifications codelab is also available on Web Fundamentals.


2:41 PM
Monday, December 14

Facebook 1:38 PM
You have 2 new notifications.

https://m.facebook.com/

Michael van Ouwerkerk from the Chrome team also has a 6 min intro to Push if you're more video inclined.

Layering in advanced features

Remember, your user experience can have different levels of sweetness depending on the browser being used to view your web app. You're in control of the hard candy shell.

Additional features coming to the web platform such as Background Syncronisation (for data sync with a server even when your web app is closed) and Web Bluetooth (for talking to Bluetooth devices from your web app) can also be layered into your Progressive Web App in this manner.

One-shot Background Sync has been enabled in Chrome and Jake Archibald has a video of his Offline wikipedia app and article demonstrating it in action. Francois Beaufort also has a number of Web Bluetooth samples available if interested in trying out that API.

Framework-friendly

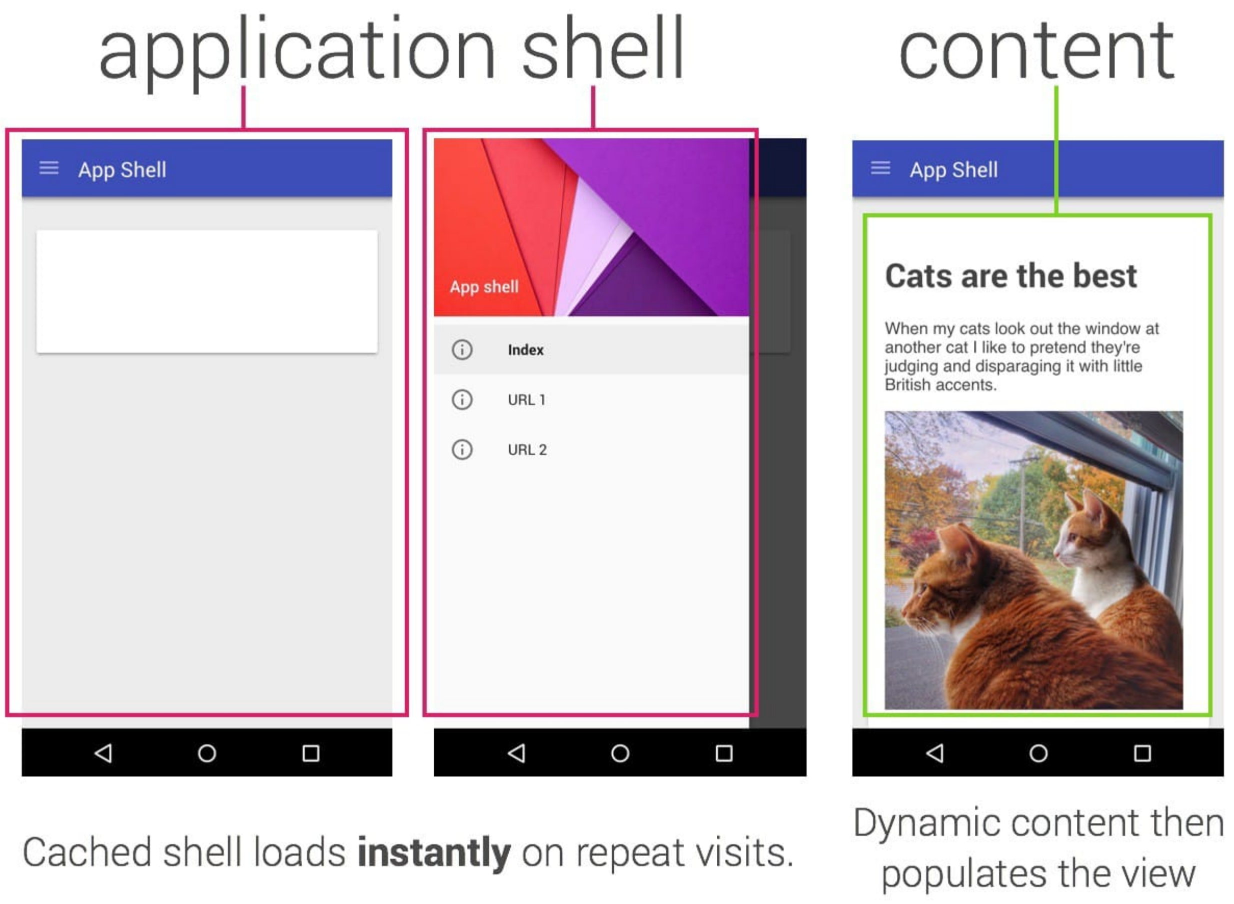
There's really nothing stopping you from applying any of the above principles to an existing application or framework you're building with. A few other principles worth keeping in mind while building your Progressive Web App are the RAIL user-centric performance model and FLIP based animations.

