

**Desarrollo en dispositivos móviles**

# **Tema 5:**

# **Progressive Web**

# **Apps**

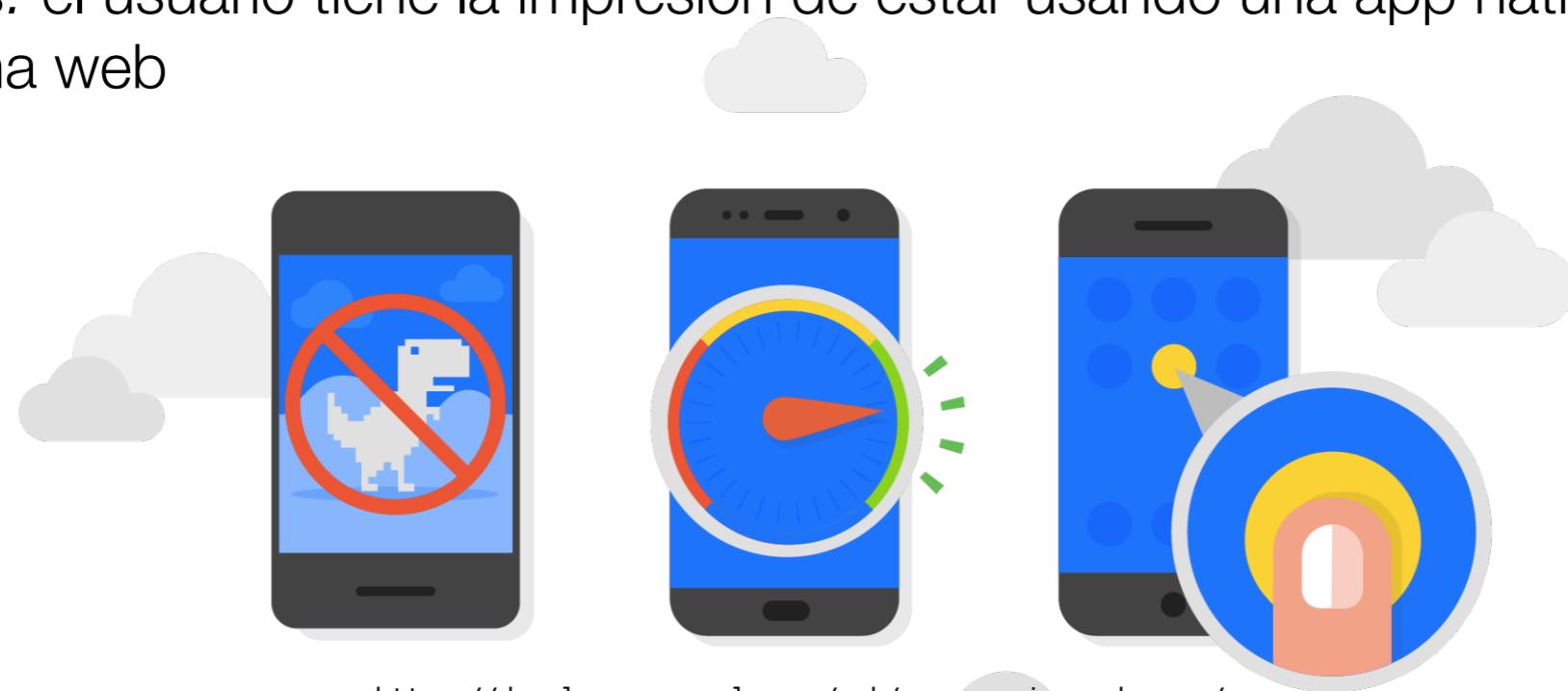
**Desarrollo en dispositivos móviles**  
**Tema 5: Progressive Web Apps**

**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



<https://developers.google.com/web/progressive-web-apps/>

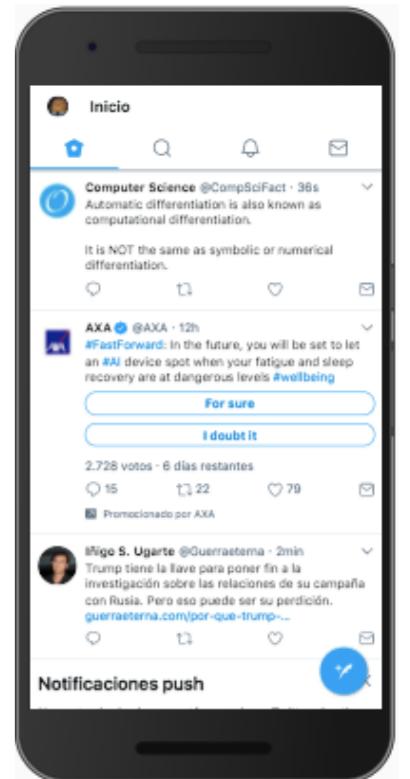
# Ejemplos

- Algunas apps en producción

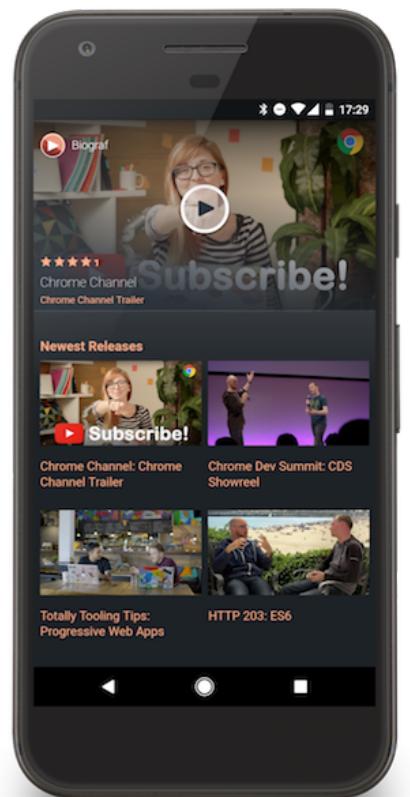
- Uber: <https://m.uber.com>
- AirBnB: <https://www.airbnb.pt/>
- Twitter: <https://mobile.twitter.com>
- Google Maps Go: <https://www.google.com/maps?force=pwa&source=mlpwa>
- Casos de estudio: <https://developers.google.com/web/showcase/2017/>
- Directorio de PWAs: <https://pwa-directory.appspot.com>

- Ejemplos “educativos”, con código fuente

- **Sample media PWA:** <https://github.com/GoogleChromeLabs/sample-media-pwa>
- **Voice memos:** <https://aerotwist.com/blog/voice-memos/>



