Aplicaciones Web Progresivas

Iván Toro
ivan.toro@correounivalle.edu.co

EISC, Universidad del Valle. Santiago de Cali, Colombia
20 de Septiembre del 2019

Resumen

El presente informe evidencia el desarrollo de una aplicación web progresiva, utilizando diferentes tecnologías y conceptos como son: app shell model, web app manifest file y service workers. Inicialmente se describen los diferentes enfoques al desarrollar aplicaciones para dispositivos móviles y sus ventajas y desventajas, así como los fundamentos en los que se basa el desarrollo de las aplicaciones web progresivas y las tecnologías y conceptos utilizados para esto. Por último se describe el trabajo realizado y se presentan las conclusiones.

Introducción

Las aplicaciones web progresivas (PWA, por sus siglas en inglés: Progressive Web Apps) son aplicaciones web con características similares a las aplicaciones nativas, tanto en dispositivos móviles como smartphones y tablets hasta en computadores como laptops y desktops, combinando experiencias rápidas, integradas, confiables y atractivas.

Las PWA difuminan la barrera entre la web y las aplicaciones nativas, realizando tareas que generalmente sólo estas últimas podían llevar a cabo y manteniendo la portabilidad y accesibilidad propias de las aplicaciones web.

Estado del arte

Las PWA son la evolución de las aplicaciones web en la búsqueda de brindar experiencias de usuario más completas, y en la actualidad se presentan como una alternativa muy atractiva para desarrollar aplicaciones accesibles desde diferentes plataformas.

Para desarrollar aplicaciones actualmente existen tres diferentes alternativas, el desarrollo nativo, el desarrollo multiplataforma y el desarrollo web, cada una con sus respectivas ventajas y desventajas.

Desarrollo nativo

En este tipo de desarrollo las aplicaciones son llamadas: aplicaciones nativas.

Las aplicaciones nativas son desarrolladas específicamente desarrolladas para el sistema operativo en el que se ejecutan, por lo tanto se debe realizar un desarrollo independiente por cada plataforma en el que se desee se ejecute la aplicación.

Para el desarrollo se utiliza el Kit de Desarrollo de Software (SDK, por sus siglas en inglés: Software Development Kit) que cada sistema operativo provee.

En el caso de dispositivos móviles los sistemas operativos más utilizados son: Android y iOS. En el caso de Android los lenguajes de programación utilizados para el desarrollo nativo son Java y Kotlin, por otro lado en iOS lo son Objective-C y Swift.

Ventajas de las aplicaciones nativas

- Al utilizar un lenguaje de programación nativo se obtienen un alto rendimiento del dispositivo.
- No requiere conexión a internet.
- Se tiene acceso a las funcionalidades del hardware de los dispositivos.
- Se respeta de forma más precisa las líneas de diseño de UI propias de cada plataforma.

Desventajas de las aplicaciones nativas

- Al ser desarrollos independientes por cada plataforma se generan más costos de desarrollo y mantenimiento.
- La distribución de este tipo de aplicaciones es por medio de una tienda de aplicaciones. Play Store en Android y App Store en iOS.
- Requieren instalación.

Desarrollo multiplataforma

El desarrollo multiplataforma brindan la posibilidad de generar una aplicación para cada plataforma con una misma base de código.

Existen 2 aproximaciones para hacerlo:

- Aplicaciones híbridas (hybrids).
- Aplicaciones puente (bridged).

Aplicaciones híbridas

Son aplicaciones que utilizan tecnologías web estándar: HTML, CSS y JavaScript para el desarrollo de la aplicación. Este tipo de aplicaciones se ejecutan dentro de contenedores dirigidos a cada plataforma, y se basan en enlaces API para acceder al hardware de cada dispositivo, como la cámara, almacenamiento, estado de la red, etc.

Apache Cordova es un framework para desarrollo de aplicaciones híbridas. También es posible utilizar el framework Ionic, que utiliza Apache Cordova como dependencia.

Ventajas de las aplicaciones híbridas

- Una única base de código.

Desventajas de las aplicaciones híbridas

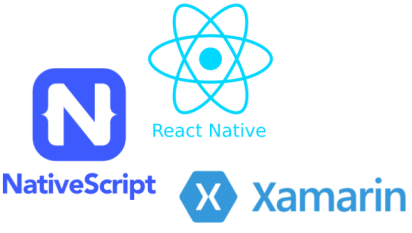
- El rendimiento es afectado.

Aplicaciones puente

Las aplicaciones puente son desarrolladas con tecnologías web y posteriormente son compiladas a código nativo dependiendo de la plataforma. Adicionalmente, código nativo se puede integrar con estas aplicaciones.

