

Parte III: Desarrollo en dispositivos móviles

Tema 5:

Progressive Web Apps

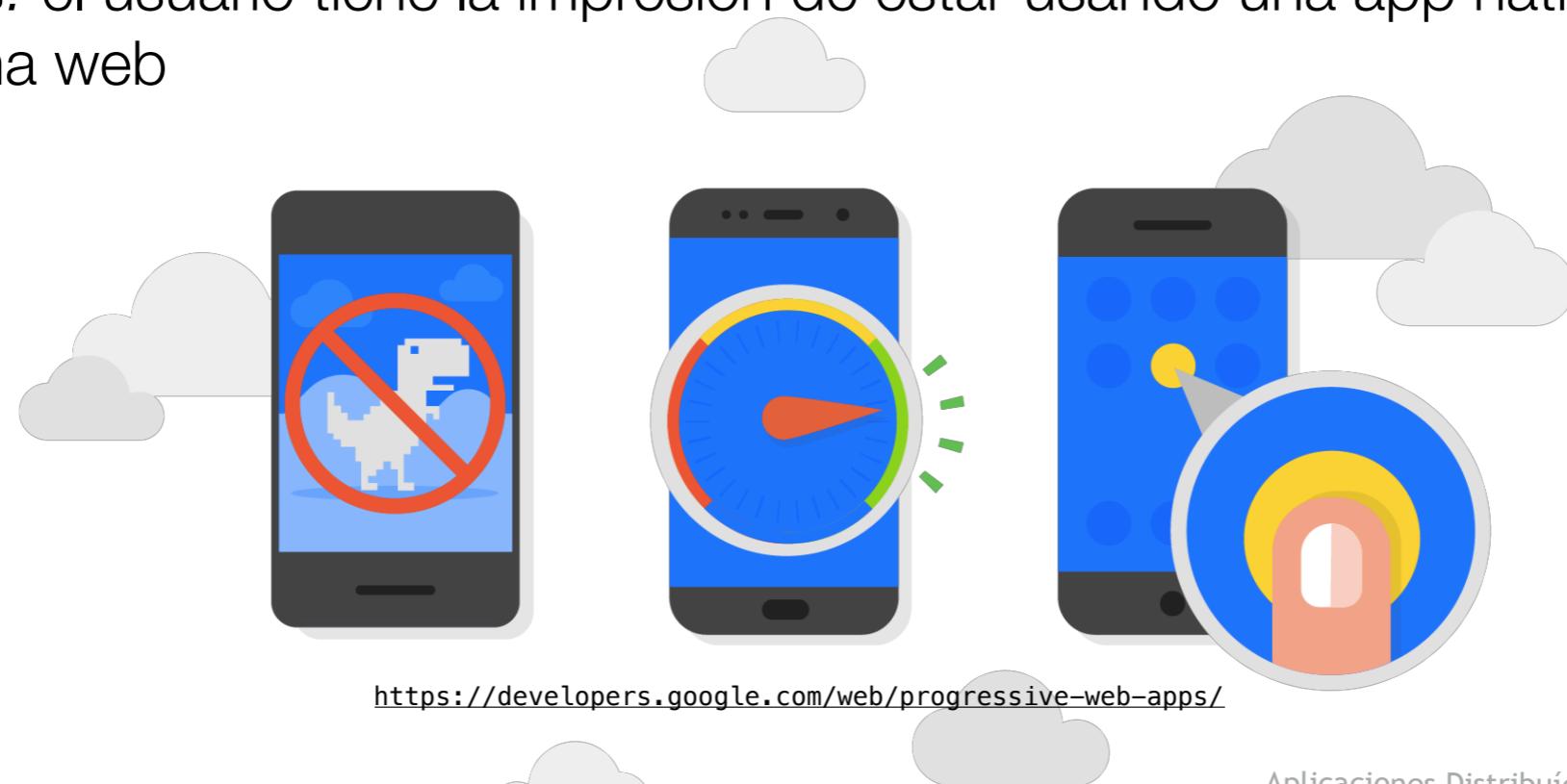
Parte III: Desarrollo en dispositivos móviles
Tema 5: Progressive Web Apps

5.1.

Introducción

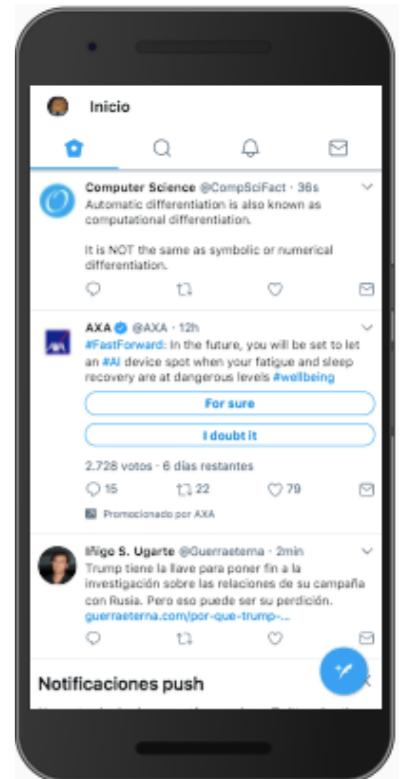
Progressive Web Apps

- “Buzzword” para denominar a las **aplicaciones web** para **móviles** que intentan proporcionar al usuario **la misma experiencia que una app nativa**
- En términos *no técnicos*, son
 - *Fiables*: funcionan (aunque sea parcialmente) aunque no haya conexión de red
 - *Rápidas*: la carga de la interfaz es rápida y también la interacción con el usuario
 - *Atractivas*: el usuario tiene la impresión de estar usando una app nativa, no de estar viendo una web

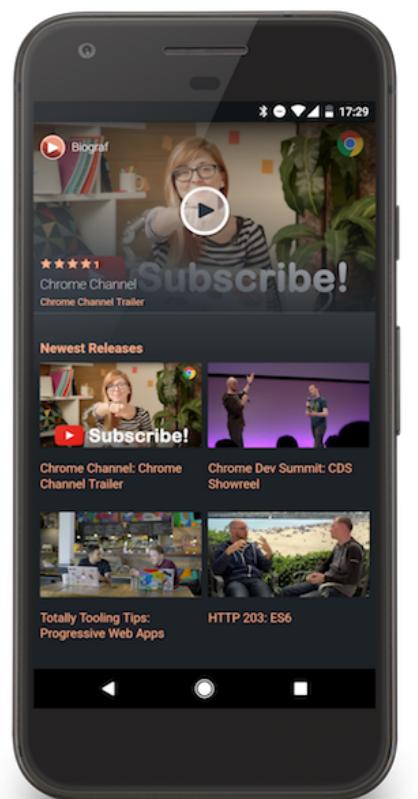


Ejemplos

- Algunas apps en producción
 - Uber: <https://m.uber.com>
 - AirBnB: <https://www.airbnb.pt/>
 - Twitter Lite: <https://mobile.twitter.com>
 - Google Maps Go: <https://www.google.com/maps?force=qVTs2FOxxTmHHo79-pwa&source=mlapk>
 - Casos de estudio: <https://developers.google.com/web/showcase/2017/>
 - Directorio de PWAs: <https://pwa-directory.appspot.com>
- Ejemplos “educativos”, con código fuente
 - **Sample media PWA:** <https://github.com/GoogleChromeLabs/sample-media-pwa>
 - **Voice memos:** <https://aerotwist.com/blog/voice-memos/>



Twitter lite <https://mobile.twitter.com/home>



Sample media PWA
online: <https://biograf-155113.appspot.com>

Hacker News PWA

<https://hnpwa.com/>

Aplicación de ejemplo: lector de Hacker News en versión PWA.
Disponibles versiones en distintos *frameworks* (React, Angular, Vue, Preact, Javascript *vanilla*,) Todos los fuentes son accesibles, para que los desarrolladores puedan comparar

React HN

kristoferbaxter/react-hn

Lighthouse: 91/100

Interactive (Emerging Markets): 2.57s

Interactive (Faster 3G): 2.09s

Framework/UI libraries: React, React Router

Module bundling: Webpack

Service Worker: Application Shell with OfflinePlugin

Performance patterns:

HTTP/2 with Server Push, Brotli and Zopfli static assets

Server-side rendering: Yes

API: In-memory cached Hacker News Firebase API

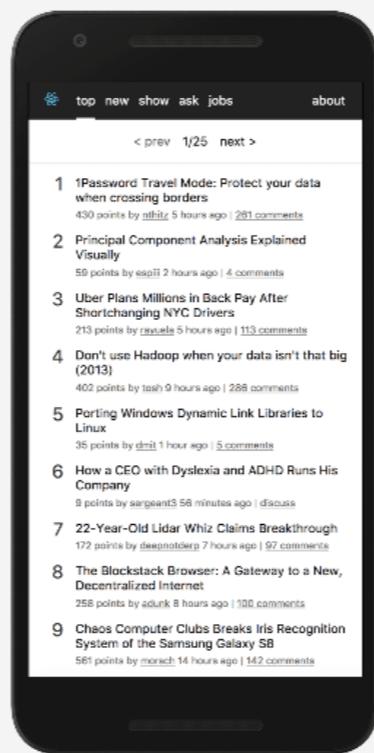
Hosting: Webfaction + Cloudflare

Author:



[VIEW APP](#)

[SOURCE CODE](#)



Angular HN

housseindjirdeh/angular2-hn

Lighthouse: 91/100

Interactive (Emerging Markets): 6.0s

Interactive (Faster 3G): 4.4s

Framework/UI libraries: Angular

Scaffolding: Angular CLI

Module bundling: Webpack

Service Worker:

