UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

DESARROLLO E IMPLEMENTACIÓN DE UNA APLICACIÓN WEB PROGRESIVA (PWA)

A Degree Thesis

Submitted to the Faculty of the

Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona

Universitat Politècnica de Catalunya

by

Gerard Castell Ferreres

In partial fulfilment

of the requirements for the degree in Telecommunications Technologies and Services ENGINEERING

Advisor: Jose Luis Muñoz Tapia

Barcelona, May 2020

Abstract

The main goal of this thesis is design and develop a progressive web application right from scratch. This new technology adds a stack of powerful capabilities to web apps in order to extend its boundaries, such as the possibility of being installed in desktop and mobile devices, no matter whether it is Android or iOS. Furthermore, it allows to interact with the web offline and receive push notifications among many other features supported, providing a more complete user experience.

Whilst PWA plays the main role of the project, React could be considered as co-star. This is a popular javascript library created by Facebook and based in a composable architecture which leads to simplicity, lightweight and speed.

Finally, to put into practice all features introduced I have developed a PWA that offers tools to build social stories. This stories are commonly used with the aim of help kids with Autism Spectrum Disorder and special needs learn to communicate.

Resum

L'objectiu principal d'aquesta tesis es dissenyar i desenvolupar una aplicació web progressiva. Esta nova tecnologia proporciona un conjunt de potents capacitats a les aplicacions web amb la fi d'estendre els seus límits, com per exemple permetre que sigui instal·lada a l'escriptori i en dispositius mòbils, sense importar si es tracta d'Android o d'iOS. A més, permet interactuar amb la web "offline" i rebre notificacions "push" entre moltes altres eines, oferint una experiència d'usuari més completa.

Mentre que les PWA son el protagonista del projecte, React podria ser considerat el coprotagonista. Es tracta d'una popular llibreria de Javascript creada per Facebook i basada en una arquitectura composable que comporta simplicitat, lleugeresa i velocitat.

Finalment, per posar en pràctica totes les eines presentades, he desenvolupat una PWA que ofereix instruments per construir històries socials. Aquestes històries socials són comunament utilitzades amb l'objectiu d'ajudar a aprendre a nens amb Trastorn de l'Espectre Autista i necessitats especials a comunicar-se.

Resumen

El objetivo principal de esta tesis es diseñar y desarrollar una aplicación web progresiva. Esta nueva tecnología proporciona un conjunto de potentes capacidades a las aplicaciones web con el fin de extender sus límites, como por ejemplo permite que la web sea instalada en el escritorio y en dispositivos móviles, sin importar si se trata de Android o iOS. Además, permite interactuar con la web “offline” y recibir notificaciones “push” entre muchas otras herramientas, ofreciendo una experiencia de usuario más completa.

Mientras que las PWA son el protagonista del proyecto, React podría ser considerado el coprotagonista. Se trata de una popular librería de javascript creada por Facebook y basada en una arquitectura componible que comporta simplicidad, ligereza y velocidad.

Finalmente, para poner en práctica todas las herramientas presentadas, he desarrollado una PWA que ofrece instrumentos para construir historias sociales. Estas historias son comúnmente utilizadas con el objetivo de ayudar a niños con Trastorno del Espectro Autista y necesidades especiales para aprender a comunicarse.

Me gustaría dedicar esta tesis a mi hermano Sergi, por inspirarme a superar barreras día a día y enseñarme a ver siempre el lado bueno de las cosas.

Reconocimientos

Esta tesis no se hubiera podido llevar a cabo sin la ayuda y colaboración de Jose Luis Muñoz, por sus recomendaciones y constante ayuda en el proyecto, tanto en la parte teórica como en la práctica.

Por otra parte, quería agradecer a mi madre por inspirarme en la idea para realizar este proyecto,

También a mi padre, y a mi hermano Xavi por su constante apoyo.