Debido a que las aplicaciones puente son compiladas estas proporcionan un mejor rendimiento que las aplicaciones híbridas.

React Native, Native Script y Xamarin son los principales exponentes de este tipo de aplicaciones.



Ventajas de las aplicaciones puente

- Una única base de código.

- Mejor rendimiento que las aplicaciones híbridas.

Desventajas de las aplicaciones puente

- Inmadurez de las tecnologías.

Aplicaciones web progresivas

Las PWA son básicamente páginas web que mediante el uso de tecnologías y APIs del navegador se comportan como aplicaciones nativas.

Por lo tanto las PWA no es una tecnología, sino una serie de tecnologías, estrategias, técnicas y APIs.

Ventajas de las PWA

- No requiere instalación.

Desventajas de las PWA

- No se puede acceder a todas las funcionalidades de hardware de los dispositivos.

Comparación de aproximaciones

Cada una de las anteriores aproximaciones tienen ventajas y desventajas, las cuales deben ser consideradas a nivel de cada proyecto y de esta forma seleccionar la opción que más se adapte el.

En la tabla 1 se detallan algunas las ventajas y desventajas que se presentan en cada aproximación.

Tabla 1. *Comparación de aproximaciones*.

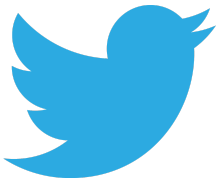
Característica	Aplic. nativa	Aplic. híbrida	Aplic. puente	PWA
Soporte offline	Si	Si	Si	Si
Acceso al hardware	Completo	Limitado y lento	Limitado	Limitado
Instalable	Si	Si	Si	Si
Tamaño de instalación	Grande	Muy grande	Muy grande	Pequeña
Disponible en tiendas	Si	Si	Si	Si
Notificaciones push	Si	Si	Si	Si
Sincronización en segundo plano	Si	Si	Si	Si

Casos de estudio

En la actualidad diferentes empresas han optado por desarrollar su aplicación web como una PWA, algunos ejemplos son: Twitter, Forbes y Alibaba.

Twitter

La red social Twitter por medio de su versión PWA logró incrementar su engagement y redujo el consumo de datos de sus usuarios.



El 80% de los usuarios activos de Twitter acceden a la plataforma por medio de dispositivos móviles, por esta razón Twitter buscaba que su experiencia web fuera lo más atractiva posible.

Una vez implementada la PWA los resultados obtenidos fueron:

- 65% de aumento en páginas por sesión.
- 75% de aumento en los Tweets enviados.
- 20% de disminución en la tasa de rebote.

Forbes

Forbes

La revista Forbes implementó su versión PWA a inicios del 2017, sus resultados fueron:

- 43% de aumento de las sesiones por usuario.
- 20% de aumento en la visibilidad de anuncios.
- 3x de aumento en profundidad de scroll.

Alibaba

Alibaba.com™

Construir experiencias de usuario móviles atractivas es una parte indispensable del éxito de Alibaba. La web móvil es su plataforma principal de sus usuarios en dispositivos móviles, por lo que siempre se han centrado en el diseño y la funcionalidad.

Por medio de su PWA se logró:

- 76% más de conversiones en los navegadores.
- 14% más de usuarios activos mensuales en iOS y 30% en Android.
- Tasa de interacción 4 veces mayor.

Fundamentos de una PWA

Las PWA deben ofrecer experiencias lo más similares posibles a las aplicaciones nativas, por esta razón se debe garantizar que su experiencia de usuario sea:

- **Rápida**: responder las interacciones de los usuarios con animaciones y cargar contenido relevante en milisegundos.
- **Integrada**: es accesible desde el navegador e instalable en diferentes dispositivos aprovechando sus capacidades.
- **Confiable**: carga de forma instantánea en redes de baja calidad e incluso sin conexión a internet.
- **Atractiva (engagement)**: ofrece experiencias que logran que los usuarios regresen a la aplicación.

Rápida

El tiempo que transcurre desde que se inicia una aplicación hasta el instante en que se tiene contenido significativo para el usuario en pantalla no puede ser mayor a 1 segundo.

Integrada

Las experiencias de usuario debe ser lo más fiel posible a la forma en que los usuarios interactúan con el dispositivo con su determinada plataforma.

La instalación de una aplicación nativa puede considerarse como una barrera para algunos usuarios, en el sentido de que requiere un compromiso extra de tiempo, red y almacenamiento.

La web se caracteriza por ser de fácil acceso, es de baja fricción, proporcionando una fácil distribución para las aplicaciones. De esta forma se logra una alta integración con baja fricción.

