Curso de Progressive Web Apps con React.js

# ¿Qué es una PWA?

No hay una definición concreta sobre que es una PWA. Lo mejor es entender como es el panorama actual.

¿Cómo es la web hoy?

* **Spoiler Alert:** No funciona muy bien en mobile.
* Más del 50% de nuestros usuarios está en mobile.
* Tenemos malas conexiones en los dispositivos móviles, conexiones tipo 3G y LTE no son particularmente confiables. Esto genera que un sitio promedio tarda 14 segundos en cargar en mobile.
* La UX no es solamente el diseño de nuestra app, tiene que ver con que tan rápido funciona nuestra aplicación en el mundo real, con un teléfono que tiene una conexión mala.

Hay estudios que demuestran la importancia de lograr que tu sitio funcione rápidamente en dispositivos móviles:  
• 50% de los usuarios se van de un sitio que tarda más de 3 segundos en cargar.  
• Cada segundo de demora nos cuesta un 5-10% de nuestras ventas.

Vamos a centrarnos en tres cosas a lo largo de este curso:

1. Performance.
2. Instalable en Homescreen.
3. **Dar** soporte Offline.

# Progressive Web App vs. Native App

Similitudes y diferencias entre una **PWA** y una **App nativa**:

**Similitudes:**

* Trabajar Offline
* Recibir notificaciones
* Ser instaladas en la HomeScreen

Diferencias:  
**Las Apps Nativas**

* Acceso de mejor calidad a las funcionalidades nativas de Android y iOS
* Mayor libertad para crear apps avanzadas
* Mejor performance
* Mejores funcionalidades específicas de cada plataforma

**Las PWAs**

* Mejor desempeño en SEO
* Más sencillo conseguir y convertir usuarios
* Son mejores para sitios de medios, noticias o aplicaciones virales
* Costos de producción menores
* Facilidad para conseguir usuarios

Al momento de decidir deberíamos hacernos la siguiente pregunta:

**¿Podemos crear una experiencia de usuario dramáticamente SUPERIOR con una app nativa?**

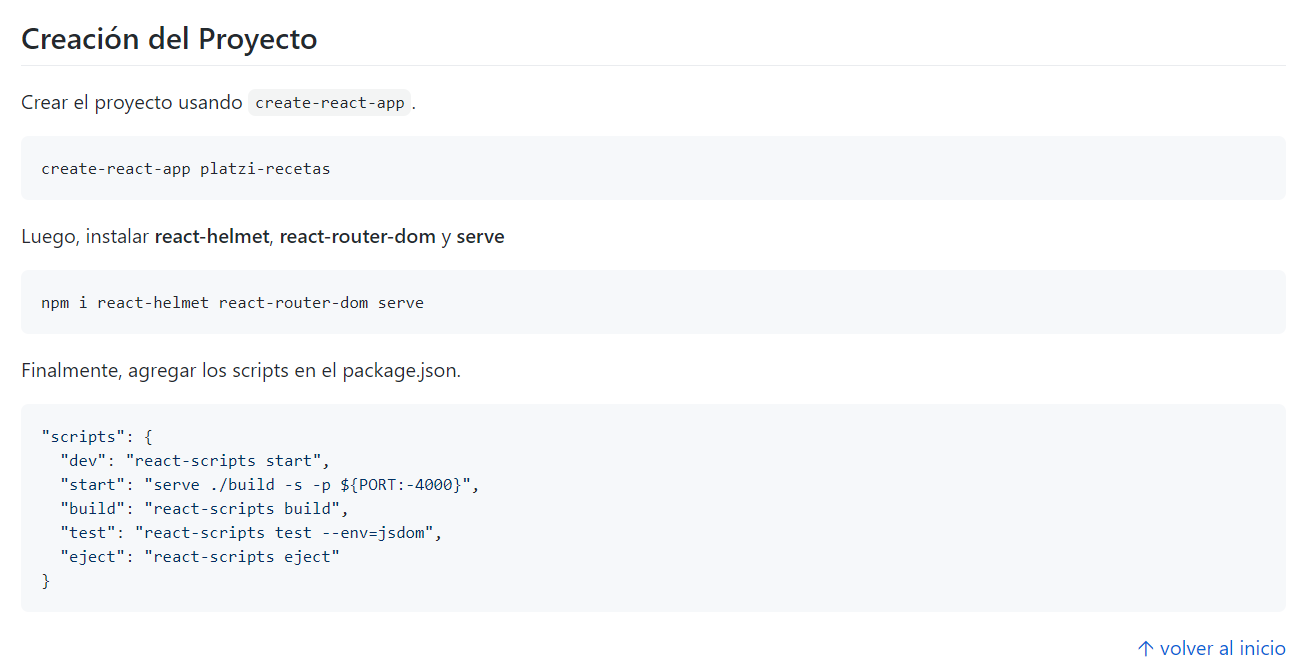
¡Si la respuesta es SI, y dispones del personal y los recursos para hacerlo, pués adelante!