Application Shell + data caching with sw-precache

Performance patterns: Lazy loaded modules

Server-side rendering: None

API: Node-hnapi (unofficial)

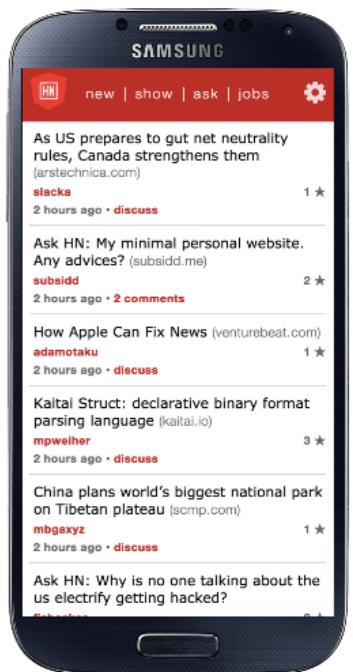
Hosting: Firebase

Author:



[VIEW APP](#)

[SOURCE CODE](#)



Características de una PWA

Funcionalidades más detalladas de una PWA y **tecnologías/técnicas** que las hacen posibles

- El diseño está adaptado a móviles (*responsive media queries, viewport*)
- No parece una web sino una app (*Web App Manifest*)
 - Se puede añadir un ícono al escritorio
 - Está a pantalla completa, no se ve el navegador
 - Tiene una pantalla de *splash*
- Funciona también *offline* (*cache con Service Workers*)
- La carga es rápida (*cache con Service Workers*)
- Puede recibir notificaciones *push* (*Service Workers+Push+NotificationAPI*)
- Lighthouse: herramienta para medir si un sitio web cumple estas condiciones

Tema 5: Progressive Web Apps

5.2.

Detalles nativos:
Web App Manifest

Web App Manifest

Archivo JSON con información sobre la app (nombre, iconos, color de fondo para la pantalla splash,...). [\(Más info en MDN\)](#)

```
{  
  "name": "HackerWeb",  
  "short_name": "HackerWeb",  
  "start_url": ".",  
  "display": "standalone",  
  "background_color": "#fff",  
  "description": "A simply readable Hacker News app.",  
  "icons": [ {  
      "src": "images/touch/homescreen48.png",  
      "sizes": "48x48",  
      "type": "image/png"  
    } ]  
  ...  
}
```

```
<link rel="manifest" href="/manifest.json">
```

En `<head>` del “`index.html`”

Banner de instalación

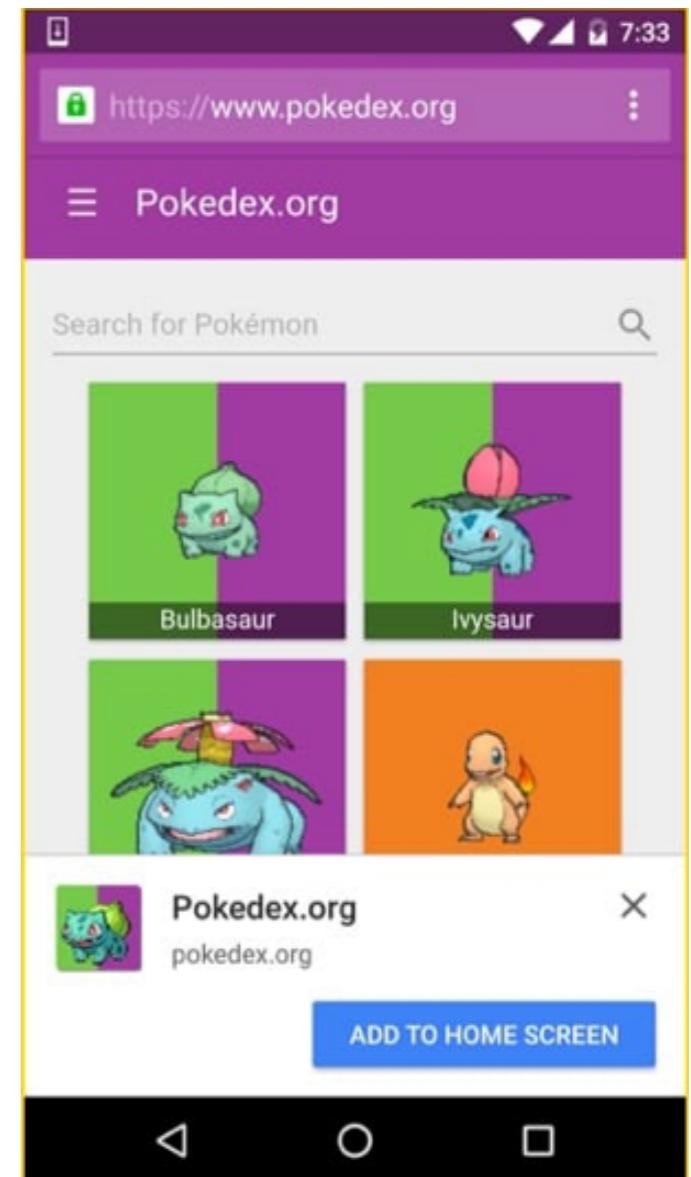
- El navegador sugiere al usuario añadir un ícono para la PWA al lanzador de apps
- El desarrollador no puede controlar este *banner*, para evitar *spam*. Lo decide Chrome en base a ciertas heurísticas

¿Cuáles son los criterios?

Chrome mostrará de manera automática el banner cuando tu app cumpla con los siguientes criterios:

- Contener un archivo de [manifiesto de aplicación web](#) con:
 - un `short_name` (usado en la pantalla de inicio);
 - un `name` (usado en el banner);
 - un ícono png de 192 x 192 (en las declaraciones del ícono se debe incluir un tipo de mime `image/png`);
 - una `start_url` que se carga.
- Contener un [service worker](#) registrado en tu sitio.
- Transmitirse a través de [HTTPS](#) (un requisito para usar el service worker).
- Recibir visitas al menos dos veces, con al menos cinco minutos de diferencia entre las visitas.

<https://developers.google.com/web/fundamentals/app-install-banners/?hl=es>



Cuestiones prácticas

- Soporte al menos parcial en la mayoría de navegadores móviles. iOS de momento no soporta funcionalidades como el splash y otras (más información)

Web App Manifest - WD

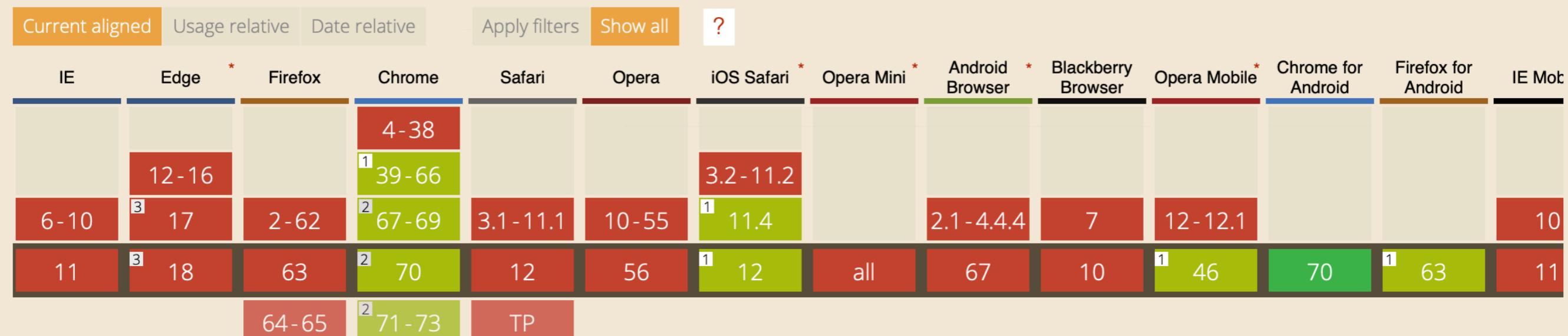
Usage

Global

% of all users

31.99% + 44.56% = 76.55%

The web app manifest provides information about an application (such as name, author, icon, and description) in a JSON file, which browsers can use to give richer offline experiences.



<http://caniuse.com/#feat=web-app-manifest>

- La parte más tediosa suele ser generar los iconos en diferentes resoluciones, se puede usar por ejemplo <http://realfavicongenerator.net>

Parte III: Desarrollo en dispositivos móviles
Tema 5: Progressive Web Apps

5.3.