**Twitter** <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

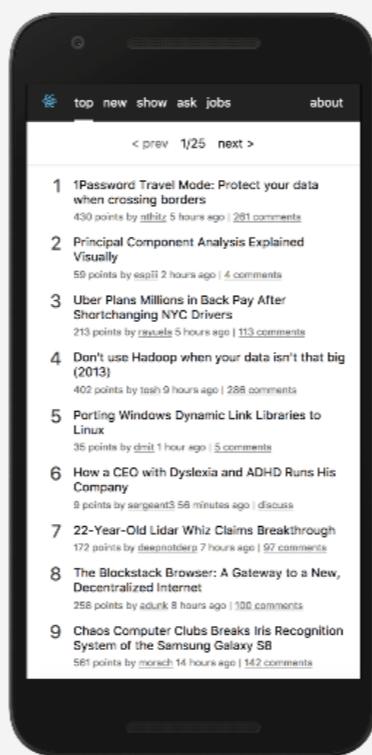
**Hosting:** Webfaction + Cloudflare

**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)

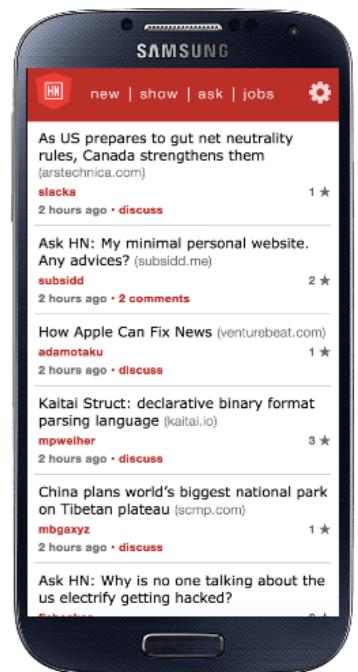
**Hosting:** Firebase

**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 la interfaz del 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 de Google para medir si un sitio web cumple estas condiciones

**Desarrollo en dispositivos móviles**  
**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

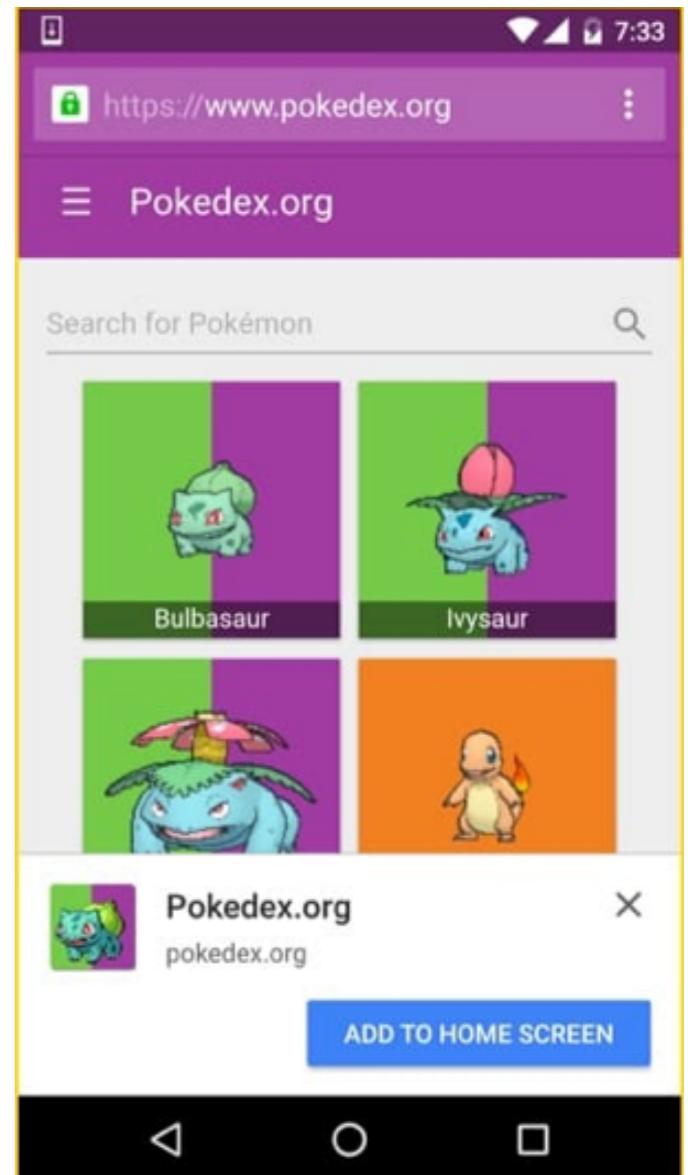
## What are the criteria?

In order for a user to be able to install your Progressive Web App, it needs to meet the following criteria:

- The web app is not already installed.
  - and `prefer_related_applications` is not `true`.
- Meets a user engagement heuristic (previously, the user had to interact with the domain for at least 30 seconds, this is not a requirement anymore)
- Includes a `web app manifest` that includes:
  - `short_name` or `name`
  - `icons` must include a 192px and a 512px sized icons
  - `start_url`
  - `display` must be one of: `fullscreen`, `standalone`, or `minimal-ui`
- Served over **HTTPS** (required for service workers)
- Has registered a `service worker` with a `fetch` event handler

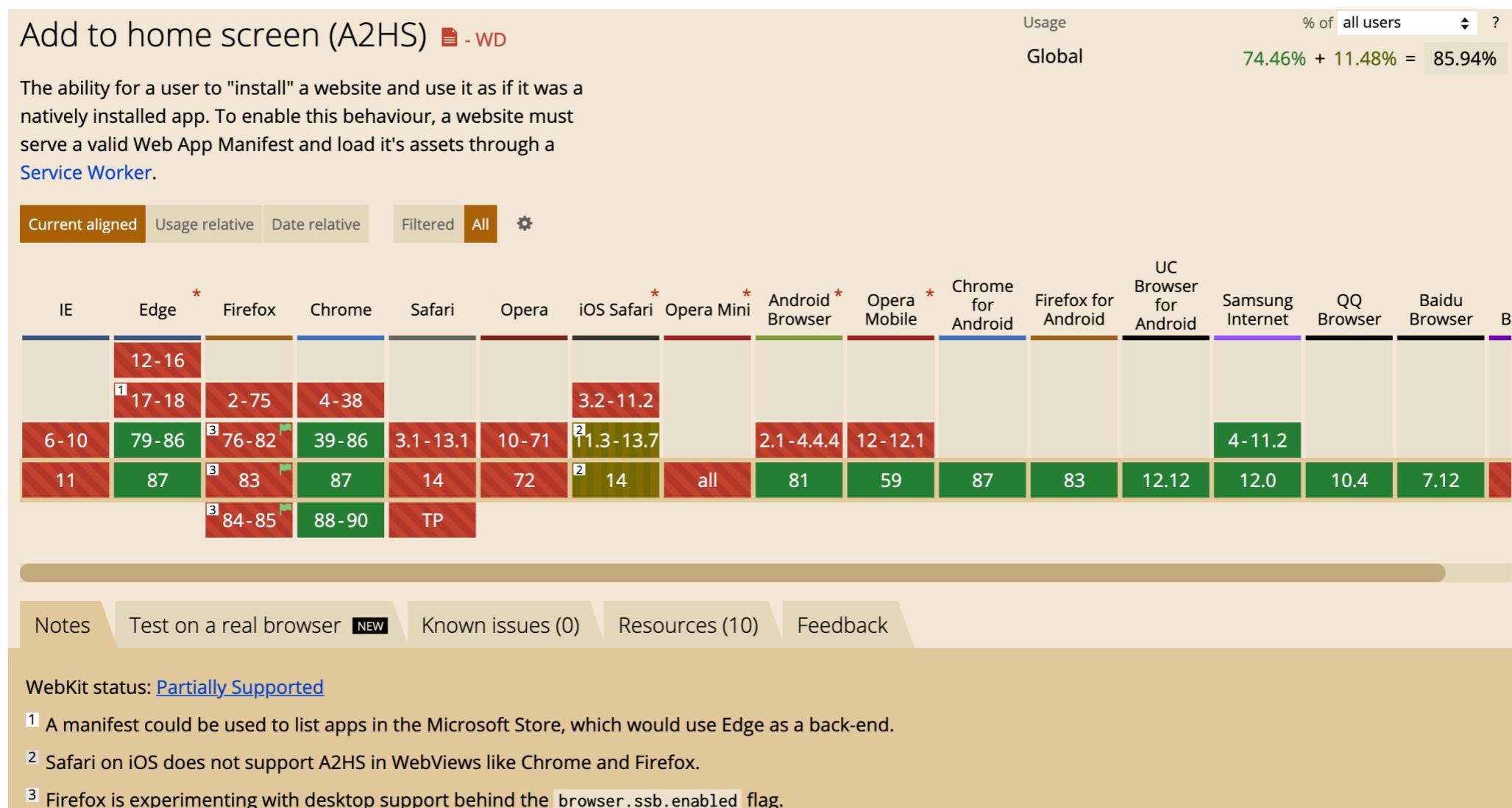
When these criteria are met, will fire a `beforeinstallprompt` event that you can use to prompt the user to install your Progressive Web App, and may show a `mini-info bar`. See [Listen for beforeinstallprompt](#).