De lo contrario, la mejor decisión es:

**¡Crear una PWA!**

# El proyecto

Nuestro proyecto será **Platzi Recetas:** una PWA para ver recetas que utiliza la API de TheMealDB.  
Solamente usaremos React, generado por Create React App, junto con React Router.



<https://github.com/MineiToshio/CursosPlatzi/tree/master/Curso%20de%20Progressive%20Web%20Apps%20con%20React.js>

La API se actualizo ahora hay que cambiar **baseUrl** y las peticiones **fetch()**:  
. // import 'isomorphic-fetch'

**const** baseUrl = "https://www.themealdb.com/api/json/v1/1";

**asyncfunctiongetLatest**() {

**const** request = **await** fetch(`${baseUrl}/search.php?f=b`);

**const** data = **await** request.json()

**const** recipes = data.meals.map((m) => normalizeMeal(m))

**return** recipes

}

**asyncfunctiongetRecipe**(recipeId) {

**const** request = **await** fetch(`${baseUrl}/lookup.php?i=${recipeId}`)

**const** data = **await** request.json()

**if**( ! data.meals ) **returnnull**

**const** recipe = normalizeMeal(data.meals.shift())

**return** recipe

}

# Google Lighthouse

Es una herramienta oficial de Google que viene con Chrome, con la cual podemos hacer un diagnóstico a una Web App. Estos diagnósticos se centran en Performance y Accesibilidad, pero también tiene una herramienta para diagnosticar si nuestra Web App se considera una PWA o no y que pasos debemos de tomar para que lo sea.

Lighthouse no sustituye hacer pruebas con un dispositivo móvil real, siempre realiza pruebas en un dispositivo móvil.