Funcionando *offline*:
Service Workers

AppCache

- Funcionalidad para forzar a cachear recursos

```
<html manifest="example.appcache">  
  ...  
</html>
```

index.html

```
CACHE MANIFEST  
# Entradas en la cache  
CACHE:  
/favicon.ico  
index.html  
stylesheet.css  
images/logo.png  
scripts/main.js
```

```
# Recursos solo disponibles si se está online
```

```
NETWORK:
```

```
*
```

```
# Si no hay disponible una cosa (images/), se verá la otra (/images/offline.jpg)
```

```
FALLBACK:
```

```
images/ images/offline.jpg
```

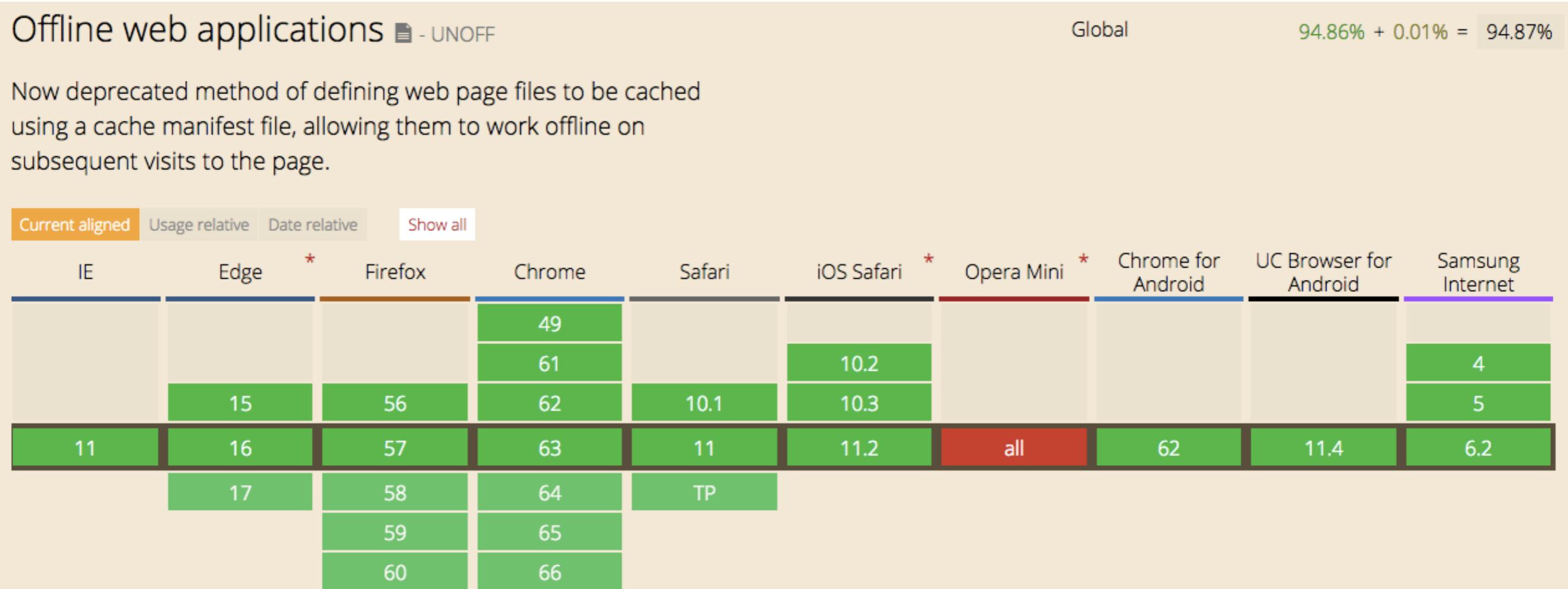
example.appcache

Lamentablemente (o no), AppCache está “deprecated”



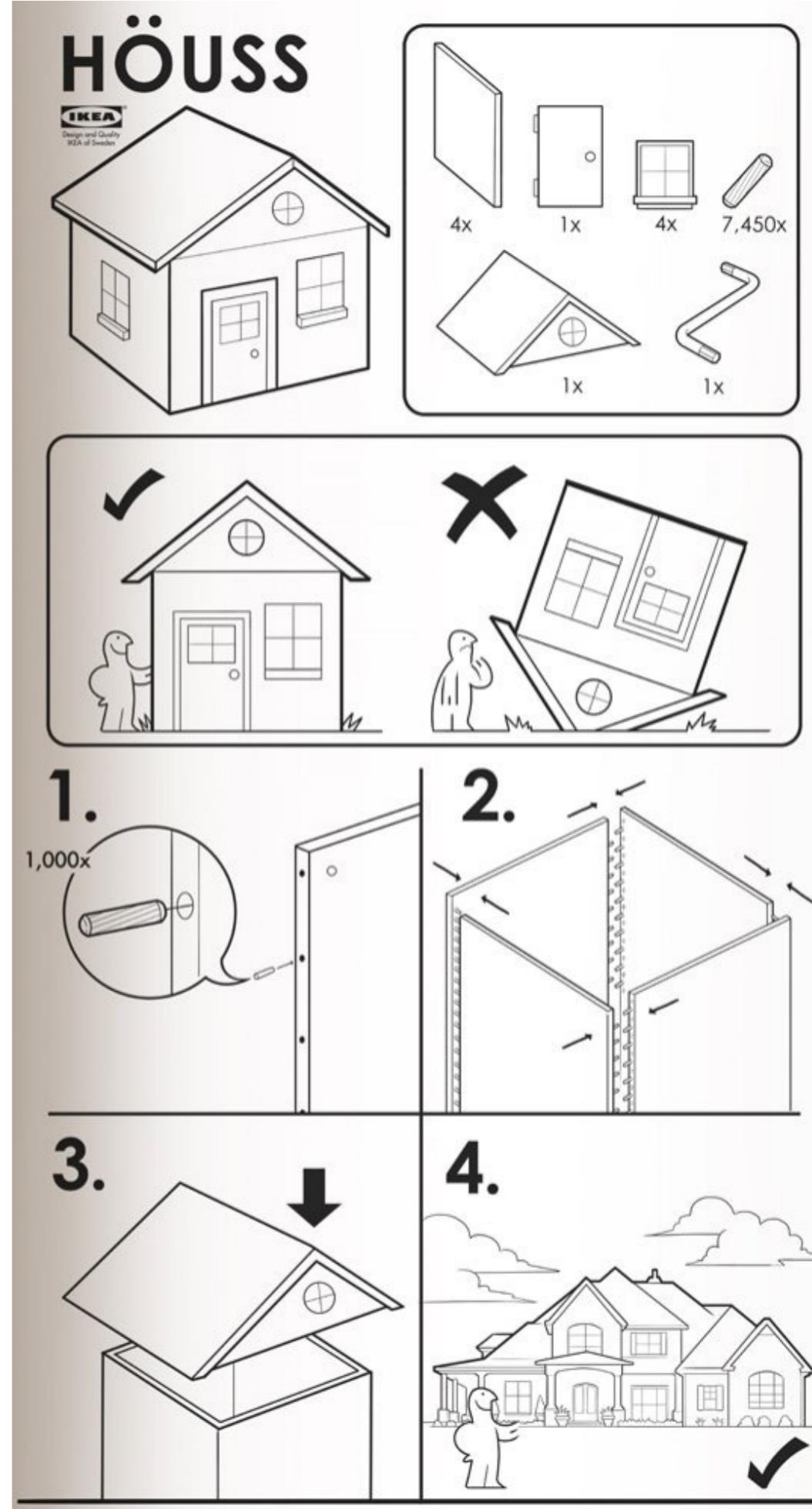
See you in your dreams!

No obstante, el soporte de este API todavía es bastante amplio



La alternativa actual son los **service workers**, que siguen un enfoque diferente: el de “háztelo tú mismo”

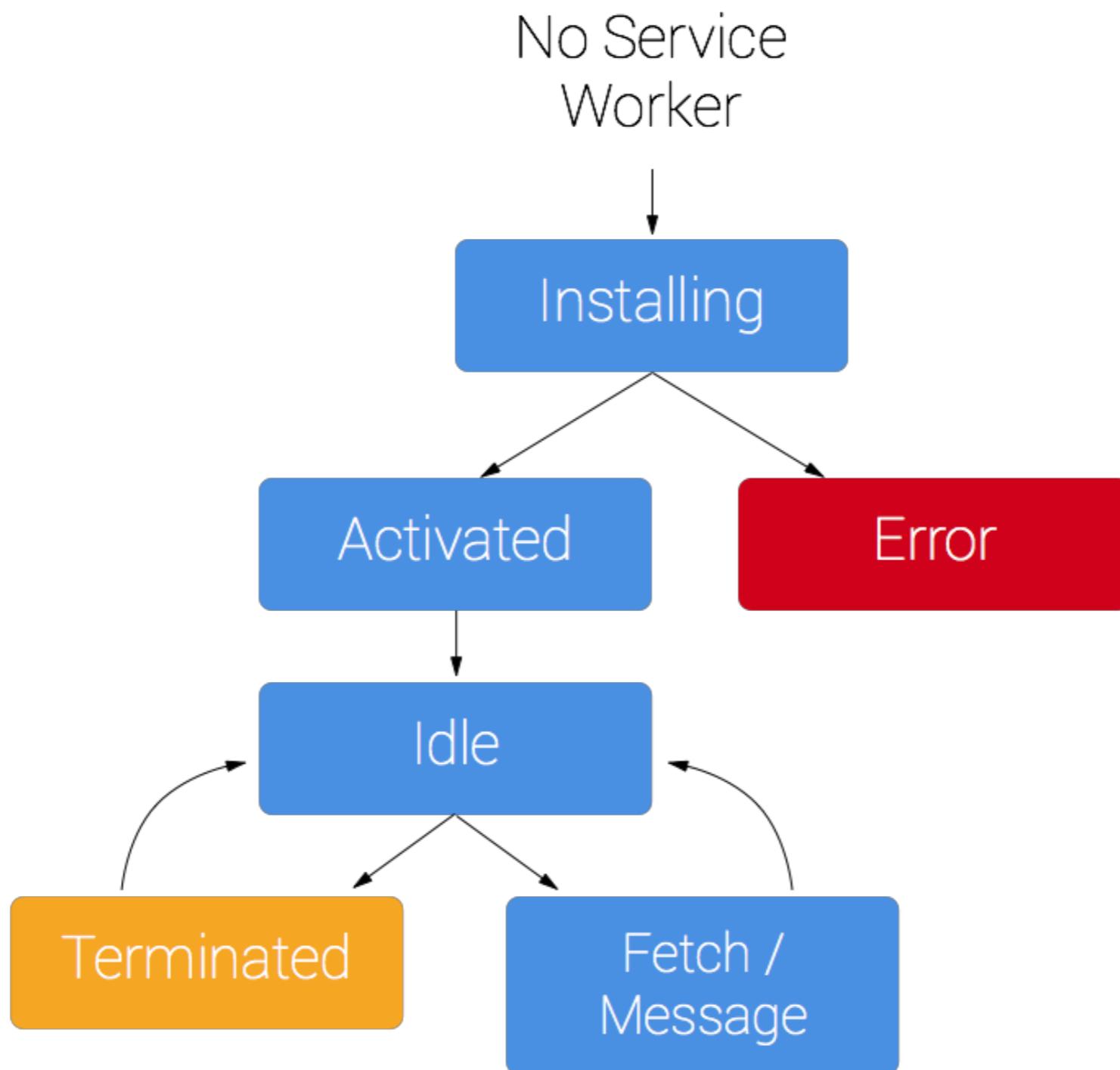
Son un API de “**bajo nivel**” con el que se pueden implementar caches además de muchas otras funcionalidades