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



# Cuestiones prácticas

- Soporte en la mayoría de navegadores móviles. En iOS es parcial (más información)



<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>

- En iOS se pueden obtener funcionalidades más o menos equivalentes con etiquetas `<link>` y `<meta>` (no estándar, es propio de Apple)
- Podéis usar esto y además un *manifest* para mantener compatibilidad

```

<!-- iconos cuando se añade a la pantalla de inicio-->
<link rel="apple-touch-icon" sizes="180x180" href= "icono-iphone.png">
<link rel="apple-touch-icon" sizes="152x152" href="icono-ipad.png">

<!-- nombre de la app, para poner bajo el icono-->
<meta name="apple-mobile-web-app-title" content="Mi App">

<!-- pantalla de splash -->
<link rel="apple-touch-startup-image" href="splash.png">

<!-- app a pantalla completa , sin interfaz de navegador -->
<meta name="apple-mobile-web-app-capable" content="yes">

```

Más info sobre estas etiquetas: <https://developer.apple.com/library/content/documentation/AppleApplications/Reference/SafariWebContent/ConfiguringWebApplications/ConfiguringWebApplications.html>

Sobre los tamaños de iconos en iOS: <https://developer.apple.com/ios/human-interface-guidelines/graphics/app-icon/>

**Desarrollo en dispositivos móviles**  
**Tema 5: Progressive Web Apps**

**5.3.**

**Funcionando offline:**  
**Service Workers**

# Lo que queremos evitar



## No internet

Try:

- Checking the network cables, modem, and router
- Reconnecting to Wi-Fi
- Running Windows Network Diagnostics

DNS\_PROBE\_FINISHED\_NO\_INTERNET

# Service Workers

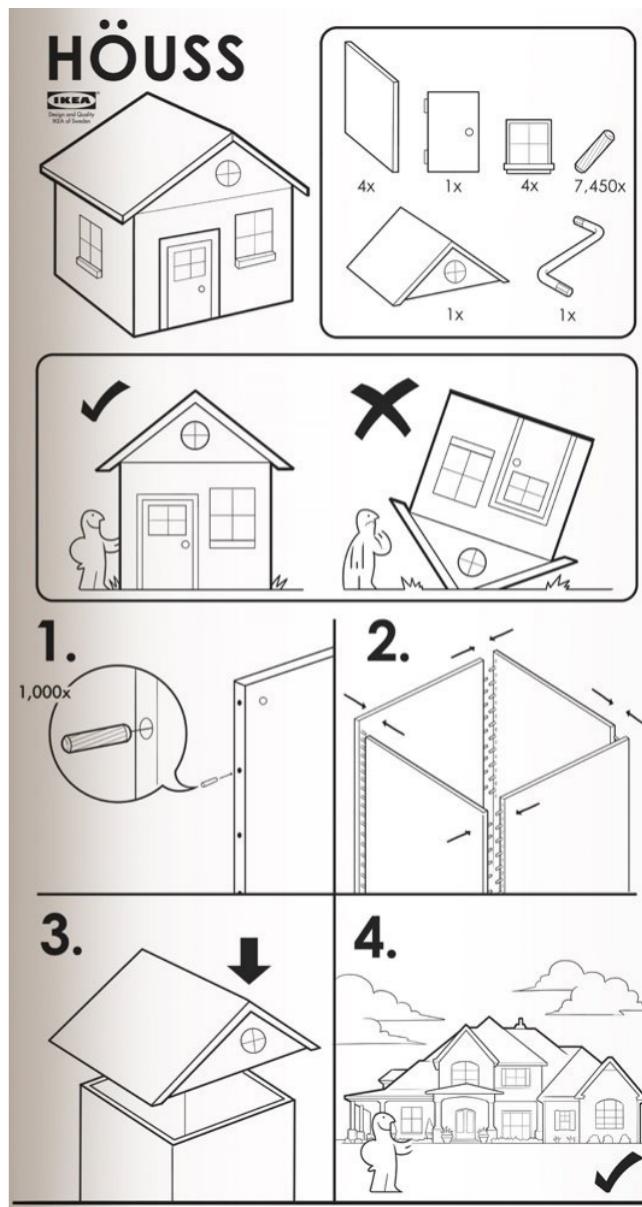
- Los “Javascript workers” son scripts que se ejecutan **en segundo plano, en otro hilo** independiente de la página “principal”. En general se usan para tareas computacionalmente costosas
- Los **Service Workers** son un tipo especial de worker. Pueden detectar ciertos eventos interesantes
  - `fetch`: 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

# 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 para permitir o no acceso
  - Añadir automáticamente credenciales a las peticiones AJAX (p.ej. tokens JWT)
  - ...
- Notificaciones push



Todas **estas funcionalidades no están directamente implementadas en el API de Service Workers**, hay que escribir código JS adicional (ganamos flexibilidad a cambio de complejidad de uso)