Confiable

Las aplicaciones deben estar disponibles en redes lentas o intermitentes, e incluso sin conexión a internet.

Atractiva

Una aplicación atractiva va más allá de lo funcional asegurando una experiencia de usuario adecuada que facilite al usuario realizar sus tareas.

Usando notificaciones se mantienen informados a los usuarios, lo que genera su retorno a la aplicación.

Tecnologías y conceptos

En este punto se puede entender que las PWA más que una tecnología, es un conjunto de tecnologías, conceptos y prácticas. Las más importantes son:

- El manifiesto de las aplicaciones web (Web app manifest file).
- El modelo de shell de la aplicación (App shell model).
- Trabajadores de servicio (Services workers).
- APIs de almacenamiento en el navegador.
- Estrategias de cacheo.
- Notificaciones web push.
- Sincronización en segundo plano (Background sync).
- Seguridad por HTTPS.

El manifiesto de las aplicaciones web (Web app manifest file)

El web app manifest file es un archivo JSON que permite definir la apariencia de una PWA en áreas donde normalmente se visualizan las aplicaciones nativas instaladas en un dispositivo, adicionalmente se pueden definir la pantalla inicial al ejecutarse la aplicación y su apariencia al iniciar.

El archivo puede tomar cualquier nombre, por convención se utiliza: *manifest.json*.

El manifiesto es agregado a una página web por medio de una etiqueta *link* en el *header* del documento HTML.

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

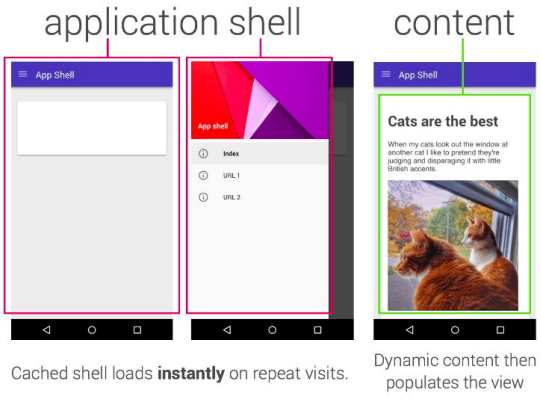
La estructura de un archivo manifiesto básico es la siguiente:

```json
{
    "short_name": "Notes!",
    "name": "Notes!",
    "start_url": "/",
    "icons": [
        {
            "src": "images/note-32x32.png",
            "type": "image/png",
            "sizes": "32x32"
        },
        {
            "src": "images/note-64x64.png",
            "type": "image/png",
            "sizes": "64x64"
        },
        {
            "src": "images/note-128x128.png",
            "type": "image/png",
            "sizes": "128x128"
        },
        {
            "src": "images/note-256x256.png",
            "type": "image/png",
            "sizes": "256x256"
        },
        {
            "src": "images/note-512x512.png",
            "type": "image/png",
            "sizes": "512x512"
        }
    ],
    "background_color": "#009688",
    "theme_color": "#009688",
    "display": "standalone",
    "orientation": "portrait"
}
```

El modelo de shell de la aplicación (App shell model)

App shell model no es una tecnología en sí, sino un modelo o patrón para desarrollar aplicaciones. La idea principal consiste en separar la aplicación en dos: funcionalidad y contenido. Y la funcionalidad y el contenido serán cargados por separado.

En el caso de las PWA el app shell es la mínima cantidad de HTML, CSS y JavaScript requeridos para representar la interfaz de usuario.



El app shell en una PWA es lo que el código publicado en una tienda de aplicaciones lo es a una aplicación tradicional.

Cuando se almacena en caché el app shell asegura un rendimiento instantáneo y de alta confiabilidad para los usuarios con redes deficientes o incluso sin conexión a internet, pues se tiene el código necesario para renderizar la interfaz en el dispositivo.

Los framework y librerías de frontend modernos usados para crear aplicaciones de página única (SPA, por sus siglas en inglés: Single Page Applications) implementan este concepto, pero vale la pena aclarar que app shell es un modelo independiente de frameworks y librerías.

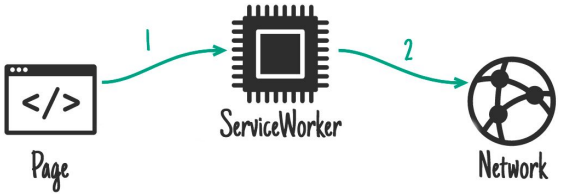
Utilizar app shell implica realizar peticiones asíncronas al servidor para obtener los datos de la aplicación, por lo tanto es necesario definir qué elementos necesitamos sean renderizados inmediatamente al usuario, los archivos relacionados a esos elementos son el app shell. Los datos que son obtenidos posteriormente del servidor son renderizados en el DOM utilizando JavaScript.