I'm hopeful that during 2016, we'll see an increasing number of boilerplates and seed projects organically baking in support for Progressive Web Apps as a first-class citizen. Until then, the barrier to adding these features to your own apps isn't very high and are IMHO, quite worth the effort.

Architecture

There are different levels of how "all-in" one goes on the Progressive Web App model, but one common approach taken is architecting them around an Application Shell. This is not a hard requirement, but does come with several benefits.

The Application Shell architecture encourages caching your application shell (the User Interface) so it works offline and populate its content using JavaScript. On repeat visits, this allows you to get meaningful pixels on the screen really fast without the network, even if your content eventually comes from there. This comes with significant performance gains.


application shell
content
App Shell
App shell
Index
URL 1
URL 2
App Shell
Cats are the best
When my cats look out the window at another cat I like to pretend they're judging and disparaging it with little British accents.
Cached shell loads **instantly** on repeat visits.
Dynamic content then populates the view

Jeremy Keith recently commented that in this type of model perhaps server-side rendering should not be viewed as a fallback but client-side rendering should be looked at as an enhancement. This is fair feedback.

In the Application Shell model, server-side rendering should be used as much as possible and client-side progressive rendering should be used as an enhancement in the same way that we "enhance" the experience when service worker is supported. There are many ways this can ultimately be approached.

My recommendation is reading our write-up on the architecture and evaluating how similar principles could be best applied to your own application and stack.

Getting started boilerplates

Application shell

View on GitHub

The **app-shell** repository contains a near-complete implementation of the Application Shell architecture. It has a backend written in Express.js and a front-end written in ES2015.

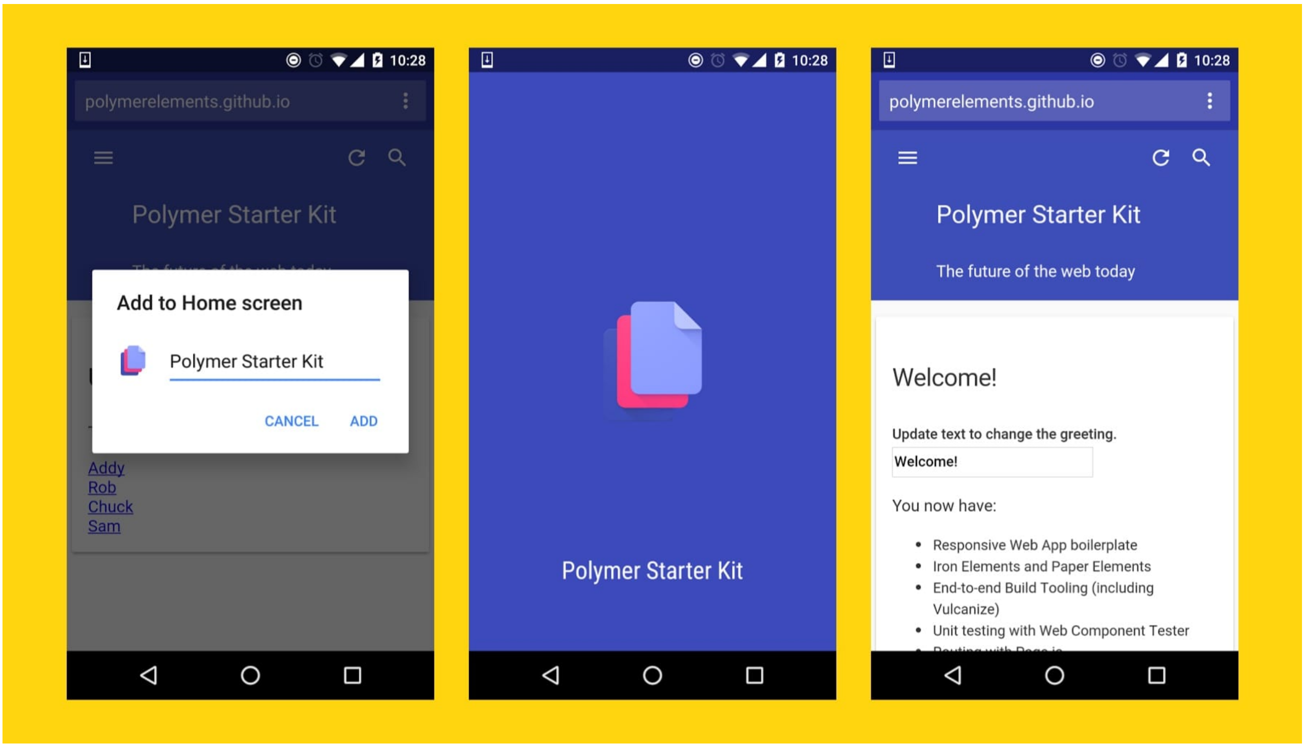
Given that it covers both client and server-side portions of the model and there's quite a lot going on there, it will take some time to familiarize yourself with the codebase. It's otherwise our most comprehensive Progressive Web App starting point right now. Docs will be our next focus for this project.