Revisión histórica y registro de aprobación

Revision	Date	Purpose
0	02/04/2020	Document creation
1	06/05/2020	Document revision

DOCUMENT DISTRIBUTION LIST

Name	e-mail
Gerard Castell Ferreres	gerardcastell97@gmail.com
Jose Luis Muñoz Tapia	jose.munoz@entel.upc.edu

Written by:		Reviewed and approved by:	
Date	02/04/2020	Date	06/05/2020
Name	Gerard Castell Ferreres	Name	Jose Luis Muñoz Tapia
Position	Project Author	Position	Project Supervisor

Tabla de contenidos

Lista de Figuras

Lista de Tablas

1. Introducción

Vivimos en una sociedad en la que la tecnología está cada vez más presente en nuestras vidas. Existe un avance constante de toda esta tecnología que nos rodea para intentar mejorar y facilitar nuestras vidas. En 1990, el informático inglés Tim Berners-Lee creó el primer servidor de páginas web de la historia en el CERN (Ginebra, Suiza), con el que pretendía crear un sistema que permitiera a los investigadores compartir fácilmente la información. Con el paso del tiempo esta tecnología fue desarrollándose estándares de lenguajes como HTML o CSS para su implementación y así expandirse en universidades, centros de investigación y empresas, hasta conformar una gran red de información que hoy conocemos como Internet. Las Aplicaciones Web Progresivas, en adelante PWA, son la evolución de las páginas webs en un mundo cada vez más centrado en la tecnología móvil y para demostrarlo explicaré brevemente cómo se fueron cubriendo las necesidades técnicas hasta llegar a las PWA, por qué he decidido implementarla en mi proyecto con React y explicar qué aporta esto, y, finalmente, la necesidad social y educativa que pretendo cubrir al haberla vivido en primera persona.

1.1. Objetivos

1.1.1. Aplicaciones Web Progresiva

Al avanzar la tecnología móvil poco a poco se ha ido convirtiendo en el centro de la actividad online. Hoy en día, desde prácticamente cualquier dispositivo móvil podemos acceder a Internet, ya sea bién a través de navegadores o aplicaciones instaladas, lo que ha permitido que el 60% del tráfico actual sea a través de teléfonos móviles. Desde el punto de vista de un usuario parece trivial que cualquier instrumento con acceso a la red pueda disfrutar de dichas prestaciones, sin embargo, los desarrolladores bien sabemos que no existe un único lenguaje de programación de alto nivel que todos los dispositivos sean capaces de interpretar. Cada lenguaje posee sus ventajas e inconvenientes y esto obliga a los programadores a ser versátiles en función de la finalidad del código. Esto explica por qué aparecieron las aplicaciones móviles nativas con Android y iOS, referentes del sector de sistemas operativos para telefonía, los cuales permiten sacar el máximo rendimiento del dispositivo y sus periféricos pero a su vez obliga a los desarrolladores a aprender lenguajes como Kotlin para Android y Swift para iOS. La gran influencia de la tecnología web hizo que los programadores no se rindieran y se crearon soluciones alternativas como las “web views” con Cordova o PhoneGap, que creaban una aplicación móvil con lenguajes orientados a web y luego lo envolvían con una capa puente que se comunicaba con el software del móvil. Más tarde aparecieron las aplicaciones híbridas como Ionic, de Google, o React Native, de Facebook, las cuales se deshacen de este envoltorio y directamente tratan con el dispositivo, lo cual acercaba el rendimiento al de una app nativa manteniendo ciertas limitaciones como la fluidez en la interacción o la dependencia de la velocidad en función del navegador de dicho dispositivo. En este punto, habíamos conseguido utilizar tecnología web para desarrollar apps móviles pero este código seguía sin ser reusable para web, aún.

Finalmente aparece nuestro protagonista, las PWA. Las PWA pretender juntar dos lo mejor de dos mundos: por una parte, el ilimitado alcance de las aplicaciones web que

nos permite buscar y compartir contenido con quien sea, cuando sea, donde sea y en cualquier dispositivo con un único código base. Además, al visitar una aplicación web el contenido está actualizado y la experiencia con este sitio puede ser tan efímera o permanente como queramos. Por otra parte, las aplicaciones nativas destacan por su riqueza y fiabilidad para el usuario. Las utilizamos incluso sin conexión a Internet ya que nos proporcionan una experiencia independiente y controlada. Son capaces de acceder a los distintos periféricos del dispositivo como el sistema de archivos local, bluetooth, cámara, micrófono e incluso interactuar con otras aplicaciones y información almacenada en el dispositivo generando una experiencia de usuario como si la aplicación formara parte del propio móvil.