Service Workers

- Scripts que se ejecutan “**en segundo plano**”, independientemente de la página original
- Pueden detectar ciertos eventos interesantes
 - `fetch`: se ha hecho una solicitud HTTP
 - `push`: se ha enviado una notificación desde el servidor al navegador
- Son asíncronos
 - Su API usa promesas
 - **No tienen acceso al DOM** ni a la mayoría de APIs síncronas (por ejemplo `localStorage`) , solo a algunos APIs asíncronos

Ciclo de vida



<http://www.html5rocks.com/en/tutorials/service-worker/introduction/>

Aplicaciones habituales de los SW

- Cache para funcionar *offline*
 - cachear automáticamente todo el sitio web
 - Marcar alguna página para “leer sin conexión”
- *Proxy* de red
 - chequear ciertas condiciones
 - añadir credenciales,
 - ...
- Notificaciones *push*



Todas estas funcionalidades no las da directamente implementadas el API de SW, hay que escribir algo de código (ganamos flexibilidad a cambio de complejidad de uso)

The screenshot shows a Mozilla Firefox browser window displaying the Service Worker Cookbook at <https://serviceworker.rs/>. The page has a green sidebar on the left containing navigation links for various service worker recipes and sections like 'Introduction' and 'Attribution'. The main content area features large green and red headings for 'Introduction' and 'Attribution', respectively, with descriptive text and a tip about developer tools. Below these are sections for 'Recipes' (with 'Immediate Claim' as the first item) and 'Message Relay'.

ServiceWorker Cookbook

mozilla

Introduction

Contribute on GitHub

General Usage

- B Immediate Claim
- B Message Relay
- I Fetching Remote Resources
- A Live Flowchart

Offline

- B JSON Cache
- B Offline Fallback
- B Offline Status

Beyond Offline

- I Virtual Server
- I API Analytics
- I Load balancer

Introduction

The Service Worker Cookbook is a collection of working, practical examples of using service workers in modern web apps.

Tip: Open your Developer Tools console to view `fetch` events and informative messages about what each recipe's service worker is doing!

Attribution

The Service Worker Cookbook was created by Mozilla with contributions from developers like you. All source code is available on GitHub. Contributions and requests welcome.

Recipes

Immediate Claim

This recipe shows how to have the service worker immediately take control of the page without waiting for a navigation event.

Message Relay

This recipe shows how to communicate between the service worker and a page and shows how to use a

Registro de un SW

```
//El código del SW está en ‘service-worker.js’, este script
//solo lo “carga”
if ('serviceWorker' in navigator) {
    navigator.serviceWorker
        .register('service-worker.js')
        .then(function(registro){
            console.log("registro SW ok");
        });
}
else {
    console.log("Service Workers no soportados");
}
```

Soporte actual de los SW (nov '18)

Service Workers - WD

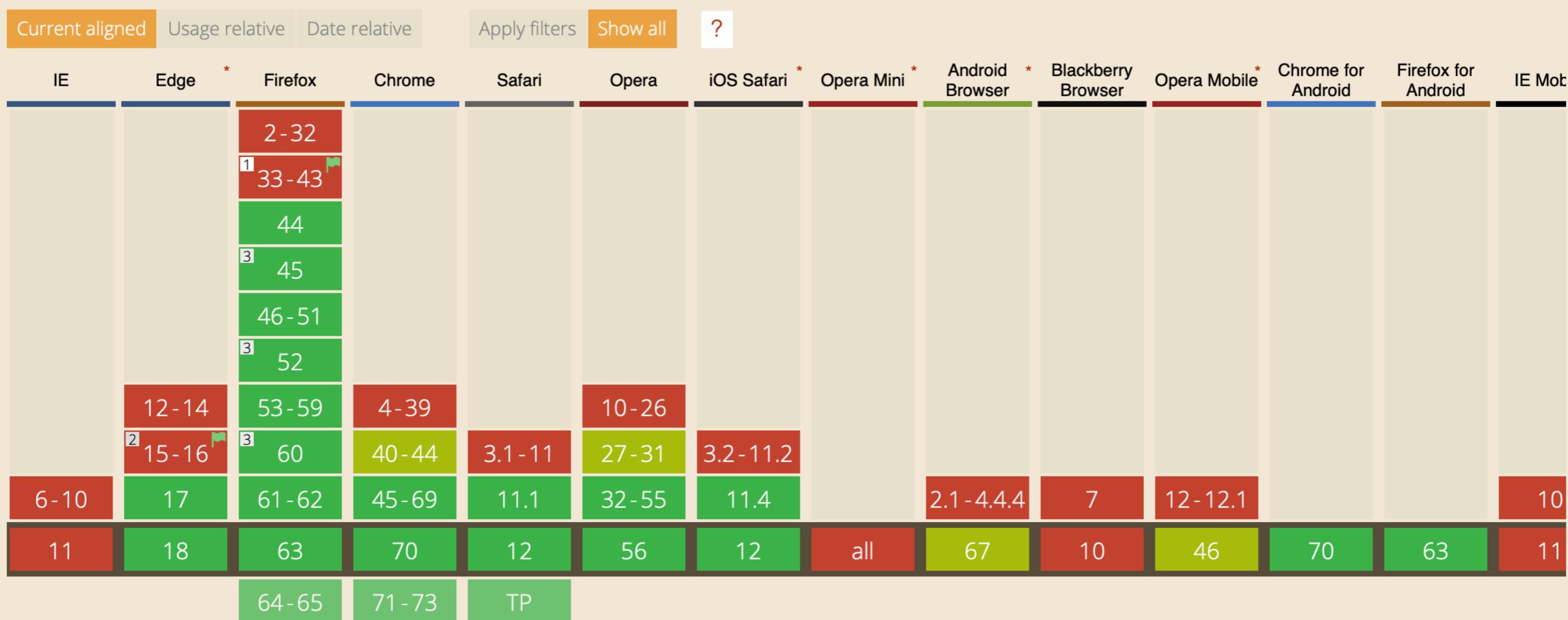
Usage

% of all users

Global

85.69% + 0.15% = 85.84%

Method that enables applications to take advantage of persistent background processing, including hooks to enable bootstrapping of web applications while offline.



<https://caniuse.com/#search=service%20workers>

Eventos

```
//Esto es el service worker ('service-worker.js')

//Podemos atender a eventos del ciclo de vida
self.addEventListener('install', function(evento) {
    console.log('[install]');
})

//...o a algunos eventos del navegador
self.addEventListener('fetch', function(evento){
    console.log('[fetch] a ' + evento.request.url);
});
```

Cache API

- Permite gestionar caches web de modo sencillo. Asociado al API de Service Workers
 - Crear una cache (`caches.open`)
 - Añadir recursos a la cache (`cache.addAll`)
 - Comprobar si un recurso está ya en cache (`cache.match`, `caches.match`)
- Es un API asíncrono, basado en promesas

```
caches.open('MI_CACHE')
  .then(function(cache) {
    return cache.addAll(['recurso1.jpg', 'recurso1.html']);
  }).then(function() {
    console.log('recursos añadidos a la cache')
  })
}
```