Services workers

Las PWA se apoyan principalmente en los service workers, cuyo propósito incluye definir la lógica almacenamiento del contenido en el caché, que estrategias se utilizan para obtener los datos de la red o caché y como y cuando se utilizan diferentes APIs del navegador.

En las aplicaciones web tradicionales se utiliza en el modelo cliente-servidor web, en este caso, sin un servidor no existe la aplicación web.

En una PWA un service worker se sitúa entre el cliente y el servidor, actuando como un proxy de red del lado del cliente. De esta manera, después de la primera visita, el contenido puede estar disponible sin conexión.



Por esta característica las PWA pueden ser cargadas más rápido, pues no importa la velocidad de la red del usuario.

Un service worker se ejecuta en su propio *event loop* (hilo de ejecución en el navegador) y este está separado de la página web, manejando eventos disparados por el navegador o la página web. De esta forma se abre la puerta a funcionalidades que no necesitan interacción de usuario directa del usuario, como por ejemplo: notificaciones web push y sincronización en segundo plano.

Podemos decir que los services workers le dan la posibilidad a las aplicaciones web de vivir fuera del navegador.

En la actualidad los service workers son soportados por la mayoría de navegadores de escritorio y móviles.

Para determinar si un determinado navegador soporta cierta API, en este caso los service worker se puede comprobar de la siguiente manera.

```
if ('serviceWorker' in navigator) {
  window.addEventListener('load', function () {
    navigator.serviceWorker.register('/sw.js').then(function (registration) {
      // Registration was successful
      console.log('ServiceWorker registration successful with scope: ', registration.scope)
    }, function (err) {
      // registration failed :(
      console.log('ServiceWorker registration failed: ', err)
    })
  })
```

Una de las mayores ventajas de utilizar service workers, como se mencionó anteriormente es la habilidad de manejar los eventos ***fetch*** del navegador, es decir, interceptar la peticiones de red y responderlas programáticamente según sea necesario.

Ciclo de vida de los service workers

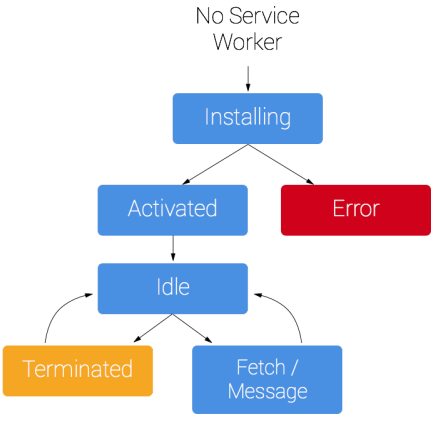
Al service worker ejecutarse en otro *event loop* tiene un ciclo de vida completamente diferente a la página web.

Para instalar un service worker en tu página web necesitar registrarlo en el navegador, este proceso está compuesto por diferentes etapas diseñadas para realizar diferentes tareas, estas etapas son el ciclo de vida de un service worker.

Cuando se instala un service worker se inicia el evento de instalación, en esta etapa lo ideal es almacenar en caché el app shell, si no se puede obtener un archivo o ocurre algún error de otro tipo la instalación fallará y se intentará la próxima vez que se cargue la aplicación.

Después la etapa de instalación sigue la etapa de activación en la cual se debe manejar el cache antiguo, gestionado por anteriores service workers, sí existen. Normalmente el cache antiguo es eliminado para evitar inconsistencia de los datos.

Por último el service worker pasa a controlar todas las páginas dentro de su alcance (scope) una vez se cargue nuevamente la página. Y puede estar en dos estados: terminado, para ahorrar memoria o manejando eventos, por ejemplo: *fetch* o *message*.



Alcance de los service workers (Scope)

El alcance de los service workers está determinado por el scope que se define en el momento que son registrados. Por defecto el scope del service worker serán todas la paginas hermanas e hijas donde se sirve el script.

Por ejemplo: mydomain.co/static/sw.js, controlara las páginas:

- mydomain.co/static/index.html
- mydomain.co/static/price.html
- mydomain.co/static/saas/style.css
- mydomain.co/static/*/*.something

Pero no las páginas:

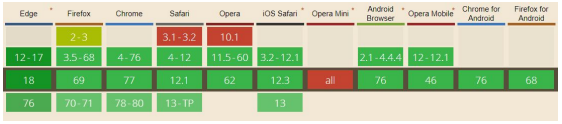
- mydomain.co/index.html
- mydomain.co/js/hello.js

En Chrome, para consultar los service workers instalados e información adicional, como su scope, se puede visitar: chrome://serviceworker-internals/.