Las aplicaciones web progresivas pretenden juntar y mejorar progresivamente las capacidades que nos aportan las aplicaciones nativas con el alcance de las aplicaciones web. En el fondo siguen siendo aplicaciones web pero que aprovechan los avances de los navegadores para aumentar sus capacidades, lo que permite mejorar la experiencia del usuario y los números reportados por algunas empresas que ya han incorporado la tecnología de las PWA lo demuestran. Por ejemplo, Twitter ha declarado un incremento del 76% en el número de tweets enviados y otro incremento del 65% de las páginas por sesión, esta es una métrica muy relevante de las Google Analytics que indica el número de páginas visitadas por un usuario durante una sesión. Cuanto mayor sea este número más inmerso está el usuario en la web.

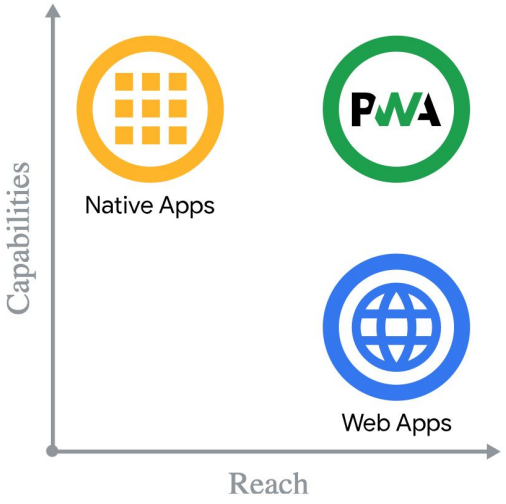


Figura 1: Capacidades vs. Alcance de las apps nativas, apps web y PWA.

1.1.2. React

Este proyecto se centra en desarrollo frontend, entendiendo el frontend como el conjunto de tecnologías de desarrollo y diseño web que se ejecutan en el navegador y se encargan de la interactividad con el usuario. Cuando se empieza a desarrollar frontend, no se trata de escoger tan solo un lenguaje de programación, que en sí ya es un análisis realmente complejo y vinculante para el resto del proyecto, sino que se trata de un

multitud de elecciones para cada una de las partes y/o funcionalidades de nuestra web.

La primera elección que hice fue escoger Javascript como lenguaje de programación. Sin duda, es el lenguaje más popular en el mundo del frontend, lo que implicaba una gran comunidad y madurez que lo respalda. Solo Javascript no suele bastar para implementar toda una aplicación web, aun siendo el lenguaje core, sería demasiado costoso desarrollar todas las funcionalidades con Javascript puro, tanto a nivel de organización como a nivel de complejidad. Para ello existen miles de frameworks y librerías que nos permiten atajar gran parte del codigo y ya utilizan algoritmos para compilar y/o transpilar el código en su core a Javascript ES15, que es el estándar que sí podrá interpretar la mayoría de navegadores. Opciones como Angular, mantenido por Google, Vue, Polymer, Svelte, ember, backbone, etc. fueron valoradas pero React fue la elegida no por cuestiones de azar. Desde mi punto de vista, esta librería presente una cantidad de ventajas respecto a los demás frameworks que lo convierten en el mejor. Es una opinión subjetiva pero lo que sí es objetivo es que la comunidad web lo sitúa indiscutiblemente como uno de los 3 mejores junto con Angular y Vue. Estas son las características que me hicieron escogerlo:

- **Rápido.** React se distingue por su velocidad a lo hora de renderizar el DOM.
- **Flexible y ligero.** Se desarrolla en un entorno sin montones de plantillas y paquetes innecesarios, sino que nos permite añadir las librerías que consideremos oportunas para el desarrollo.
- **Simple.** Es una librería de Javascript y para cualquier desarrollador que conozca el lenguaje no le será difícil aprender a usarlo.
- **Reusable.** La verdadera clave de React es que está totalmente enfocada para reusar código, tanto componentes del UI como de la lógica de negocio. Una vez programado podemos reusarlo donde queramos. A medida que el proyecto crec, esto simplifica el mapa mental, lo cual nos facilitará el razonamiento y los tests.
- **Respaldado.** La mayoría de desarrolladores lo prefieren y no cambiarían React por otra librería o framework.
- **Virtual DOM.** React posee un algoritmo realmente optimizado para gestionar los cambios en un DOM virtual cuando un componente varía y este realiza un proceso conocido como reconciliación para compararse con el DOM actual y de esta forma actualizar solo los elementos que se requieran y no todo el DOM, lo cual aporta una gran velocidad en la operación.
- **Multiplataforma.** Su ampliamente valorada filosofía ha impulsado su expansión a otras plataformas más allá de web, como podría ser en escritorio gracias al framework Electron, o bien aplicaciones móviles híbridas con React Native