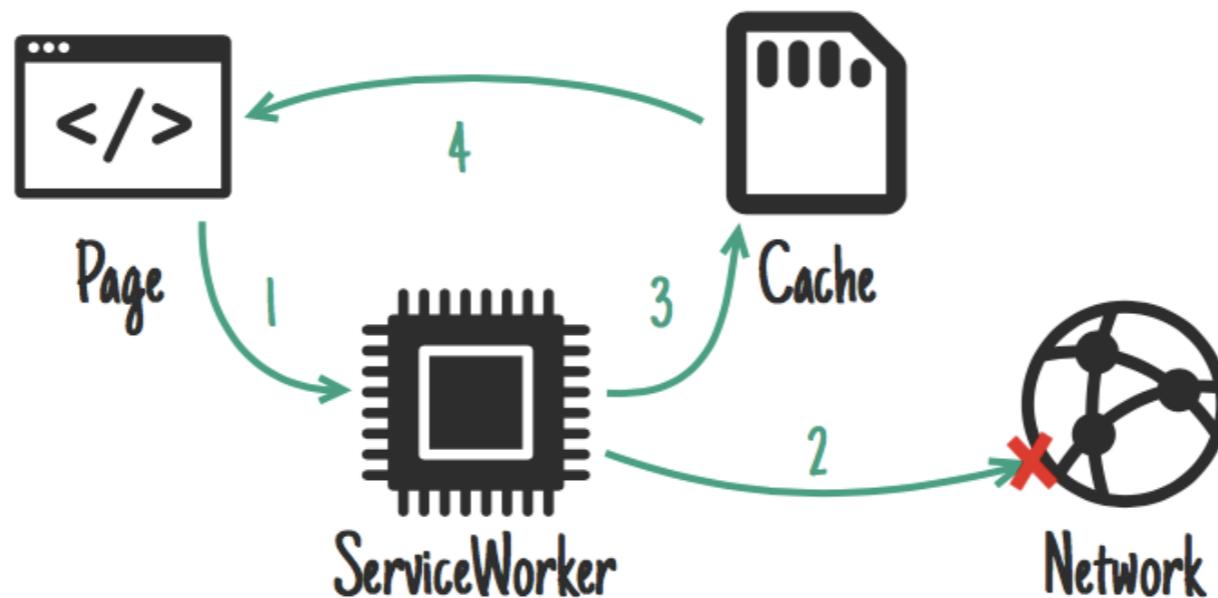
Crear cache y añadir recursos

```
var CACHE_NAME = 'my-site-cache-v1';
var urlsACachear = [
  '/',
  '/styles/main.css',
  '/script/main.js'
];

self.addEventListener('install', function(event) {
  //waitUntil asegura que el event listener espera
  //hasta que se resuelva la promesa
  event.waitUntil(
    caches.open(CACHE_NAME)
      .then(function(cache) {
        return cache.addAll(urlsACachear);
      })
  );
});
```

“Receta ejemplo”: cache fallback

- Si al hacer una petición falla la red, “tirar” de cache



<https://jakearchibald.com/2014/offline-cookbook/#network-falling-back-to-cache>

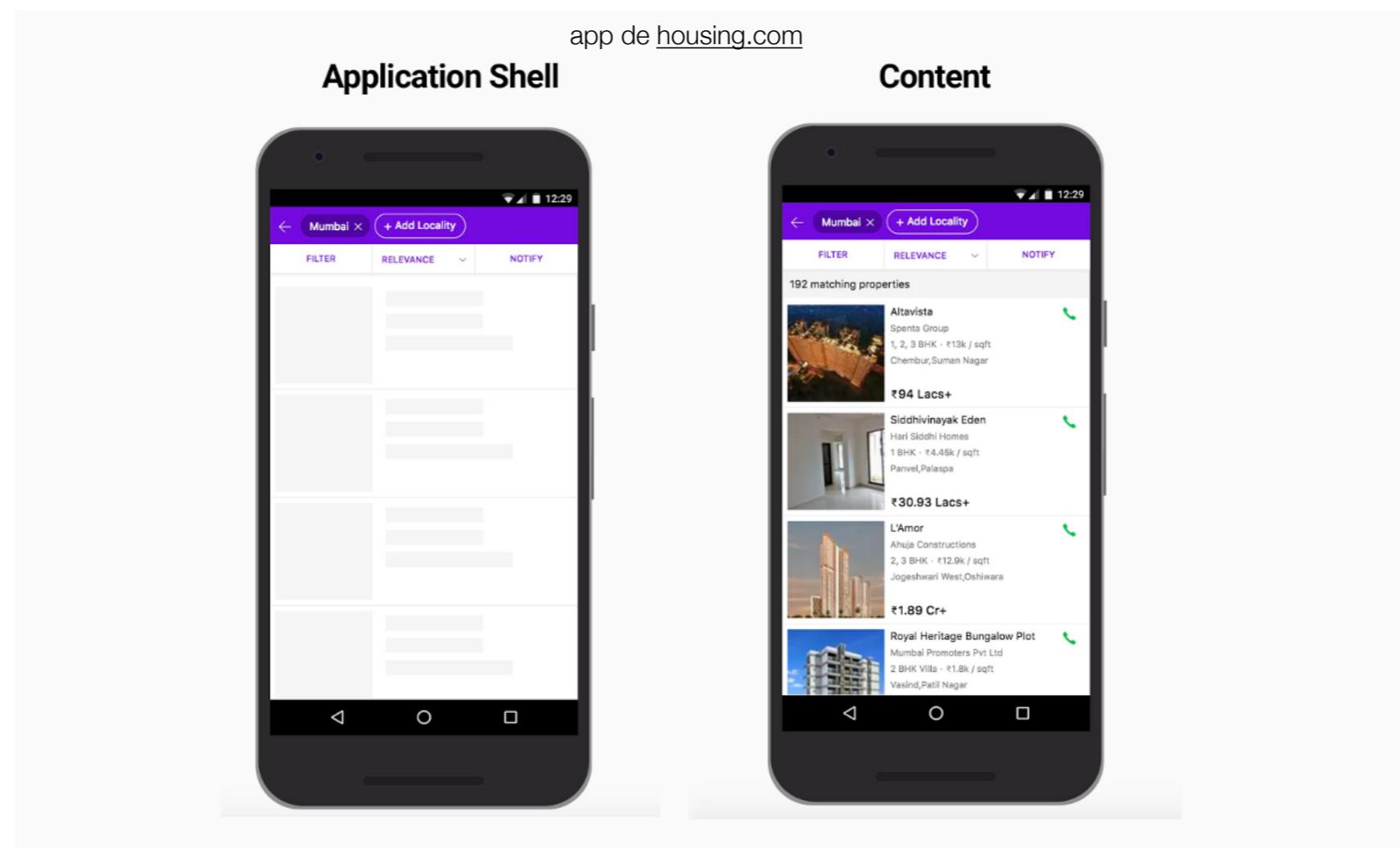
```
self.addEventListener('fetch', function(event) {  
  //respondWith nos permite cambiar la respuesta a devolver ante  
  //una petición HTTP hecha con fetch  
  event.respondWith(  
    fetch(event.request).catch(function() {  
      return caches.match(event.request);  
    })  
  );
```

Si hay error (porque la red no va) se dispara el catch, y devolvemos el recurso correspondiente si es que está en la cache

En un listener de ‘fetch’, event.request representa la petición original. Así que si no hay error (no se dispara el catch) realizamos la petición original

Ejemplo: Application Shell

- Es el HTML/CSS que forma el “esqueleto estático” de nuestra app
 - Podemos cachearlo con service workers, así cuando se cargue la app al menos el esqueleto aparece instantáneamente
- El contenido se obtiene dinámicamente del servidor con un fetch (JS)



De: <https://medium.com/@addyoosmani/progressive-web-apps-with-react-js-part-3-offline-support-and-network-resilience-c84db889162c#.deat1b69s>

“Receta ejemplo”: cachear el shell

1. Primero habrá que cachear los recursos que forman el *shell*, en el evento “install” del service worker (como en la transparencia 24)
2. Interceptar cualquier petición de red y si esta encaja con alguna cache, servirla desde ahí. En caso contrario, ir a la red (o sea, primero cache, si falla a la red, al contrario que en el “cache fallback”)

```
self.addEventListener('fetch', function(event) {  
  event.respondWith(  
    caches.match(event.request).then(function(response) {  
      return response || fetch(e.request);  
    })  
  );  
});
```

Tutorial con app de ejemplo <https://codelabs.developers.google.com/codelabs/your-first-pwapp/>

¡Demasiado bajo nivel!

- El API de SW es muy **flexible** y potente pero también **tedioso**
- Ya hemos visto patrones o recetas de uso
 - **Service Workers cookbook:** <https://serviceworkerjs.com/cookbook/>
 - **The offline cookbook:** <https://jakearchibald.com/2014/offline-cookbook/>
- Algunas librerías proporcionan una capa de abstracción sobre el API de SW: por ejemplo, **workbox**, de Google (<https://developers.google.com/web/tools/workbox/>)



Ejemplos workbox

- “cache first” para las imágenes. Además fijamos una expiración

```
workbox.routing.registerRoute(  
  /\.(\?:png|gif|jpg|jpeg|svg)$/,  
  workbox.strategies.cacheFirst({  
    cacheName: 'images',  
    plugins: [  
      new workbox.expiration.Plugin({  
        maxEntries: 60,  
        maxAgeSeconds: 30 * 24 * 60 * 60, // 30 Days  
      }),  
      ],  
    }),  
  );
```

- “stale while revalidate”: tomar el dato de la cache, pero actualizar en background para que la próxima vez esté actualizado

```
workbox.routing.registerRoute(  
  /\.(\?:js|css)$/,  
  workbox.strategies.staleWhileRevalidate(),  
);
```