Polymer starter kit

View on GitHub

The official starting point for Polymer web apps supports the following Progressive Web App features:

- Web Application manifest
- Chrome for Android Splashscreen
- Service worker offline caching with the Platinum SW elements
- Push Notifications (manual setup required) with the Platinum Push elements


10:28
polymerelements.github.io
Polymer Starter Kit
Add to Home screen
Polymer Starter Kit
CANCEL
ADD
Addy
Rob
Chuck
Sam
10:28
Polymer Starter Kit
10:28
polymerelements.github.io
Polymer Starter Kit
The future of the web today
Welcome!
Update text to change the greeting.
Welcome!
You now have:
Responsive Web App boilerplate
Iron Elements and Paper Elements
End-to-end Build Tooling (including Vulcanize)
Unit testing with Web Component Tester

The current version of PSK is missing support for some of the more advanced performance patterns (e.g Application Shell model, async loading) you find in some Progressive Polymer web apps.

We aim to try baking these patterns into PSK in 2016, but early experiments around this can be found in the Polymer Zuperkulblog app by Rob Dodson and the excellent Polymer Perf Patterns talk by Eric Bidelman.

Web Starter Kit

View on GitHub

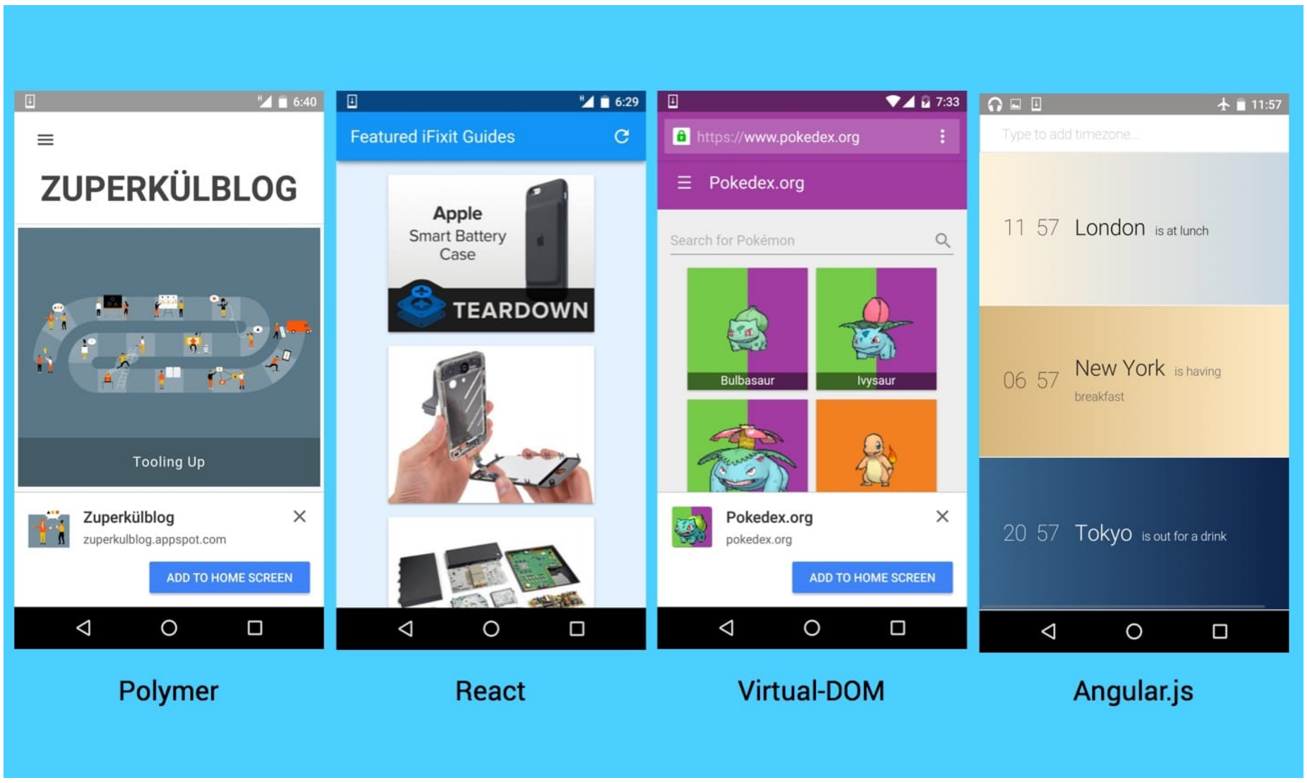
Our opinionated starting point for new vanilla projects includes the following Progressive Web App features:

- Web Application manifest
- Chrome for Android Splashscreen
- Service-worker pre-caching thanks to sw-precache

If you have a preference for working with vanilla JS/ES2015 and are unable to use Polymer, Web Starter Kit may prove useful as a reference point you can reuse or steal code snippets from.

Progressive Web Apps with and without frameworks