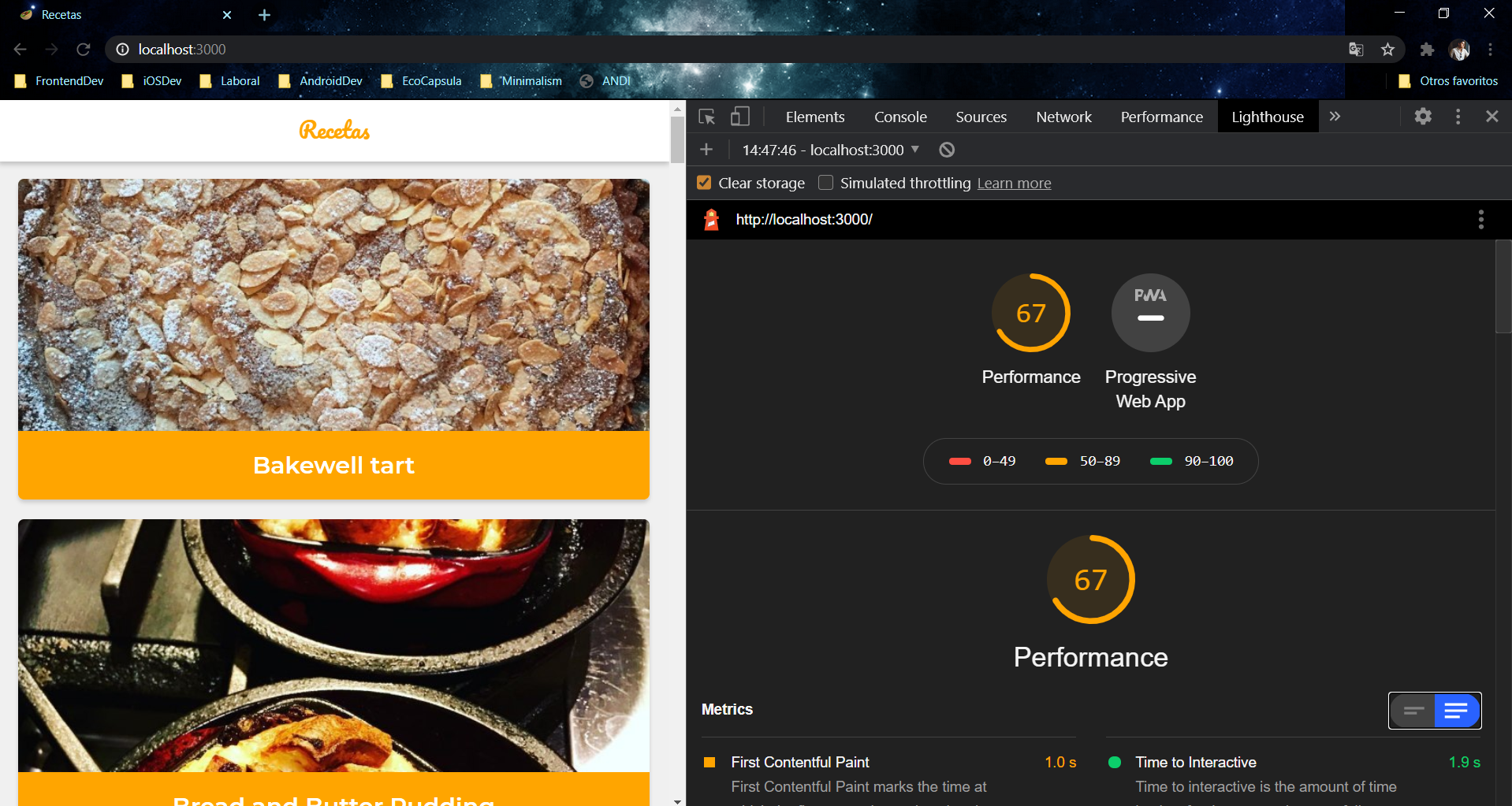
El diagnostico de Performance nos muestra dos de los conceptos más importantes en performance: First meaningful Paint y First interactive.

**First meaningful Paint** o primer pintado significativo, esto señala cuanto tiempo tardo el navegador en renderizar la aplicación de una forma que tenga sentido. Generalmente queremos que este situado entre 1 y 2 segundos.

**First interactive** o primera interacción, señala el tiempo cuando ya se cargó React, inicializo la aplicación y que podamos correr comandos dentro de la aplicación.

¿Cómo bajamos estos tiempos?

Para bajar el Time To First Meaningful Paint podemos hacer Server Side Rendering, reducir el tamaño de nuestro HTML y CSS o simplemente teniendo servidores más rápidos.  
El **Time To Interactive** tiene mucho que ver con el framework que estemos utilizando, usualmente queremos que tarde menos de 5 segundos.



# Creando un Web Manifest

En esta clase vamos a ver cómo implementar la funcionalidad de Add to Homescreen.

**Nota IMPORTANTE:**

Al momento de crear este curso esta saliendo Chrome 68, dicha versión va a cambiar el comportamiento del Add to Homescreen sutilmente.  
create-react-app nos da un Web Manifest pre armado el cual debemos configurar. Todo lo que tiene que ver con nuestro Web Manifest está dentro de los archivos index.html y manifest.json de la carpeta public de nuestro proyecto.