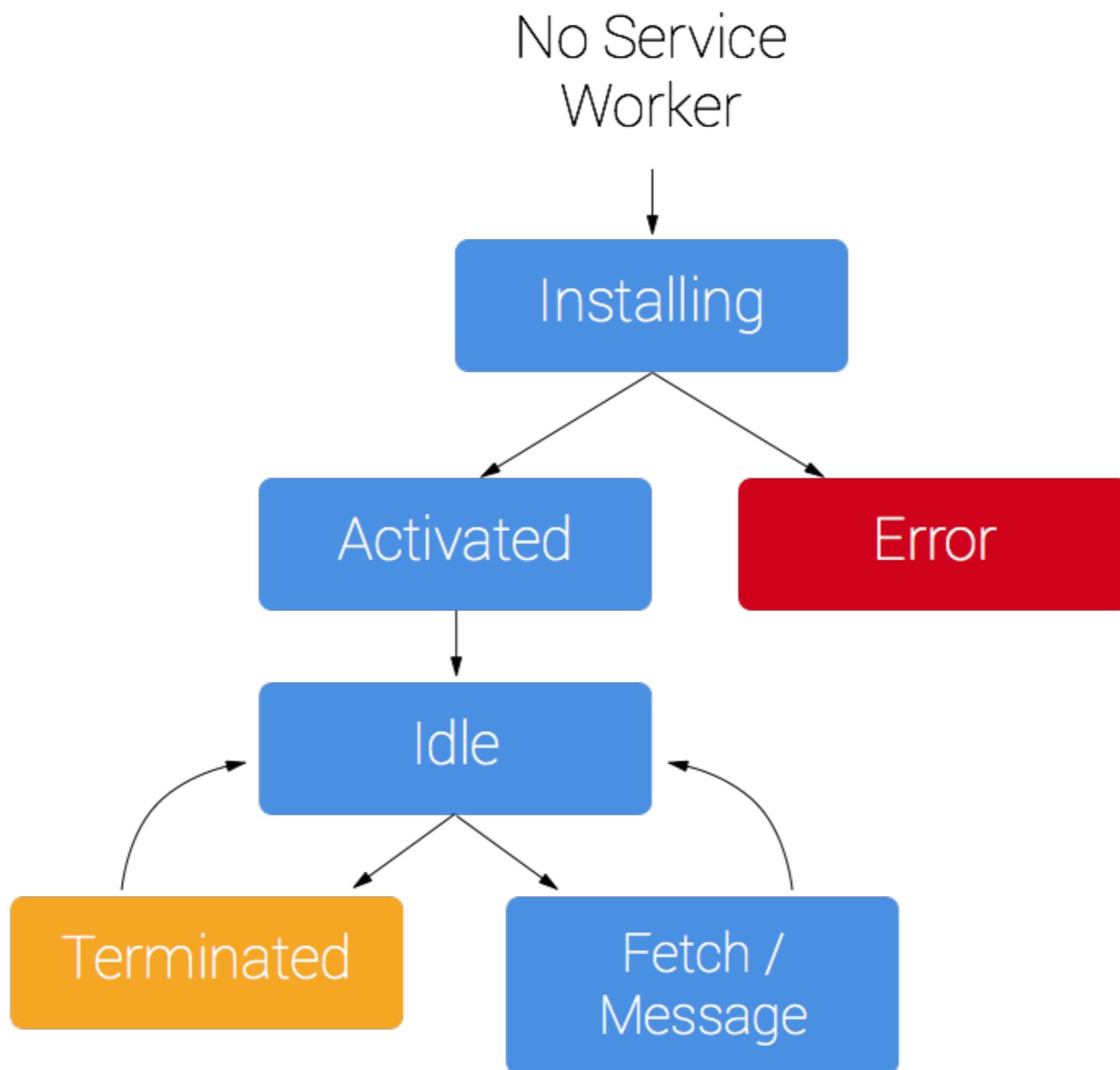


# Registro de un SW

El service worker normalmente está en un fichero aparte (en el ejemplo en “service-worker.js”), lo registramos desde nuestro JS “principal”

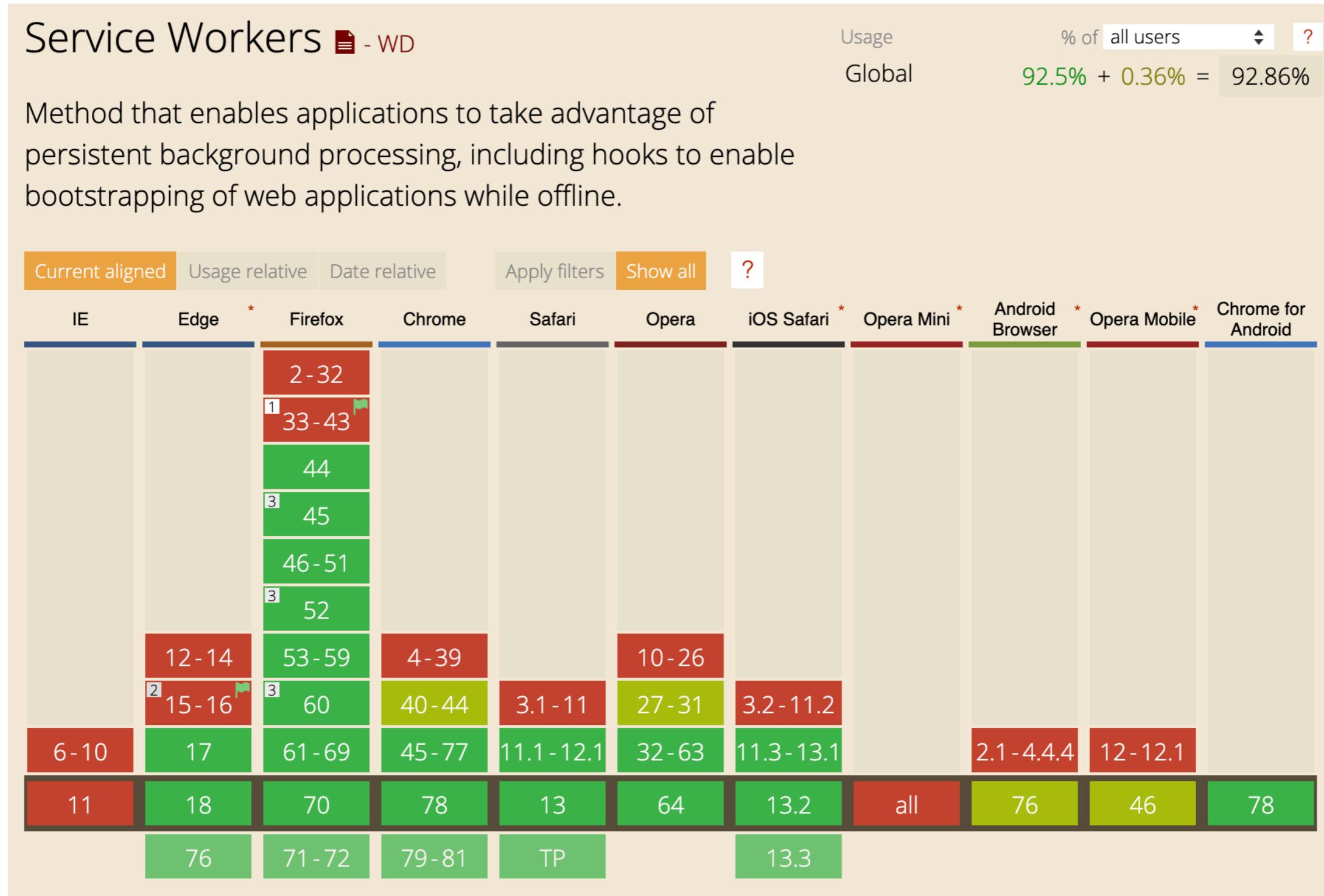
```
if ('serviceWorker' in navigator) {      //comprobamos que el API
  navigator.serviceWorker                    //está soportado
    .register('service-worker.js')           //registro
    .then(function(resultado){              //si registro OK...
      console.log("registro SW ok");
    });
}
else {
  console.log("Service Workers no soportados");
}
```

# Ciclo de vida



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

# Soporte actual de los SW (nov '19)



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

# Eventos accesibles a un SW

No puede acceder al DOM ni a los eventos “normales” de click, etc.