Con estas pinceladas que he presentado, se puede entrever que introduce una filosofía totalmente distinta al resto. La mayoría de los frameworks, establecen todas las reglas, metodologías y mecanismos que nos permitirán usarlo para desarrollar contenido web, es decir, definen una solución completa con una arquitectura cerrada. En cambio, React abre un mundo de posibilidades donde él solo establece como se renderizará la vista en el navegador y se encargará de mantenerla sincronizada con el estado. Todo el resto es customizable a elección del programador lo cual permite una gran libertad y creatividad. En consecuencia, impulsó la creación de librerías de terceros para aportar soluciones

alternativas a los mecanismos ya existentes. Con filosofía, React marcó en su momento un nuevo paradigma en el desarrollo de frontend que se convirtió en referencia para los demás frameworks y librerías competidoras, lo cual me convenció de que React estaba aportando la innovación y escogerlo no sería un error.

1.1.3. Sistemas de Comunicación Aumentativa y Alternativa

A diferencia de cómo he presentado esta introducción, la idea del proyecto no se conformó en el mismo orden. Cuando empecé a pensar en el tema que quería tratar en mi trabajo final de grado no tenía nada claro la tecnología, pero sí tenía claro que quería aportar algo que ayudara a los niños con necesidades especiales. Uno de mis hermanos pequeños, Sergi, es autista y el hecho de haber crecido junto a él me ha enseñado muchísimo a lo largo de mi vida. Por ello, quería devolverle un poco de todo lo que él me ha dado. Empecé a indagar qué podía aportar para ayudar y me vino a la cabeza un recuerdo que tengo muy marcado: cuando era pequeño, mi madre se pasaba muchas noches en vela preparando historias sociales con pictogramas como éste que veréis a continuación que aún conservamos en la puerta de casa:

Figura 2: Fotografía de una historia social

A estos instrumentos se les denomina **Sistemas aumentativos y alternativos de comunicación (SAAC)**. Son formas de expresión distintas al lenguaje hablado que tiene como objectivo aumentar (aumentativos) y/o compensar (alternativos) las dificultades de comunicación y lenguaje de personas con discapacidad, trastornos de lenguajes y otras necesidades especiales. Como bien sabemos, la comunicación y el lenguaje son esenciales para que todo ser humano pueda relacionarse y participar en la sociedad. Gracias a estos sistemas podemos afrontar las dificultades del lenguaje oral y ayudar, ya sea niños, adultos o bien ancianos y por la causa que sea, a poder comunicarse de forma satisfactoria mejorando su nivel de habla. Los comunicación aumentativa y alternativa utiliza tanto **sistemas de símbolos**, como fotografías, pictogramas, dibujos, palabras o letras, como **gestuales**, por ejemplo mímica, gestos o signos manuales.

En el caso de Sergi estos sistemas tuvieron un impacto muy relevante en el aprendizaje de Sergi. Con los sistemas de símbolos representábamos historias sociales que nos servían para comunicarnos con Sergi, ya que el padecía un trastorno de lenguaje y su principal problema era la comprensión y la expresión. Él, como muchos otras autistas necesitaba secuencias en su día a día, con el orden se sentía seguro y gracias a estas historias sociales nosotros éramos capaces de contarle cómo sería su día, explicar y

corregir malos hábitos y comportamientos, así como premiar otros; y además permitía a Sergi comunicarse. Os podéis hacer una idea del papel fundamental que jugaba esta herramienta y lo crucial que ha sido para su desarrollo. Para que Sergi siguiera aprendiendo mi madre pasaba horas y horas buscando y preparando imágenes de Internet que podían sernos útiles. Buscar, descargar, imprimir, forrar y así preparaba los paneles, cada uno asociado a una temática. El que hemos visto en la figura anterior se encontraba en la puerta de casa para explicar dónde íbamos al salir por la puerta. Habían otros más en la cocina, baño, dormitorio… Este es el que tenemos en la cocina que aún conservamos:

Figura 3: Fotografía de un panel con pictogramas

Como podéis deducir para poder realizar una buena comunicación se precisa de un extenso vocabulario por lo que necesitamos muchos pictogramas y hubo muchas horas de trabajo. En este punto encontré cómo podría aportar mi grano de arena. El objetivo era crear un recurso tecnológico que permitiera buscar pictogramas y poder construir una historia social en cualquier dispositivo. Investigué sobre soluciones híbridas y multiplataforma, ya que pensé que este recurso sería útil para muchos logopedas y centros escolares y se debería poder utilizar en cualquier dispositivo. Me topé con una nueva tendencia, las PWAs y al ver que se acoplaban perfectamente a lo que estaba buscando decidí proponerle el proyecto a Jose, mi tutor. Él aceptó y me habló de implementarlo con React, el cual trataba y conocía realmente bien, así que tras investigarlo y evaluar sus pros y contras vi su potencial y que era la mejor opción. De esta forma se definió el objetivo principal del proyecto: construir una PWA con React para mejorar la experiencia a la hora de crear historias sociales, aunque todavía quedaba un largo camino de formación tecnológica por delante antes de poder empezar a desarrollar dicha aplicación.

1.2. Requerimientos y especificaciones

Para desarrollar el proyecto estos han sido los requerimientos que he necesitado:

- **Requerimientos físicos**
  - Un ordenador Mac OS

- **Requerimientos de software**
  - **Aplicaciones de M**
    - VS Code como IDE para desarrollar todo el código
    - Google Chrome para visualizar y depurar la aplicación
    - Una versión gratuita de Sketch para diseñar el UI
    - Iterm2 como emulador de terminal para Mac OS
  - **Librerías core para el proyecto:**
    - Git para el control de versiones del código
    - Node como entorno de ejecución para poder servir Javascript
    - React y ReactDOM como base del proyecto
    - Firebase como BaaS
    - Curso online de la plataforma Udemy de React y Redux Moderno

1.3. Plan de trabajo, Hitos y Diagrama de Gantt

Dado que tanto el documento “TFG Proposal and Workplan” como el “Project Critical Review” se entregaron el primer cuatrimestre cuando todavía no se había decidido realizar la extensión del proyecto se presentó un nuevo plan de trabajo que desgloso y justifico en los siguientes apartados.

1.3.1. Tareas

<table>
<tr><td>Project: Desarrollo de una PWA</td><td colspan="2">WP ref: (WP1)</td></tr>
<tr><td>Major constituent: Análisis</td><td colspan="2">Sheet 1 of 8</td></tr>
<tr><td rowspan="2">Short description: Definir las herramientas de desarrollo para frontend que se utilizarán. Además, se debía definir las funcionalidades de la aplicación para ver sus limitaciones y que herramientas extras se necesitarán.</td><td colspan="2">Planned start date: 01/09/19
Planned end date: 20/09/19</td></tr>
<tr><td colspan="2">Start event:01/09/19
End event: 20/09/19</td></tr>
<tr><td>Internal task T1:
Definir las herramientas para el desarrollo
Internal task T2:
Definir las funciones y experiencia de usuario de la app</td><td>Deliverables:
Esquema del flujo funcional de la app y informe de las herramientas necesarias</td><td>Dates:
20/09/19</td></tr>
</table>

Tabla 1. Tarea 1.

<table>
<tr><td>Project: Desarrollo de una PWA</td><td>WP ref: (WP2)</td></tr>
<tr><td>Major constituent: Formación de React</td><td>Sheet 2 of 8</td></tr>
<tr><td>Short description: Adquirir experiencia con React. Para ello, compré un tutorial de la plataforma educativa virtual Udemy con el que realice un curso durante varias</td><td>Planned start date: 20/09/19
Planned end date: 18/10/19</td></tr>
</table>

semanas y desarrollé varios mini proyectos para llevar a la práctica lo aprendido.	Start event:20/09/19 End event: 30/11/19	
Internal task T1: Realizar el curso y sus prácticas	Deliverables: -	Dates: -

Tabla 2. Tarea 2.