Por el momento vamos a trabajar dentro del archivo manifest.json, en el podemos ver varios atributos, los cuales son:  
• short\_name: Es el nombre que se utiliza en la Homescreen.  
• name: Es el nombre de nuestra aplicación.  
• icons: Especifica un array de imágenes que servirán como iconos de la aplicación. Cambiaremos el “favicon.ico” por “icon.png”, especificamos el tamaño a 512x512 y el tipo a “image/png”.  
• start\_url: Nos indica en que página comienza nuestra aplicación, por compatibilidad siempre conviene que sea “/” en lugar de “./index.html”.  
• display: Define el modo de visualización para la aplicación. Standalone significa que la aplicación puede correr por su misma.  
• theme\_color: Define qué color vamos a usar en la barra de tareas de Android para que combine con nuestra aplicación.  
• related\_applications: Sirve si queremos que Chrome en el Add to Homescreen recomiende una aplicación del Store.

Para probar nuestro Add to Homescreen debemos tener en cuenta que un requisito fundamental de las PWA es que todo funcione con HTTPS.

Nuestra aplicación por defecto es fullscreen, así que NO OLVIDES de brindar un camino al home.

En iOS necesitamos añadir alguna metadata al index.html de nuestro proyecto. Al momento de probar nuestra aplicación en iOS nos daremos cuenta de que el Add to Homescreen en este caso debe ser añadido manualmente por el usuario.

Tuve problemas al ejecutar

npm run build && npm start

Lo que hice fue modificar el script de start

"start": "serve ./build -s -p 4000",

luego, ejecute de nuevo:

npm run build && npm start

¡Guao! … el dato sobre este servicio **ngrok** es una joya! … Gracias.

Instrucciones para instalación en: [https://ngrok.com](https://ngrok.com/)

Para que ngrok funcione correctamente se debe tener corriendo el servidor local en una terminal, y el ngrok en otra terminal, ambas apuntando al mismo puerto … ya que ngrok … hace las veces de proxy o similar.

Al momento de acceder a la webapp desde el dispositivo móvil hay que asegurarse de escribir **https://** ya que de lo contrario no tiene el mismo comportamiento, aún cuando no falla, ya que ngrok pone a disposición ambos protocolos… http y https

Así se va viendo…



# Implementar el Manifest

Para finalizar este módulo pon a prueba lo aprendido sobre el Web Manifest y Google Lighthouse. Prueba con diferentes colores en el Web Manifest, experimenta con todas las opciones que puedes poner en el documento. En el mundo real, particularmente con proyectos complejos, nuestra PWA vendrá acompañada de otras aplicaciones móviles por lo cual te servirá ir practicando con la propiedad “related\_applications” para vincular dichas aplicaciones.

Varios ejemplos de PWAs  
<https://pwa-directory.appspot.com/>

Agrego para el que le interese.  
<https://pwa.rocks/>

Service Workers

# ¿Qué es un Service Worker?

Son la característica más importante de una PWA.

Es lo que hace posible que las PWA funcionen, es un script que nuestro navegador corre detrás de escena y este no puede tocar el DOM.

Podemos tener control absoluto a nivel red de nuestra aplicación gracias a los service workers.

Dato clave: los ServiceWorkers son scripts que corren en el navegador pero en background, y NO tiene acceso al DOM de la aplicación!

# Introducción a Workbox

En esta clase vamos a ver como implementar service workers en nuestra aplicación.

Los service workers solo funcionan en producción.

Una recomendación siempre que trabajemos con service workers es ir a Clear Storage en la tab de Application de las DevTools, y limpiar la información del sitio. Esto desinstalara todo lo que es cache y limpiara los service workers.

Quienes habrán prestado atención a la documentación de create-react-app habrán leído que este incluye un service worker.  
El service worker de Create React App hace algo llamado “SW Precache“, lo que hace es precargar y dejar disponibles offline todos los archivos necesarios para correr la aplicación. Una recomendación a la hora de hacer debugging es refrescar el sitio pues un service worker por lo general se inicializa después de la primera carga.

NUNCA conviene escribir nuestro propio service worker especialmente con herramientas de bajo nivel.

Para implementar nuestro propio service worker usaremos Workbox, una librería creada por Google para crear Service Workers.