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

//Podemos atender a eventos de nuestro 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')
  })
```

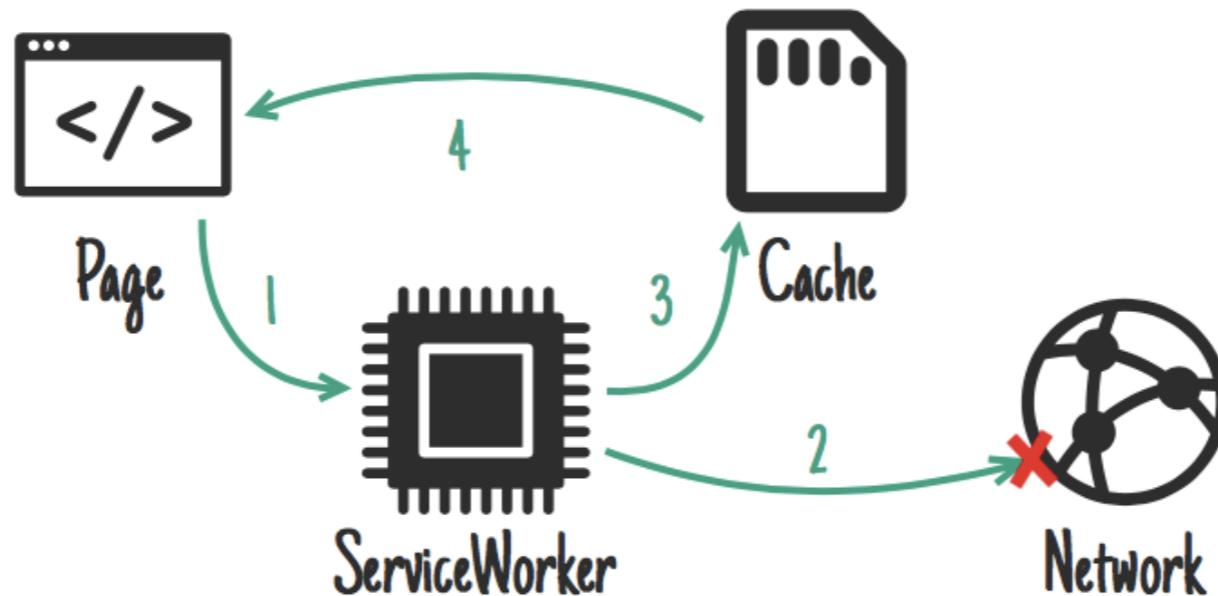
# Ejemplo: crear cache y añadirle recursos

```
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>

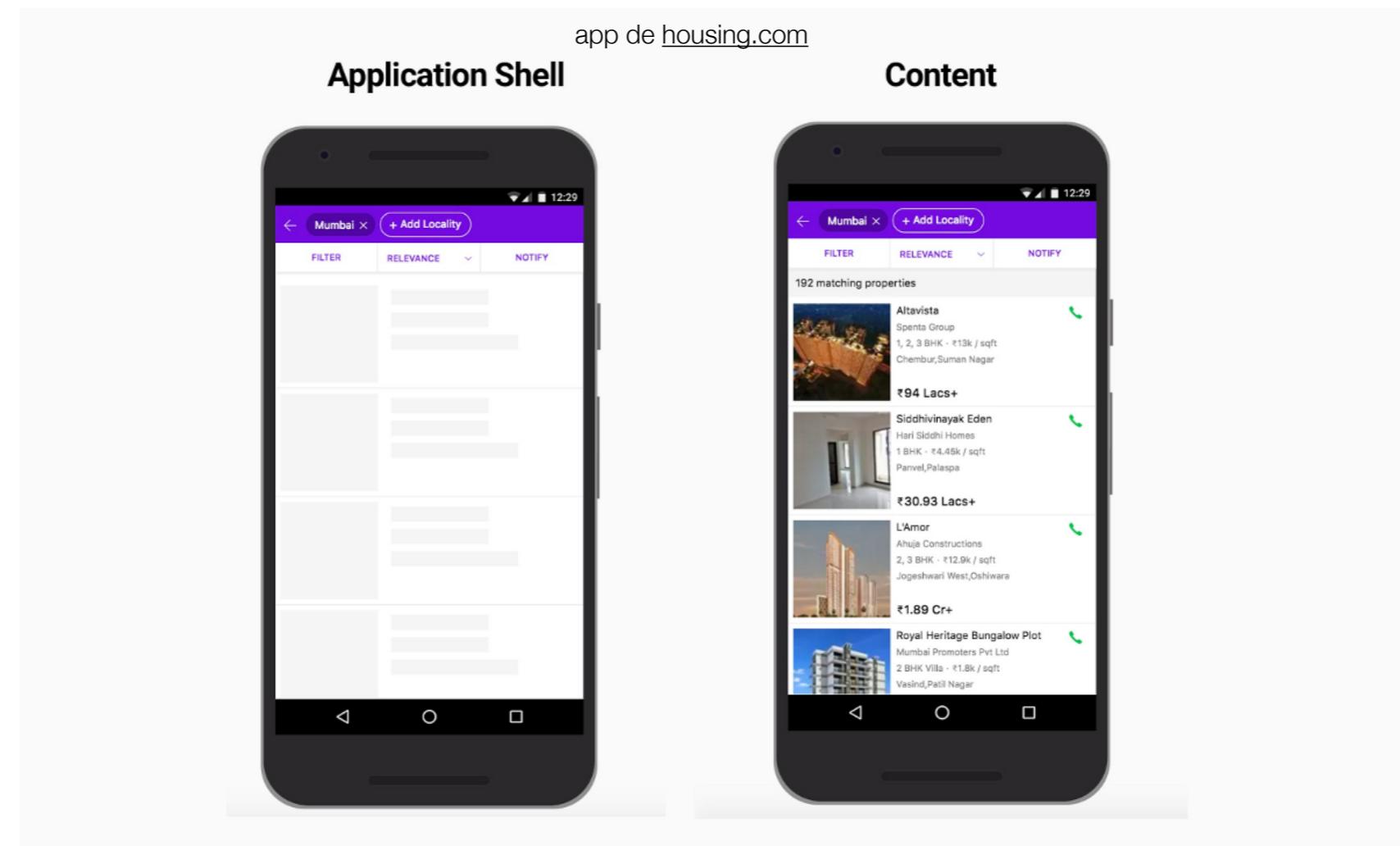
```
self.addEventListener('fetch', function(event) {  
    //respondWith nos permite cambiar la respuesta a devolver ante  
    //una petición HTTP hecha con fetch  
    event.respondWith(  
        fetch(event.request).catch(function() {  
            return caches.match(event.request);  
        })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

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

# 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 más tarde 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 21)
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) {  
        if (response)  
          return response  
        else //si el dato no estaba en cache, match se resuelve a undefined  
          return fetch(event.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**
- Hay varios sitios con patrones o recetas de uso
  - **Service Workers cookbook:** <https://serviceworke.rs/>
  - **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(),  
);
```