APIs de almacenamiento en el navegador

localStorage and sessionStorage

Son propiedades de solo lectura del navegador, es un almacenamiento de tipo llave y valor (key/value) fácil de usar y disponible en casi todos los navegadores.



La principal diferencia entre localStorage y sessionStorage es que sessionStorage los datos persisten sólo en la ventana/tab que los creó y estos son eliminados una vez se cierra la ventana/tab, mientras que con localStorage los datos persisten entre ventanas/tabs con el mismo origen.

Tanto las llaves como los valores siempre son de tipo string.

Una característica de suma importancia para los desarrolladores es que estas APIs son síncronas, lo que significa que bloquean el hilo de ejecución.

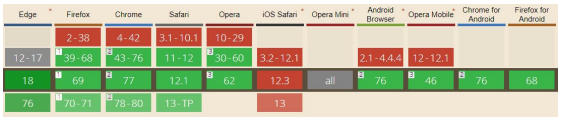
Cache API

Cache API proporciona un mecanismo de almacenamiento para objetos de tipo Request y Response.

Esta API está disponible tanto en los ámbitos de la página web como en el ámbito del service worker.

Es posible tener diferente caches en un mismo dominio, es responsabilidad del desarrollador administrar estas por medio de un nombre único para cada caché.

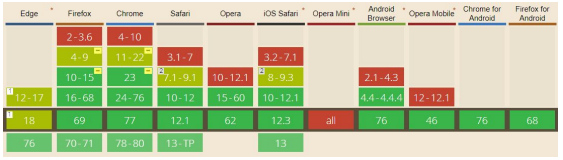
Esta API está disponible en la mayoría de los navegadores modernos.



IndexedDB

Es una API de bajo nivel que ofrece almacenamiento en el navegador de datos estructurados, es decir, datos primitivos, archivos e incluso blobs (binarios de gran tamaño). Adicionalmente usa índices para realizar búsquedas de alto rendimiento.

Esta API es soportada por la mayoría de los navegadores.

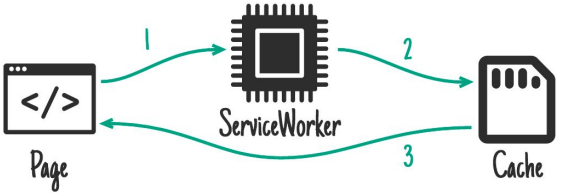


IndexedDB ofrece tanto acceso síncrono y como asíncrono.

Estrategias de cacheo

Las estrategias de cacheo son la metodología que se utiliza en una aplicación para el manejo de cache y peticiones. Existen diferentes estrategias, y la mejor opción dependen del tipo de aplicación o datos que se quieran consultar.

Sólo caché

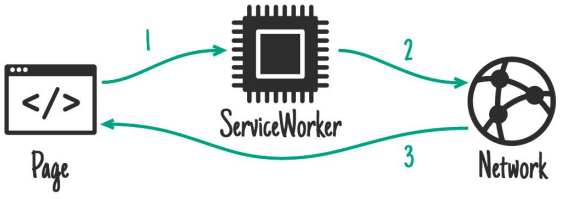


Esta estrategia es ideal para recursos que considera estáticos y que está seguro que se encuentran en caché.

La forma de acceder a estos recursos es la siguiente:

```
self.addEventListener('fetch', function (event) {
  // If a match isn't found in the cache, the response
  // will look like a connection error
  event.respondWith(caches.match(event.request))
})
```

Sólo red

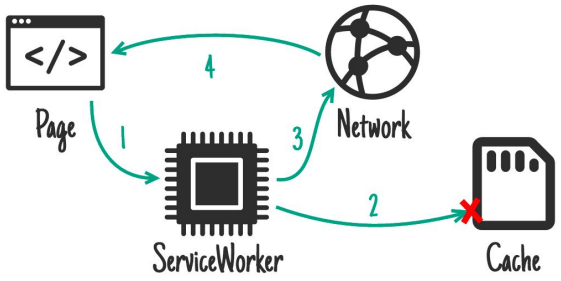


Este caso aunque no es una estrategia de cacheo es una forma de obtener recursos, en este caso son recursos que no pueden tener un equivalente en cache, como por ejemplo pings analiticos y solicitudes POST.

El comportamiento por defecto del navegador es utilizar esta estrategia.

Primero caché sino red

En esta estrategia cuando se ejecuta el evento fech primero se consulta en caché, si no existe el recurso se consulta en la red.