Hay un pequeño detalle al momento de implementar Workbox en nuestro proyecto y es que estamos yendo en contra de los principios de Create React App y esto solo significa una cosa “eject”, esto nos llenaría de archivos que no nos sirven. Para evitar hacer eject vamos a instalar react-app-rewired y el plugin para webpack de workbox.

npm add workbox-webpack-**plugin** react-**app**-rewire-workbox react-**app**-rewired

ADEVERTENCIA:

Nunca conviene escribir nuestro propio ServiceWorker, especialmente con herramientas de bajo nivel. Esto no tiene que ver con un tema de confianza, sino con cómo funciona un SW y sobre todo lo qué un SW tiene control  
Una vez que el ServiceWorker se instala en el cliente, el sitio queda fuertemente cacheado cómo está y es muy difícil corregir bugs en este contexto.

RECOMENDACIÓN:

Siempre conviene confiar o utilizar alguna herramienta que esté muy testeada y probada, y que tenga buenas herramientas de testing. Y … NO intentar crear nuestros propios Service Workers.

¡Workbox es un ejemplo de estas herramientas probadas! y es by Google

npm **add** workbox-webpack-**plugin**

npm **install** react-app-rewired react-app-rewire-workbox

Aunque el proyecto esta desactualizado a nivel de dependencias, para que funcione deben copiar las versiones del package.json utilizadas en el curso

"workbox-webpack-plugin": "^3.3.1",

"react-app-rewire-workbox": "^2.0.1",

"react-app-rewired": "^1.5.2"

e instalar las dependencias de la forma

npm **install** --exact

Y funcionando hoy 2020/05/04 a pesar del versionamiento

# Implementando Workbox

En esta clase vamos a ver como implementar estrategias de carga con Workbox.

El funcionamiento de un service worker por defecto toma una lista de assets para precargarlos y si la ruta coincide exactamente con un asset entonces lo tomara de cache.

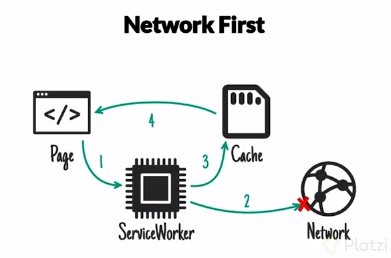
Workbox tiene una característica llamada registerNavigationRoute la cual se encarga de hacer el funcionamiento por defecto de un service worker más aparte si encuentra una url que no conoce va a buscar una url, en este caso index.html y que el se encargue de lo que va a mostrar.

Existen diferentes estrategias de carga.  
La primera y fundamental se llama Network Only. Esta se encarga checar si hay conexión a internet, si existe una conexión realiza la petición de información, en caso de no haber conexión se rompe la página.

**¿Cuándo usar Network Only?**  
Por defecto si no queremos cache o manejamos información en tiempo real.

Network First es otra estrategia de carga, se encarga mandar la petición a internet, si la conexión a internet esta caída entonces tomara la información que tenga almacenada en cache.

**¿Cuándo usar Network First?**  
Cuando queremos la última versión de un asset y tener soporte offline.



Algunas anotaciones de interés:

* **Network Only:** Por defecto. Si hay conexión a Internet, busca el recurso, si no la hay pues dinosaurio de Chrome.
* **Network First:** Es la forma “por defecto” de manejar el modo Offline de una PWA, intenta buscar en la red algún recurso, en caso de no tener conexión lo busca en el caché

Actualmente hay otra manera de definir una estrategia por defecto:

workbox.routing.setDefaultHandler(

**new** workbox.strategies.NetworkFirst(),

);

Ahora bien, la que presenta el profesor también funciona:

workbox.routing.registerRoute(

/^https?.\*/,

**new** workbox.strategies.NetworkFirst()

);

Para solucionar la situación en la que se hace caché sólo de los elementos base del home de la app, pero no de las páginas internas, debemos entender un poco y emular lo que hace el parámetro -s en el script “start” con serve …

serve ./build -s -p ${PORT:-5000}

el parámtero **-s** lo que hace es: identificar cualquier ruta que no sea un archivo estático, y si no es estático, lo sirve a través de un archivo **index.html** que es el core de la aplicación y es quién va a decidir finalmente si una ruta es válida o no.