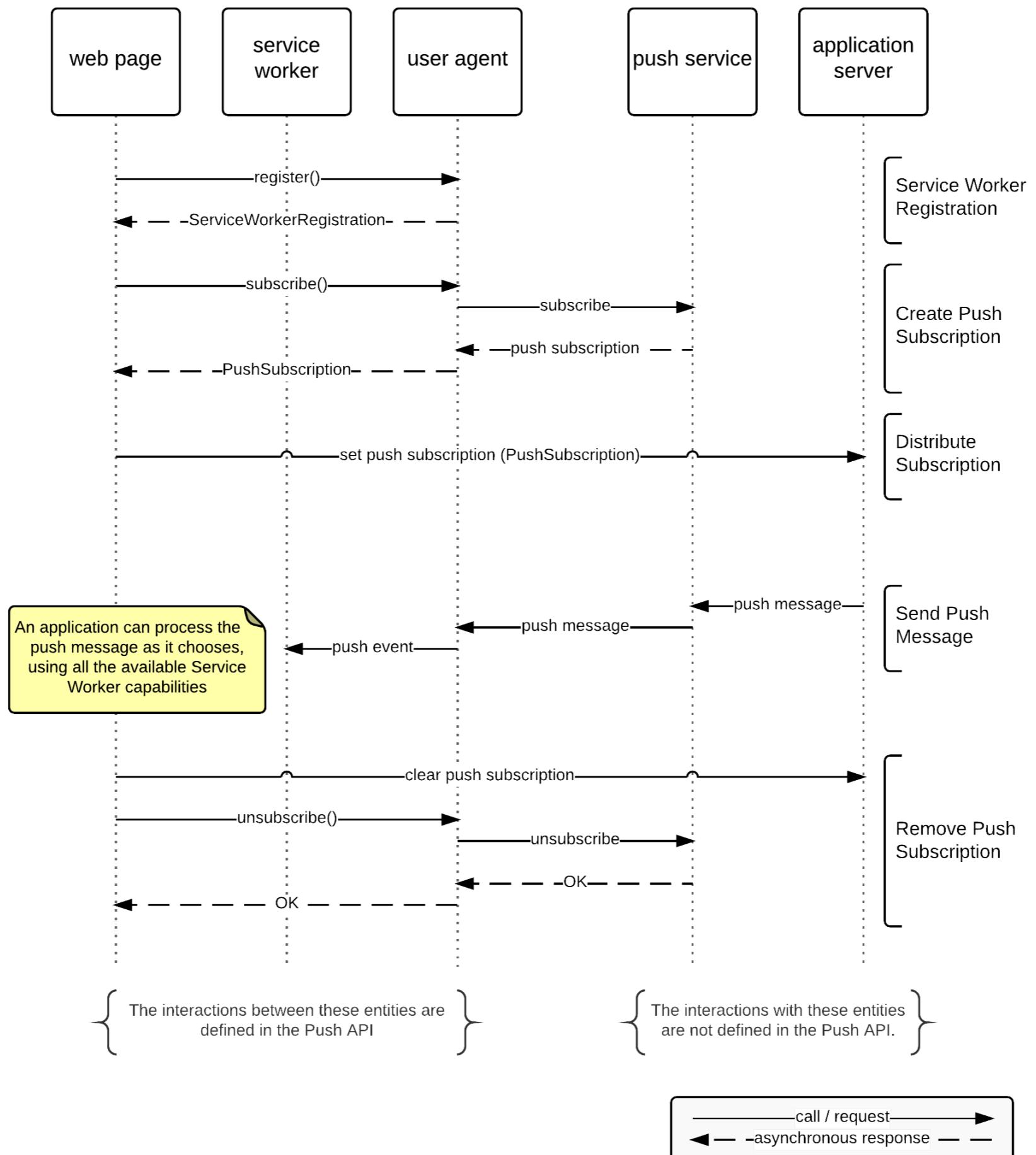
**Desarrollo en dispositivos móviles**  
**Tema 5: Progressive Web Apps**

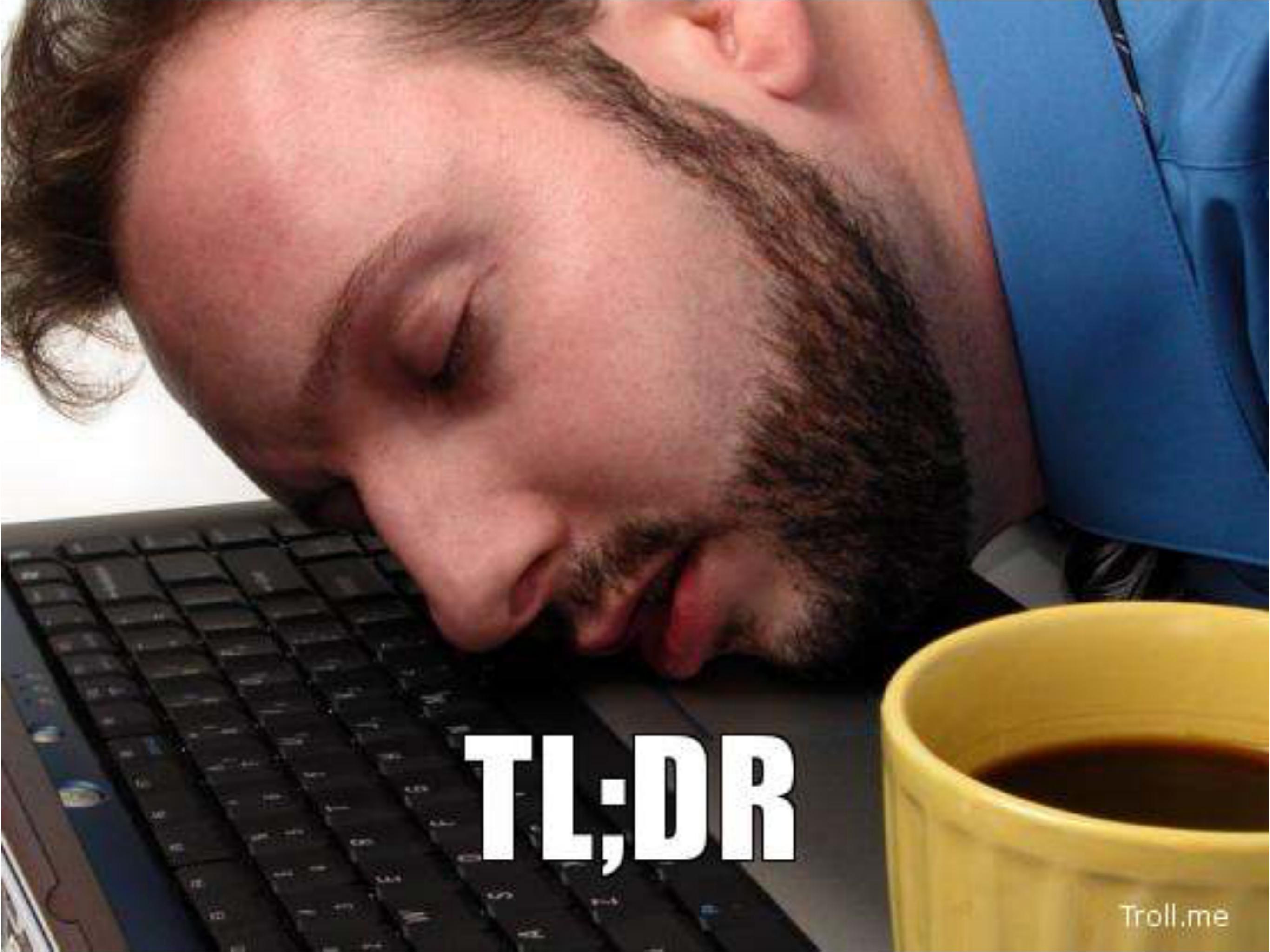
**5.4**  
**“Enganchando” al  
usuario:**