Parte III: Desarrollo en dispositivos móviles
Tema 5: Progressive Web Apps

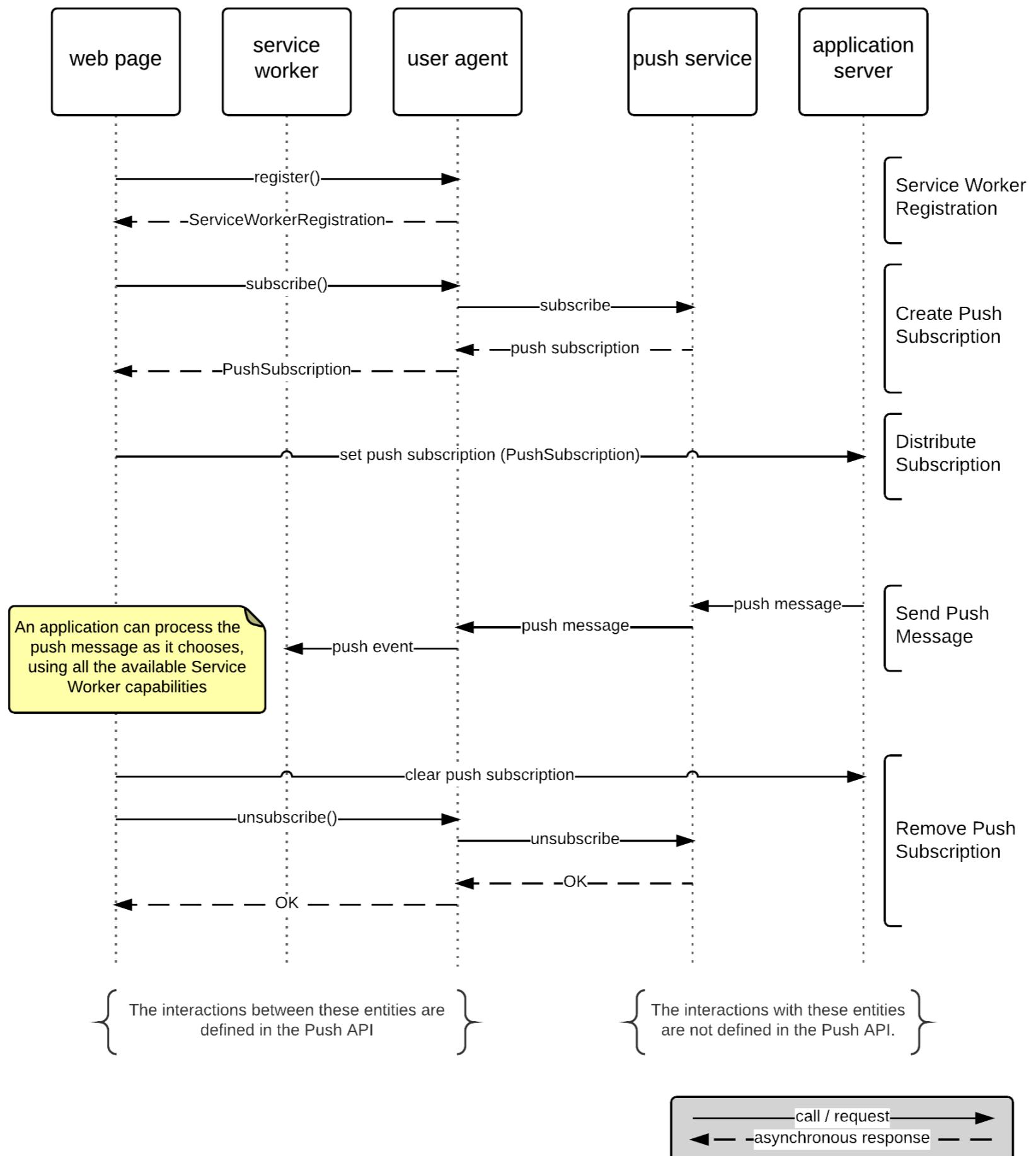
5.4

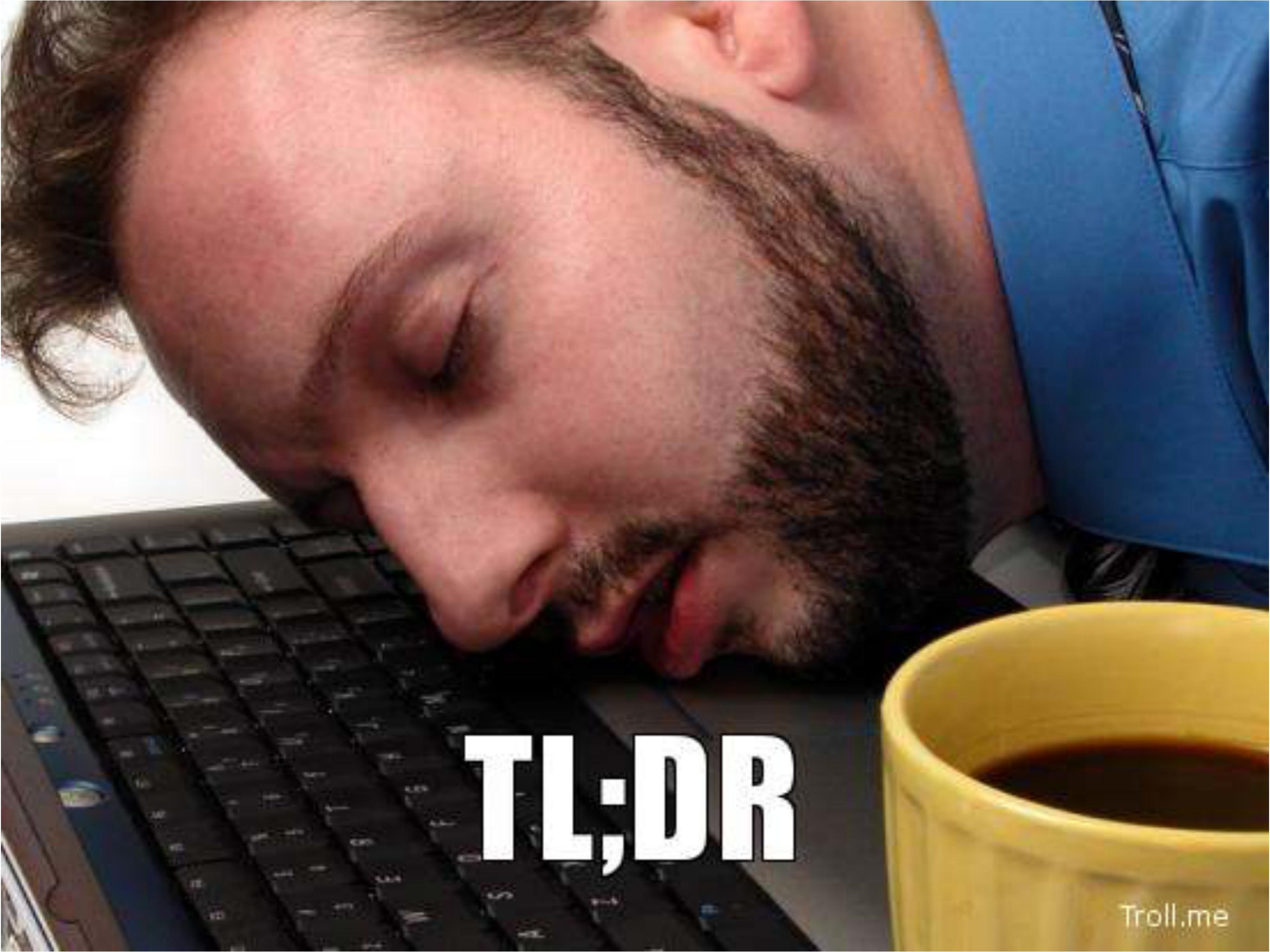
“Enganchando” al
usuario:
Notificaciones

Web push

- Desde hace tiempo las **aplicaciones nativas** pueden recibir notificaciones *push*
- También es posible en **aplicaciones web** con la ayuda de
 - **Service Workers**
 - **Notifications**
 - **Push API**
- Colaboración adicional de un **servidor de mensajes** *push* (por ejemplo *FCM-Firebase Cloud Messaging* o *Mozilla push service*: <https://mozilla-push-service.readthedocs.io/en/latest/>)
- Las notificaciones se reciben **aunque el navegador esté en otra web o cerrado (en móviles)**
- Por ahora **solo en Android (Chrome/Firefox)**. Safari no implementa el estándar de *push*, sino uno propio, y además solo en OSX, no en iOS