En Workbox hay que hacer lo mismo. Se le llama, navigation route.

workbox.routing.registerNavigationRoute('/index.html')

# Aplicando Estrategias de Carga

En esta clase vamos a analizar diferentes estrategias de carga, veremos sus ventajas y desventajas.

• Cache First.

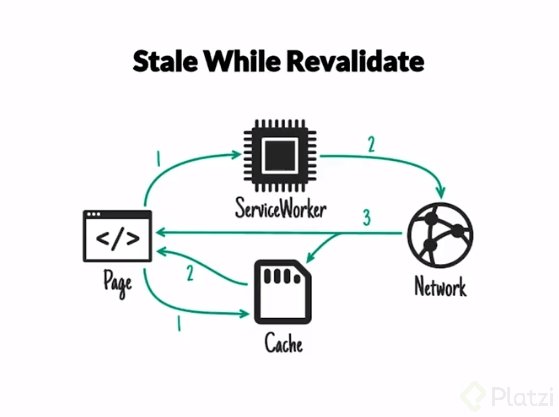
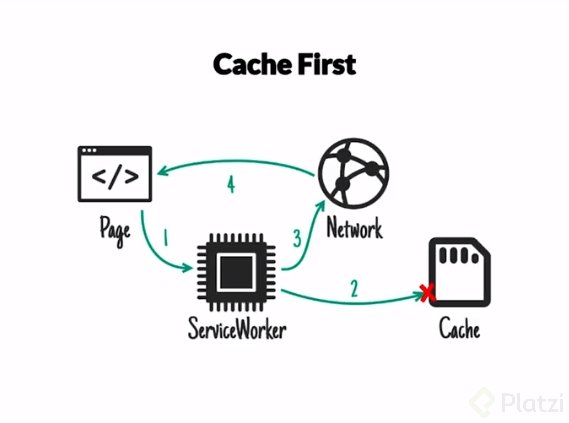
Es una estrategia de carga que lo primero que hace es ir al cache y si encuentra el recurso lo sirve directamente. En caso de no encontrarlo va a ir a red, guardar la información en cache y servir esa versión.

Esta estrategia puede ser peligrosa y solo es recomendable cuando queremos máxima velocidad y estamos manejando un recurso que nunca cambia, como una imagen o alguna fuente.

• Stale While Revalidate

Esta es una estrategia de carga muy particular y que mejor funciona a la hora de mejorar el rendimiento. Lo que hace es ir a cache y a red al mismo tiempo, toma la versión más rápida que siempre será la de cache y en cuanto recibe la de red va a actualizar la versión de cache.

Es recomendable esta estrategia cuando queremos mucha velocidad y estamos manejando un recurso que puede estar levemente desactualizado.  
Al momento de escribir nuestras estrategias en Workbox SI IMPORTA el orden en que pongamos las cosas, si queremos una estrategia o regla por defecto debemos poner esa regla hasta el final de todo.



Algunas anotaciones de interés:

* **Cache First:** Primero busca el recurso en caché, en caso de no encontrarlo va a la red, lo trae, lo guarda en caché y nos lo sirve desde el caché. Con esto no sale nuevamente a la red a menos que sea eliminado del caché. Esta estrategia puede ser peligrosa, con este tipo de estrategias se deben cachear cosas que no cambiarían (normalmente) con el pasar del tiempo, como por ejemplo: Fuentes, Imágenes, Estáticos.
* **Stale While Revalidate:** Va a la chaché y a la red al mismo tiempo, es obvio que caché será más rápido, por eso trae primero el recurso desde el caché pero al regresar de la red con una actualización de dicho recurso lo guarda en caché y actualiza la UI.

**Importante Recordar**  
El orden de las reglas en service-worker.js es importante  
La primera que se matchea es la que afecta y el resto se anula.  
la regla por defecto siempre al final de todo. (en este caso el network first)

**Service Workers necesitan una conexión encriptada. Si ya tienes un server relativamente económico, puedes utilizar "Let's encrypt" para poder tener un HTTPS.**

# Google Analytics Offline

En esta clase vamos a implementar Google Analytics con soporte offline en nuestra aplicación.