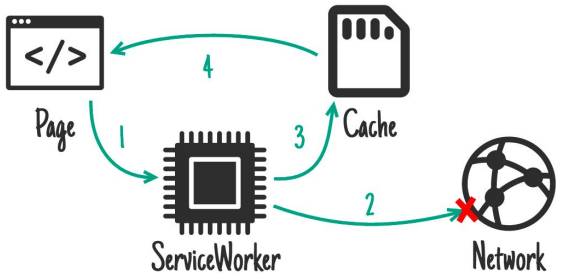
Está estrategias es recomendada para los recursos más comunes, como por ejemplo el app shell. Esta estrategia no es recomendada para la data de la aplicación o recursos que cambien con frecuencia.

La forma de utiliza esta estrategia es la siguiente:

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

Primero red sino caché

En esta estrategia se consulta primero la red, y si la consulta falla se consulta el cache.

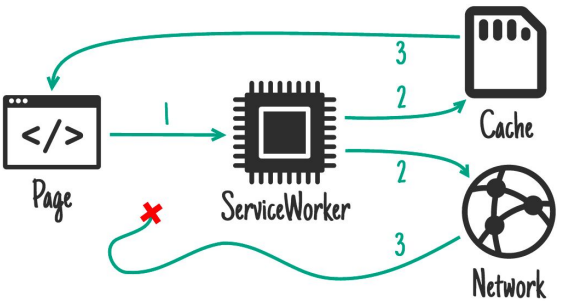


Esta estrategia busca ofrecer siempre el contenido más actualizado, ofreciendo una respuesta desde caché para los usuarios sin conexión a internet.

Para implementar esta estrategia se utiliza el siguiente script:

```
self.addEventListener('fetch', function (event) {
  event.respondWith(
    fetch(event.request).catch(function () {
      return caches.match(event.request)
    })
  )
})
```

Carrera entre caché y red



En este caso, por cada consulta se realizan peticiones en paralelo tanto al caché como a la red, y se utiliza la respuesta que se

complete primero (usualmente caché), adicionalmente cuando el recurso es traído de la red el caché se actualiza, lo que garantiza que la caché esté actualizada para la siguiente petición.

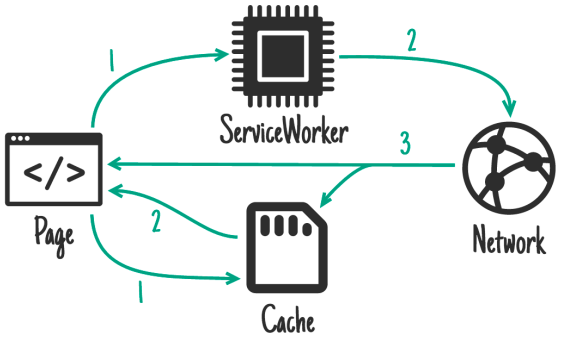
Esta estrategia se puede implementar de la siguiente manera:

```
function promiseAny(promises) {
  return new Promise((resolve, reject) => {
    // make sure promises are all promises
    promises = promises.map(p => Promise.resolve(p));
    // resolve this promise as soon as one resolves
    promises.forEach(p => p.then(resolve));
    // reject if all promises reject
    promises.reduce((a, b) => a.catch(() => b))
      .catch(() => reject(Error("All failed")));
  });
};

self.addEventListener('fetch', function (event) {
  event.respondWith(
    promiseAny([
      caches.match(event.request),
      fetch(event.request)
    ])
  )
})
```

Caché y después red

Al igual que en la estrategia de carrera entre caché y red en esta estrategia se realizan dos peticiones en paralelo, al caché y a la red, la idea es que el caché es utilizado para presentar información al usuario, y cuando la data de red llega la caché y la página son actualizada.



La implementación de esta estrategia consta de dos partes, el script de la página web encargado de realizar la petición y el manejador del service worker.

- Service worker:

```
self.addEventListener('fetch', function (event) {
  event.respondWith(
    caches.open('cache-name').then(function (cache) {
      return fetch(event.request).then(function (response) {
        cache.put(event.request, response.clone())
        return response
      })
    })
  )
})
```

- Página web:

```
var networkDataReceived = false

startSpinner()

// fetch fresh data
var networkUpdate = fetch('/data.json').then(function (response) {
  return response.json()
}).then(function (data) {
  networkDataReceived = true
  updatePage(data)
})

// fetch cached data
caches.match('/data.json').then(function (response) {
  if (!response) throw Error("No data")
  return response.json()
}).then(function (data) {
  // don't overwrite newer network data
  if (!networkDataReceived) {
    updatePage(data)
  }
}).catch(function () {
  // we didn't get cached data, the network is our last hope:
  return networkUpdate
}).catch(showErrorMessage).then(stopSpinner)
```

Cada estrategia es ideal para cada tipo de recurso, por esta razón es trabajo del desarrollador decidir qué estrategia implementar en cada caso, pues se pueden utilizar diferentes estrategias en diferentes peticiones.