A number of open-source Progressive Web Apps have already been built by members of the community both with and without JS libraries and frameworks. If you're looking for inspiration, the below repos might prove useful as reference. They're also just pretty damn good apps.



Vanilla JavaScript

- Voice Memos by Paul Lewis is built using a similar architecture to `app-shell` (write-up)
- Offline Wikipedia by Jake Archibald (video)
- Air Horner by Paul Kinlan
- Guitar Tuner by Paul Lewis (write-up)

Polymer

- Zuperkulblog by Rob Dodson (slides)

React

- iFixit by Jeff Posnick - uses `sw-precache` for application shell caching (slides)

Virtual-DOM

- Pokedex by Nolan Lawson - excellent progressive web app applying a "do everything in a web worker" approach to help with progressive rendering. (write-up)

Angular.js

- Timey.in by Kenneth Auchenberg - also uses `sw-precache` for resource precaching

Closing notes

As mentioned, Progressive Web Apps are still in their infancy but it's an exciting time to play around with the methodologies behind them and see how well they can apply to your own web apps.

Paul Kinlan is currently planning out the Web Fundamentals guidance on Progressive Web Apps and if you have input on areas you would like to see covered, please feel free to comment on-thread.

Further reading

- Progressive Web Apps: Escaping Tabs Without Losing Our Soul
- Why Progressive Web Apps Are The Future Of Web Development
- Progressive Web Apps: ready for primetime
- Making a Progressive App with ServiceWorker
- Progressive Web Apps Are the Future
- Progressive Web App: A New Way to Experience Mobile
- Introducing Pokedex.org: a progressive webapp for Pokémon fans
- Chrome Developer Summit Recap: Progressive Web Apps