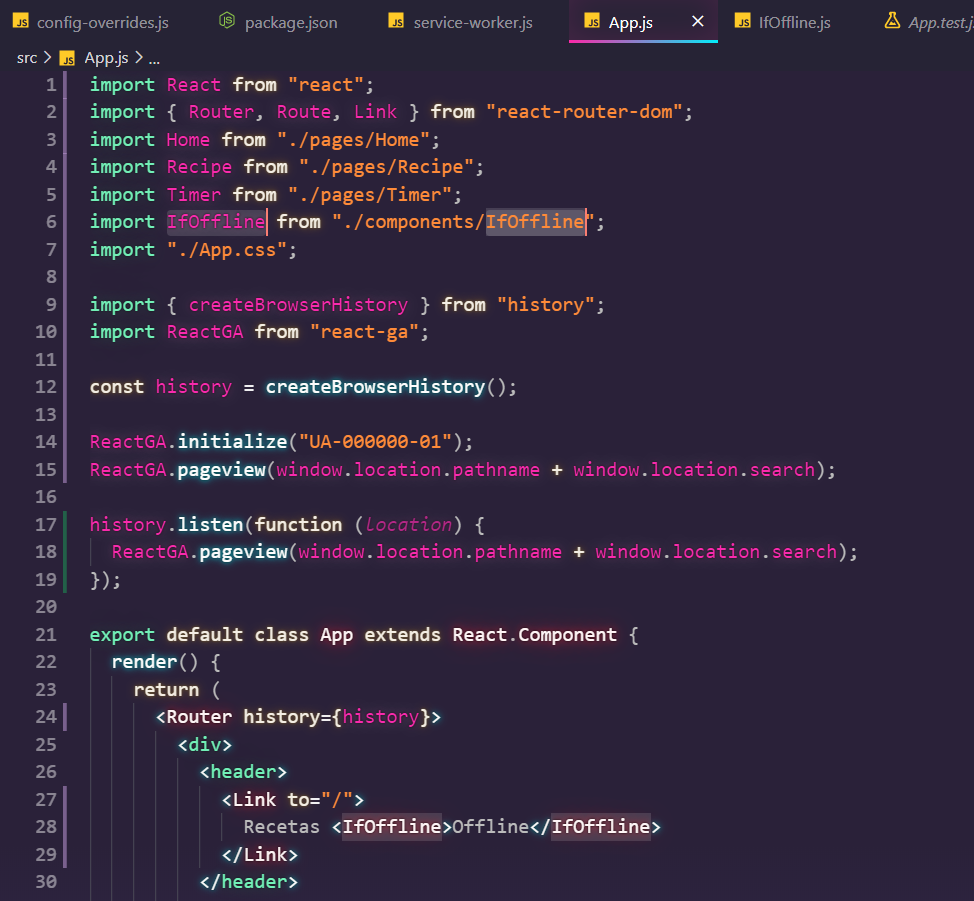
Como primer paso debemos incorporar react-ga, un plugin que nos permite correr Google Analytics dentro de React.

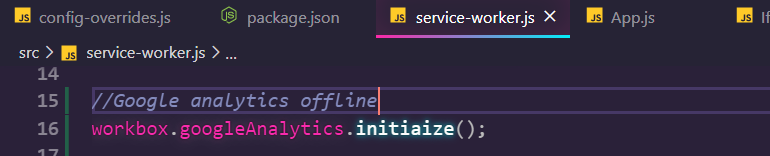
Para unir nuestro plugin a la historia de React Router la mejor opción es incorporarlo dentro de la historia de la aplicación cambiando el BrowserRouter por un Router común, creamos un nuevo history para poder extender los métodos del Router, y que cada vez que el usuario cambie de pagina haga tracking de una page view.

Si tienes algún AdBlocker desactívalo cuando estés desarrollando tu sitio para que evitar que bloqueé Google Analytics.