# Web push

- Como todos sabéis 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.

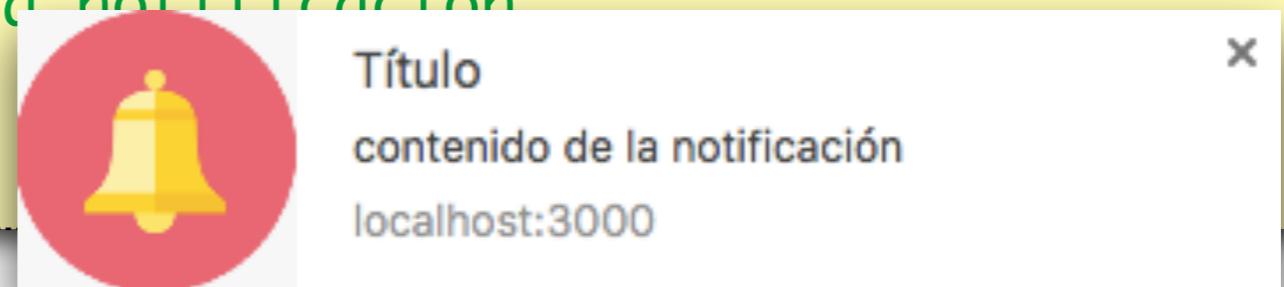
**TL;DR**

Troll.me

# APIs básicos

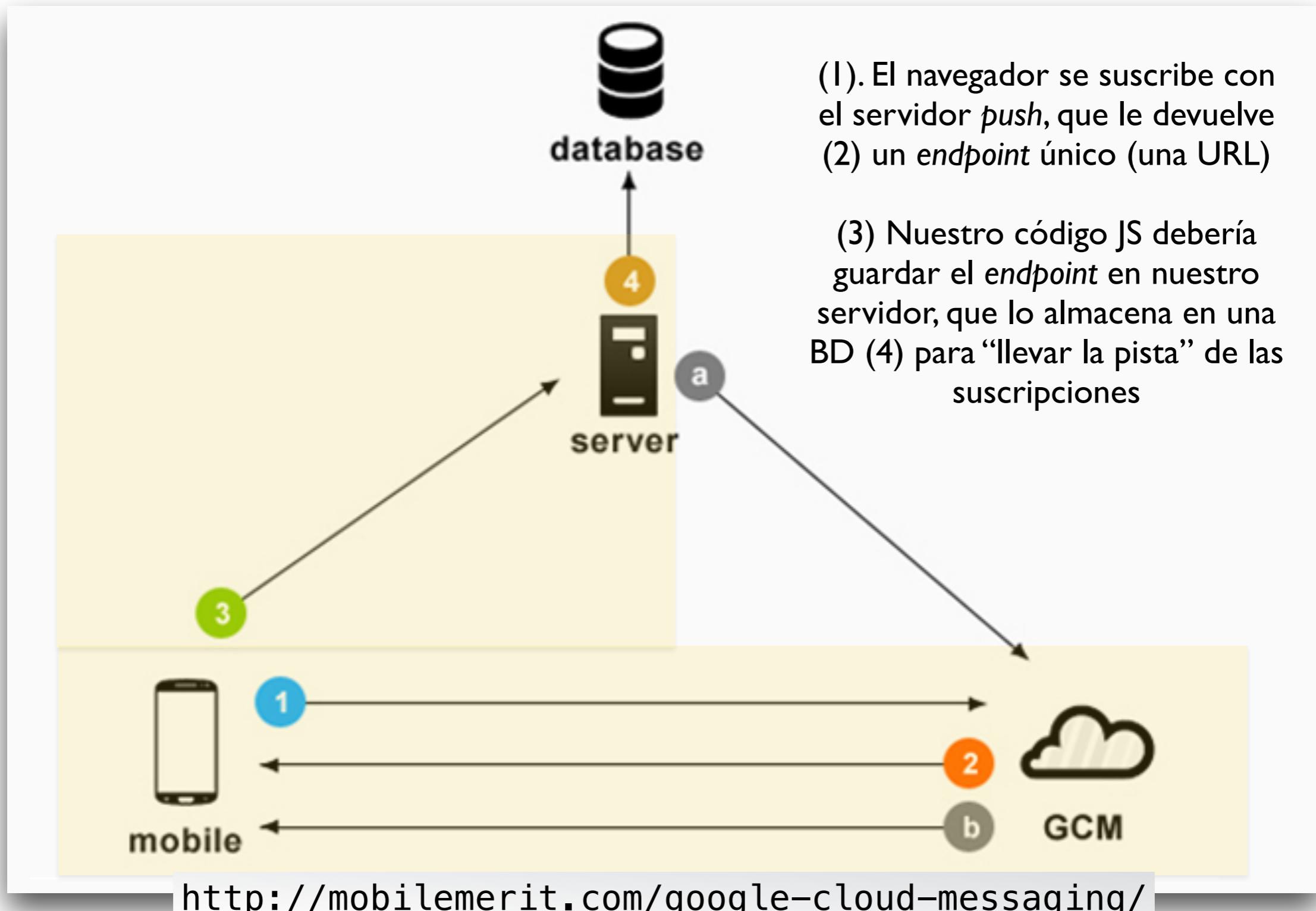
- **Push API:** *recibir* notificaciones push (no son más que eventos)
- **Notifications API:** *mostrar* notificaciones: pueden tener texto, iconos, imágenes, vibrar,...
- Todo esto se hace en un *service worker* para que funcione aunque el navegador esté cerrado

```
//Esto debe estar 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



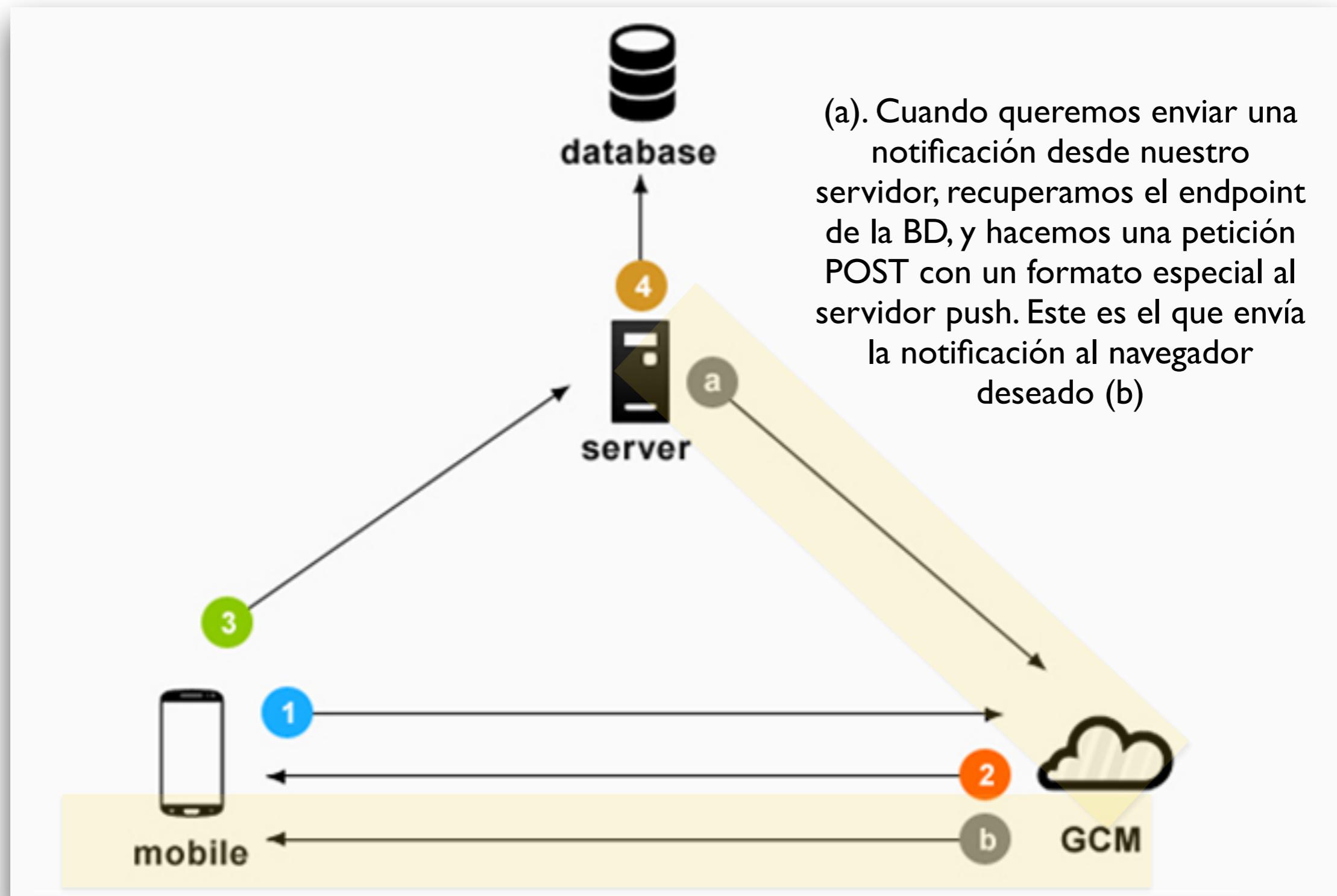
Nota: en la actualidad, en Chrome se usa Firebase Cloud Messaging en lugar de Google Cloud Messaging, pero el diagrama sigue igual

# Suscripción en código

- Para suscribirse/desuscribirse o ver el estado de la suscripción usamos el **Push API** (*necesita de un service worker*)
- Como hemos dicho, cada suscripción en cada dispositivo tiene un **endpoint** único (una URL con un identificador “empotrado”)
- Tenemos que guardar este **endpoint** en algún sitio (típicamente en nuestro servidor, ya que será el que dispare las notificaciones push)