Notificaciones web push

Las notificaciones utilizadas en las PWA son una combinación de dos APIs, Notifications API y Push API, la primera sirve para presentar notificaciones similares a las nativas de cada plataforma y la segunda es utilizada para recibir mensajes enviados a ellas desde un servidor.

Por lo anterior podemos definir Web Push como el proceso de enviar notificaciones desde el servidor a un usuario de una PWA.

En necesario que el usuario otorgue el permiso para que una aplicación web pueda enviarle notificaciones. El estado de los permisos se puede comprobar de la siguiente manera.

```
Notification.requestPermission(function (status) {
    console.log('Notification permission status:', status)
})
```

Esta API está disponible en la mayoría de los navegadores modernos.

Sincronización en segundo plano (Background Sync)

Para lograr una experiencia de usuario completamente offline es necesario que el usuario pueda enviar peticiones de tipo POST al servidor incluso sin conexión, por este motivo la sincronización en segundo plano es necesaria.

Utilizando este API, lo que se consigue es que las peticiones de red que fallen sean almacenadas en una cola de peticiones utilizando IndexedDB y en el momento que se restablezca la conexión, estas peticiones serán enviadas.

Para utilizar esta API inicialmente se debe registrar la sincronización de la página y posteriormente se maneja el evento desde el service worker.

```
// Register your service worker
navigator.serviceWorker.register('/sw.js')

// Then later, request a one-off sync
navigator.serviceWorker.ready.then(function (registration) {
  return registration.sync.register('sync')
})
```

```
self.addEventListener('sync', function (event) {
  if (event.tag == 'sync') {
    event.waitUntil(doSomeStuff())
  }
})
```

Sin embargo, esta API en la actualidad es soportada por pocos navegadores.

Seguridad por HTTPS

Los service workers pueden controlar nuestra aplicación web completamente, como anteriormente se presento, por esta razon es indispensable utilizar HTTPS para de esta manera asegurar que el service worker que es instalado en nuestro navegador, no ha sido manipulado en su viaje a través de la red, sin embargo para facilitar el desarrollo también son instalables en *localhost* y sus equivalentes.

Línea base

Los desarrolladores de Google proporcionan un check list con los requerimientos mínimos para que una aplicación web se pueda considerar como una PWA.

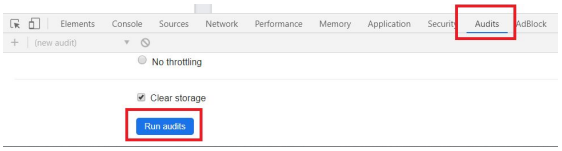
De esta forma los requerimientos que debe cumplir una PWA de referencia son:

- Todo el sitio se sirve a través de HTTPS.
- La aplicación web es responsive, es decir, se adapta en dispositivos de diferentes tamaños de pantalla.
- Todas las URLs involucradas en la aplicación web responden con un HTTP Status 200 incluso sin conexión internet.
- La aplicación se puede agregar a la pantalla de inicio en diferentes dispositivos. Que sea instalable.
- La primera carga de la aplicación no puede superar los 10 segundos en redes 3G lentas.
- La aplicación funciona en por lo menos los navegadores: Chrome, Edge, Firefox y Safari.

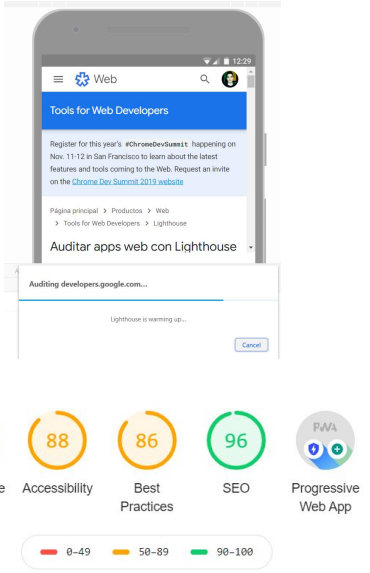
- Las transiciones entre páginas deben ser inmediatas, si se requieren datos de red se debe mostrar un indicador de carga y/o “pantallas de esqueleto”.
- Todas la páginas deben tener una URL única.

Estos requerimientos son comprobados por medio de Lighthouse, la herramienta de auditoría de aplicaciones web de Chrome Devtools.