Workbox ya cuenta con un método para facilitar que Google Analytics funcione de forma offline, va a capturar todas las peticiones que hagamos a GA, las va a guardar en memoria y cuando el usuario retome la conexión a internet se enviaran las peticiones.

* npm add react-ga
* import ReactGA from 'react-ga'
* import { createBrowserHistory } from 'history'
* window.location.pathname + window.location.search
* ReactGA.initialize('UA-000000-01')
* workbox.googleAnalytics.initialize()

****

****

# Experimentar con Workbox

Para que experimentes con las diferentes estrategias de carga que ofrece workbox realiza el siguiente reto:

1. Las imágenes deben tener una estrategia **Cache** **First**.
2. Las imágenes guardadas **en** cache deben tener un vencimiento **de** 1 semana.
3. Probar Plugins y Estrategias.

Mi solución, espero les sea de ayuda.

workbox.routing.registerRoute(

/^https?:\/\/www.themealdb.com\/images\/.\*/,

**new** workbox.strategies.CacheFirst({

cacheName: 'image-cache',

plugins: [

**new** workbox.expiration.Plugin({

maxAgeSeconds: 7 \* 24 \* 60 \* 60,

maxEntries: 20

})

]

})

)

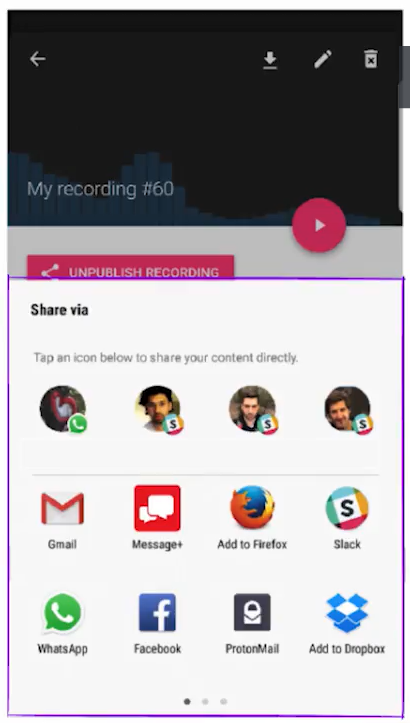
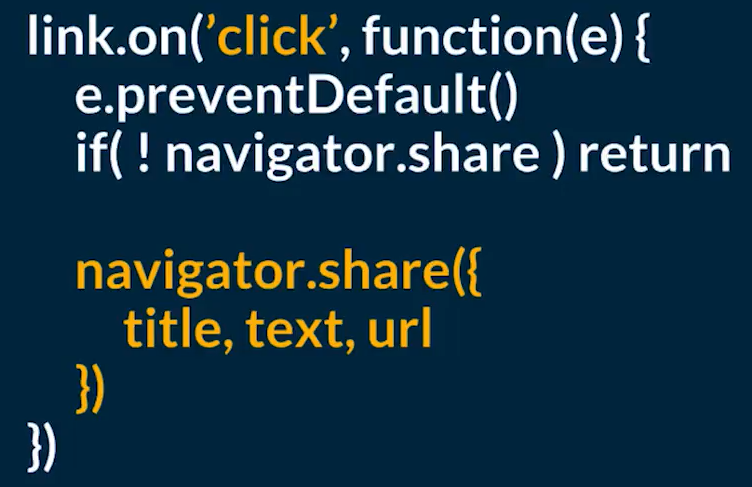
Funcionalidades Avanzadas

# Web Share API

Web Share API es una API reciente de Android que nos permite usar el Share nativo del sistema operativo.

Para implementarlo hay que tener presente que solo funcionara si hacemos click a algún link, esto es una medida de precaución para que nadie abuse de la API obligándonos a tener que compartir algo que no queremos. Además, Web Share API por el momento solo funciona en Android así que tenemos que detectar si tenemos la característica para poder usarla.

Web Share API solamente funciona con HTTPS.

****  ****

****

****

Parece ser que a día de hoy Web Share API ya funciona en iOS

<https://caniuse.com/#feat=web-share>