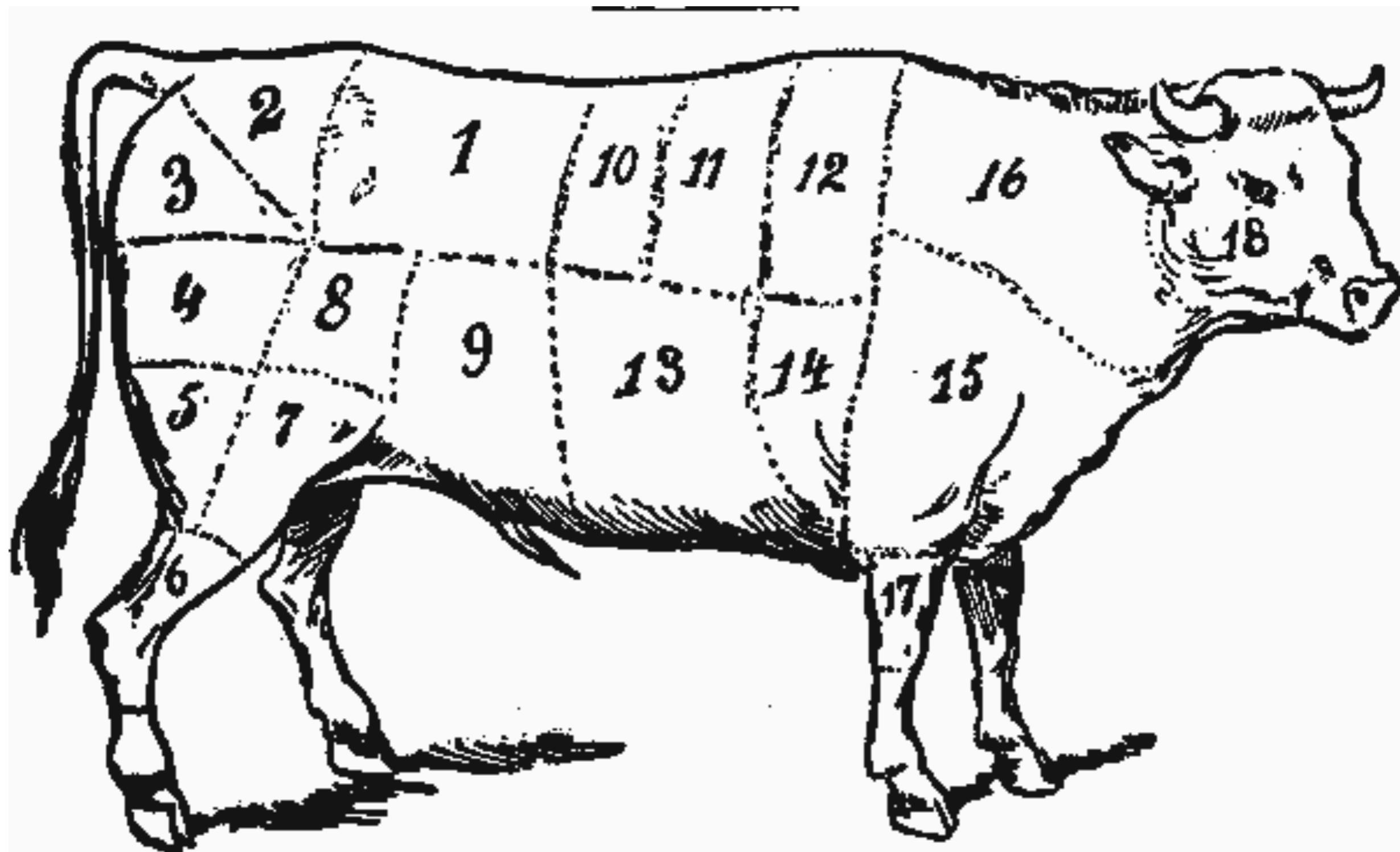


A close-up photograph of a man with a dark beard and mustache, wearing a blue suit jacket over a white shirt. He is looking down intently at a black computer keyboard. A yellow coffee mug is visible on the right side of the frame.

TL;DR

Troll.me

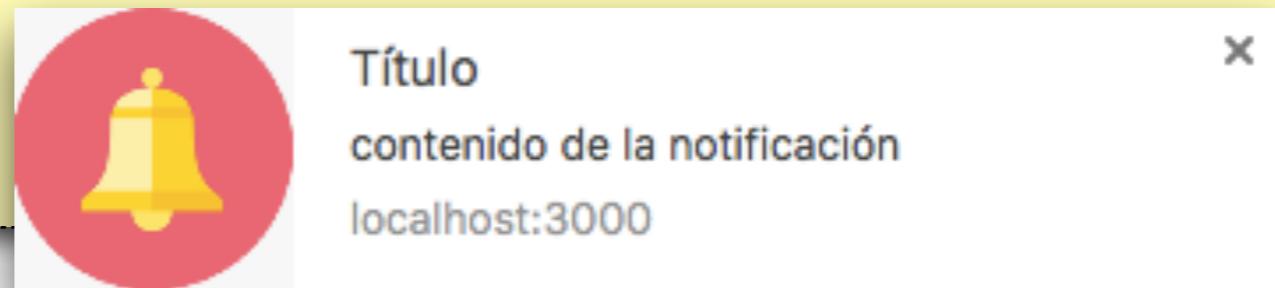
Vayamos por partes



Notifications API

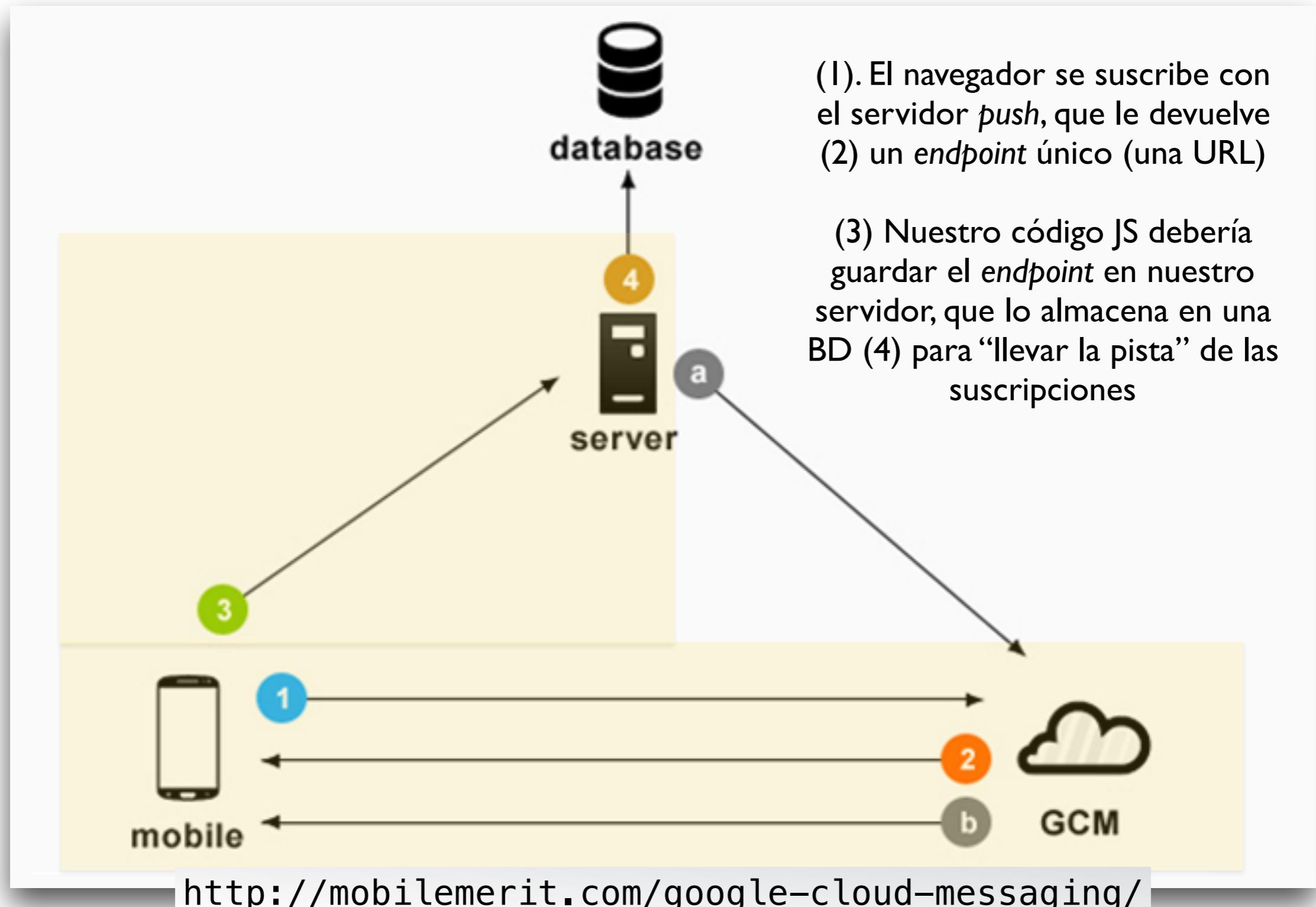
- **Recibir** notificaciones push (no son más que eventos)
- **Mostrar** notificaciones: pueden tener texto, iconos, imágenes, vibrar,...
- Se hace en un service worker

```
self.addEventListener('push', function(notificacion) {  
    self.registration.showNotification('Título', {  
        body: 'contenido de la notificación',  
        icon: 'icono.png'  
    });  
});
```



Tutorial: <https://developers.google.com/web/fundamentals/push-notifications/display-a-notification>