Para auditar una aplicación web se debe ingresar a Devtools, en Chrome, ir a la pestaña Auditoría y iniciar una Auditoría.



Se ejecutarán diferentes pruebas a la página web y por último se obtendrá un informe detallado.



Aplicación desarrollada

Para el desarrollo de esta práctica se desarrolló una PWA para administrar notas (sticky notes).

El repositorio del proyecto es: https://github.com/ivanmtoroc/notes.

Las herramientas utilizadas para el desarrollo de la aplicación son:

- Django.
- Django Rest Framework.
- Vue.js.
- Workbox.

Workbox es un conjunto de bibliotecas y módulos que facilitan el almacenamiento en caché de activos y aprovechan al máximo las funciones utilizadas para crear PWA.

Workbox facilitó el desarrollo de la PWA pues se utilizaron las estrategias de caché que brinda la librería y su implementación de Background Sync. Para ver la implementación consultar el archivo service-worker.js.

Conclusiones

- Las PWA, como un conjunto de tecnologías, se encuentran madurando y en la actualidad se puede contar con un ecosistema lo suficientemente completo para desarrollar aplicaciones complejas ofreciendo una experiencia web muy similar a la nativa.
- El acceso a hardware de los dispositivos para las PWA es limitado, aunque el aumento de las API de acceso a hardware incrementa a gran velocidad.
- PWA es una alternativa competitiva para el desarrollo multiplataforma, sin embargo, para aplicaciones de alto rendimiento como por ejemplo juegos el desarrollo nativo o el desarrollo brigde utilizando motores especializados es lo ideal.
- El mayor tráfico de internet ocurre en los dispositivos móviles, es por esta razón que tener una experiencia web atractiva y funcional es indispensable.

- El web app manifest file, el app shell model y los service workers son los pilares de las PWA.
- Las estrategias de cacheo deben ser utilizadas dependiendo del tipo de recursos que queremos gestionar.
- Cache API y IndexedDB son las APIs más completas de administrar la persistencia a datos en nuestras aplicaciones web.
- Background Sync y las más recientes APIs son en su mayoría utilizadas en Chrome, los demás navegadores les toma más tiempo soportar estas características.

Referencias

- Google Developers. (20 de Septiembre del 2019). The core foundations of a delightful web experience. Recuperado el 20 de Septiembre del 2019, de Web Fundamentals: https://developers.google.com/web/fundamentals/.
- Google Developers. (20 de Septiembre del 2019). Measure Performance with the RAIL Model. Recuperado el 20 de Septiembre del 2019, de Web Fundamentals: https://developers.google.com/web/fundamentals/performance/rail.
- Google Developers. (20 de Septiembre del 2019). Introducing Background Sync. Recuperado el 20 de Septiembre del 2019, de Web Fundamentals: https://developers.google.com/web/updates/2015/12/background-sync.
- Google Developers. (20 de Septiembre del 2019). Introduction to Push Notifications. Recuperado el 20 de Septiembre del 2019, de Web Fundamentals: https://developers.google.com/web/ilt/pwa/introduction-to-push-notifications.
- Google Developers. (20 de Septiembre del 2019). The Offline Cookbook. Recuperado el 20 de Septiembre del 2019, de Web Fundamentals: https://developers.google.com/web/fundamentals/instant-and-offline/offline-cookbook.
- Google Developers. (20 de Septiembre del 2019). Web Storage Overview. Recuperado el 20 de Septiembre del 2019, de Web Fundamentals: https://developers.google.com/web/fundamentals/instant-and-offline/web-storage.
- Google Developers. (20 de Septiembre del 2019). Service Workers: an Introduction. Recuperado el 20 de Septiembre del 2019, de Web Fundamentals: https://developers.google.com/web/fundamentals/primers/service-workers/.
- Google Developers. (20 de Septiembre del 2019). El manifiesto de las apps web. Recuperado el 20 de Septiembre del 2019, de Web Fundamentals: https://developers.google.com/web/fundamentals/web-app-manifest/?hl=es.
- Workbox. (20 de Septiembre del 2019). Workbox. Recuperado el 20 de Septiembre del 2019, de Workbox: https://developers.google.com/web/tools/workbox.
- MDN. (20 de Septiembre del 2019). Cache. Recuperado el 20 de Septiembre del 2019, de Cache: https://developer.mozilla.org/en-US/docs/Web/API/Cache.
- MDN. (20 de Septiembre del 2019). IndexedDB. Recuperado el 20 de Septiembre del 2019, de IndexedDB: https://developer.mozilla.org/es/docs/IndexedDB-840092-dup.