```
//Esto está en la página principal
//necesitamos acceder al objeto "registration" del service worker.
//Esta es una forma sencilla de obtenerlo
navigator.serviceWorker.ready      //cuando el service worker esté listo...
  .then(function(registration) {
    return registration.pushManager.subscribe(); //suscríbete a las notif.
  })
  .then(function(suscripcion) { //si la suscripción ha tenido éxito
    //el endpoint está en la propiedad del mismo nombre
    console.log(suscripcion.endpoint)
    //método propio que enviaría el endpoint a nuestro servidor
    guardarSuscripcionEnNuestroServidor(suscripcion.endpoint);
  })
}
```

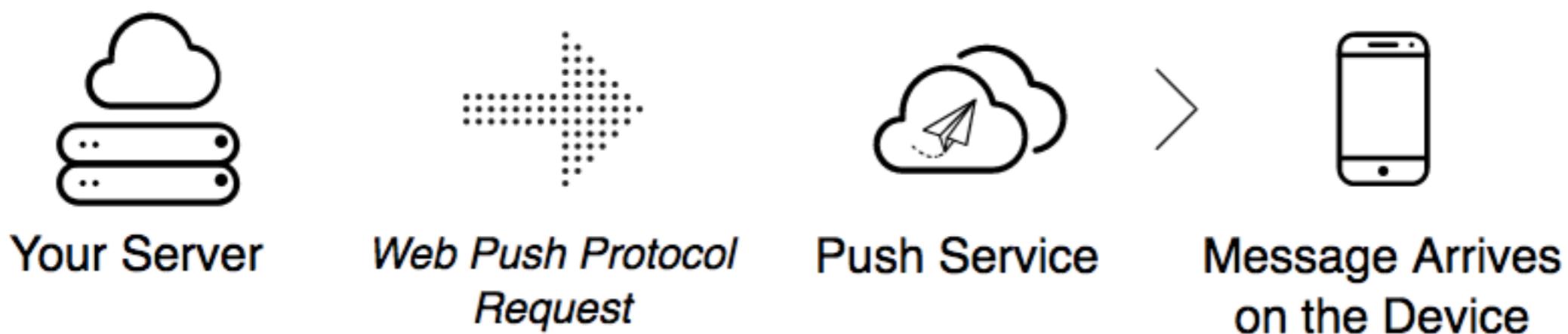
# Envío de notificaciones



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 Android con Chrome, [Firebase Cloud Messaging](#)
  - En Android con Firefox, un servidor llamado [Autopush](#) en push.services.mozilla.com
- **Nuestro servidor** le pide al servidor push de la plataforma que envíe la notificación, enviando una petición HTTP con un formato especial que siga el “[web push protocol](#)” (no es trivial, por ejemplo hay que cifrar el *payload*, para evitar que el servidor de push pueda acceder a él)



# Tutoriales de notificaciones push

- Tutoriales:
  - <https://developers.google.com/web/fundamentals/push-notifications/>
  - <https://web-push-book.gauntface.com/>
- Ejemplo de código funcionando: <https://github.com/gauntface/simple-push-demo>

# Referencias sobre PWAs

- <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,...)
- En <http://learning.oreilly.com> tenéis bastantes libros/videos sobre PWAs
- <https://pwafire.org/>