Suscripción



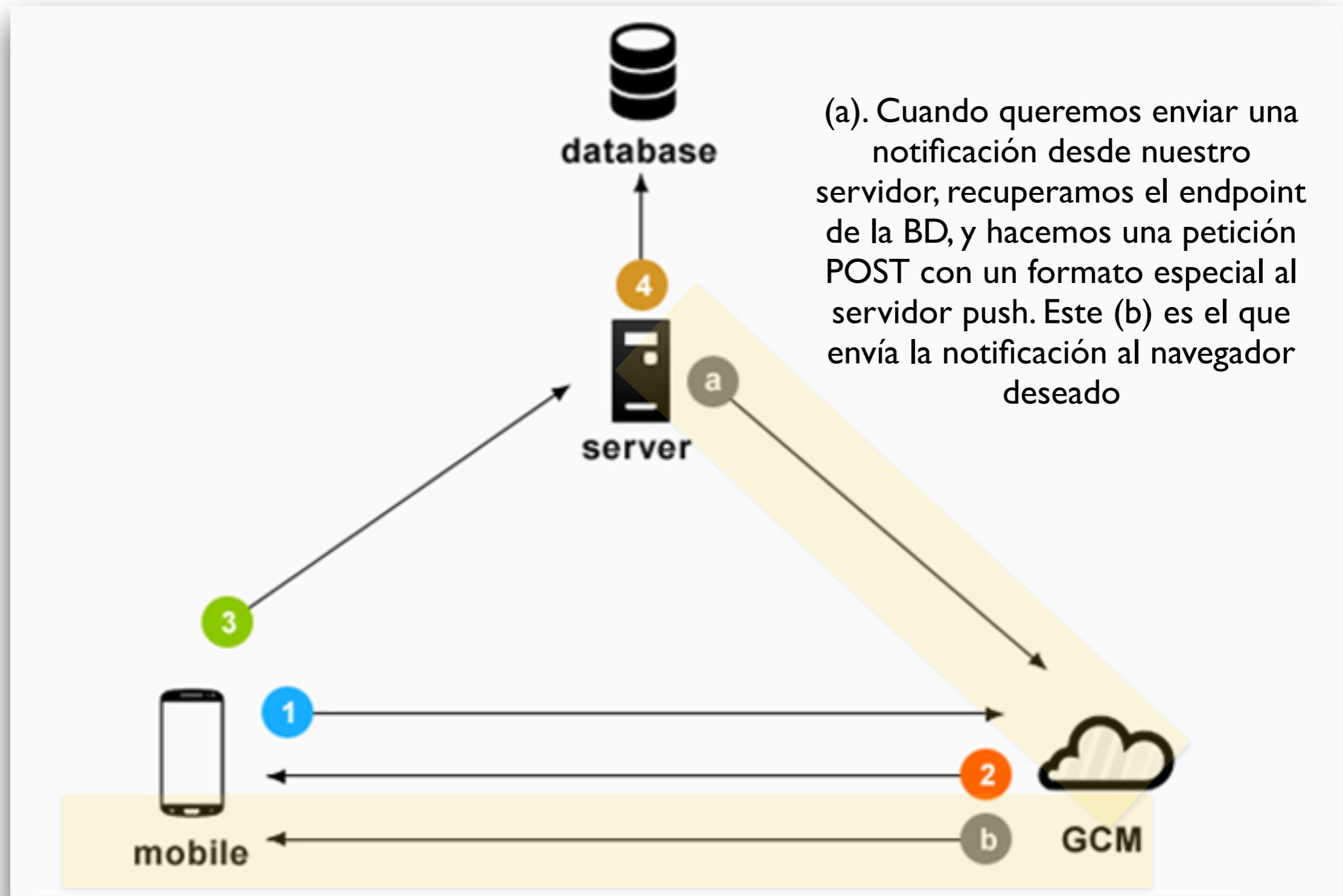
Nota: en la actualidad en Chrome se usa Firebase Cloud Messaging en lugar de Google Cloud Messaging, pero el diagrama sigue igual

Suscripción a notificaciones

- Como hemos dicho, cada suscripción en cada dispositivo tiene un **endpoint** único
- **Push API**: para suscribirse/desuscribirse o ver el estado de la suscripción. Necesita de un *service worker*

```
navigator.serviceWorker.ready
  .then(function(registration) {
    return registration.pushManager.subscribe();
})
  .then(function(suscripcion) {
    //este sería el endpoint
    console.log(suscripcion.endpoint)
    //método propio que enviaría el endpoint a nuestro servidor
    guardarSuscripcionEnNuestroServidor(suscripcion.endpoint);
})
```

Envío de notificaciones

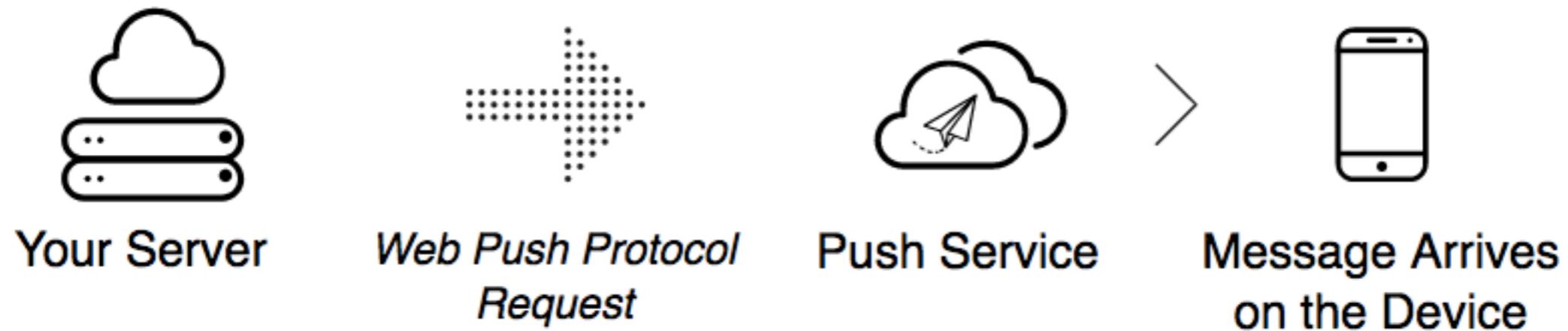


(a). Cuando queremos enviar una notificación desde nuestro servidor, recuperamos el endpoint de la BD, y hacemos una petición POST con un formato especial al servidor push. Este (b) es el que envía la notificación al navegador deseado

Nota: en la actualidad en Chrome se usa Firebase Cloud Messaging en lugar de Google Cloud Messaging, pero el diagrama sigue igual

Envío de notificaciones

- El envío en sí lo hace el servidor *push* de la plataforma. En el caso de Android, **Firebase Cloud Messaging**
- **Nuestro servidor** le pide al servidor push de la plataforma que envíe la notificación, enviando una petición HTTP según el “web push protocol”

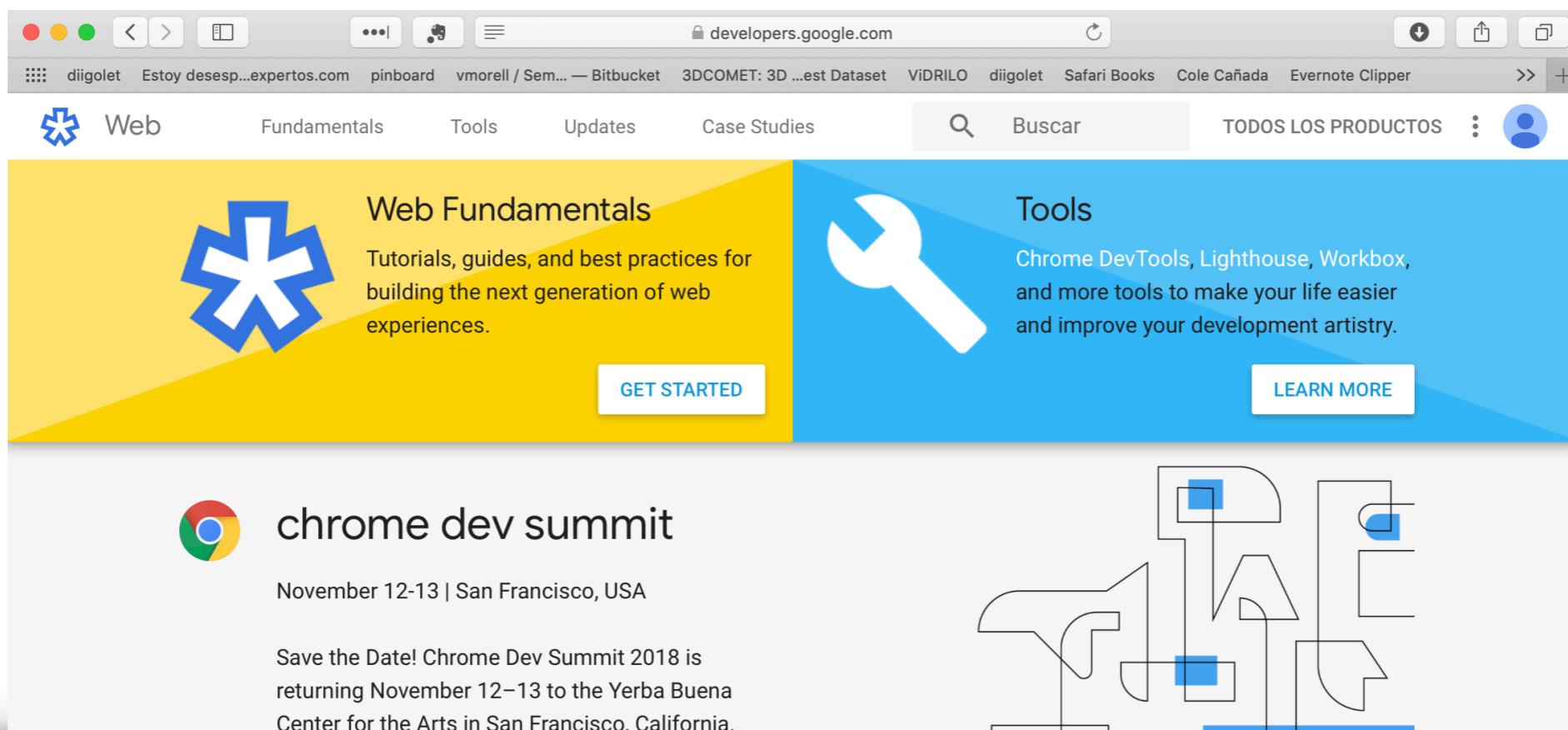


Tutoriales de notificaciones push

- Bastante detallado y cubre todos los aspectos: [https://
developers.google.com/web/fundamentals/push-notifications/](https://developers.google.com/web/fundamentals/push-notifications/)
- Ejemplo de código funcionando: [https://github.com/gauntface/
simple-push-demo](https://github.com/gauntface/simple-push-demo)

Referencias sobre PWA

- <https://developers.google.com/web/fundamentals/> excelentes tutoriales de los “evangelistas” de Google (**algunos en español**), no solo sobre PWAs sino también muchas otras tecnologías modernas en el cliente (web payments, device motion,...)



- <https://medium.freecodecamp.org/progressive-web-apps-101-the-what-why-and-how-4aa5e9065ac2> Tutorial en 2 partes (esta es la primera)