Project: Desarrollo de una PWA	WP ref: (WP3)	
Major constituent: Sketch y diseño	Sheet 3 of 8	
Short description: Aplicar las funciones definidas previamente de la aplicación para realizar el diseño de la interfaz de usuario, es decir, las distintas vistas.	Planned start date: 18/10/19 Planned end date: 25/10/19	
	Start event:30/11/19 End event: 10/12/19	
Internal task T1: Realizar el diseño de UI	Deliverables: Diseño	Dates: 10/12/19

Tabla 3. Tarea 3.

Project: Desarrollo de una PWA	WP ref: (WP4)	
Major constituent: Investigar API de Pictogramas	Sheet 4 of 8	
Short description: Realizar una búsqueda de alguna API pública y gratuita que pueda utilizar para aportar las imágenes de los pictogramas y investigar el comportamiento de los service workers para ver cómo podemos sacarle provecho a las capacidades de las PWA.	Planned start date: 25/10/19 Planned end date: 01/11/19	
	Start event:10/12/19 End event: 02/01/20	
Internal task T1: Investigar sobre una API de pictogramas Internal task T2: Investigar qué apps encontramos en el mercado actual Internal task T3: Investigar los service workers y sus funcionalidades	Deliverables: Listas de APIs sugeridas y cómo utilizar los service workers	Dates: 02/01/20

Tabla 4. Tarea 4.

Project: Desarrollo de una PWA	WP ref: (WP5)	
Major constituent: Desarrollo versión 1.0 de la app	Sheet 5 of 8	
Short description: Desarrollar todo la lógica y vistas de la aplicación, integrar la API de pictogramas y las demás herramientas preseleccionadas para mejorar la aplicación como Redux.	Planned start date: 01/11/19 Planned end date: 27/12/19	
	Start event:02/01/20 End event: 01/03/20	

Internal task T1: Desarrollo del core de la app, desde las vistas hasta la integración de la API y sus herramientas	Deliverables: V.1.0 de la app sin backend	Dates: 01/03/20

Tabla 5. Tarea 5.

Project: Desarrollo de una PWA	WP ref: (WP6)	
Major constituent: Creación de la base de datos y autentificación	Sheet 6 of 8	
Short description: Hacer un investigación para ver las posibles soluciones de BaaS y implementar la base de datos y sistema de autentificación. Una vez definido, integrarlo con la app. Desarrollar las vistas de “Sign In” y “Sign Up”	Planned start date: - Planned end date: -	
	Start event:01/03/20 End event: 29/03/20	
Internal task T1: Análisis de las posibles soluciones BaaS Internal task T2: Desarrollo del modelo de datos y autenticación así como su integración con la V.1.0 de la app.	Deliverables: V.2.0 de la app con flujo de usuario	Dates: 29/03/20

Tabla 6. Tarea 6.

Project: Desarrollo de una PWA	WP ref: (WP7)	
Major constituent: Retocar y arreglar versión final	Sheet 7 of 8	
Short description: Acaba de configurar el comportamiento del service worker y realizar las pruebas finales de rendimiento y experiencia de usuario para solucionar los bugs finales.	Planned start date: - Planned end date: -	
	Start event:29/03/20 End event: 18/04/20	
Internal task T1: Desarrollo del service workers co sus respectivas en tiempo de ejecución Internal task T2: Comprobar rendimiento de la app y hacer QA	Deliverables: V.3.0 de la app, release estable	Dates: 18/04/20

Tabla 7. Tarea 7.

Project: Desarrollo de una PWA	WP ref: (WP8)	
Major constituent: Documentación	Sheet 8 of 8	
Short description: Redactar la versión final de la memoria explicándolo todo el proceso y las tecnologías investigadas.	Planned start date: - Planned end date: -	
	Start event: 03/04/20 End event: 03/05/20	
Internal task T1: Redactar la memoria final	Deliverables: Final report	Dates: 03/05/20

Tabla 8. Tarea 8.

1.3.2. Hitos

WP#	Task#	Short title	Milestone / deliverable	Date (week)
1	1	Análisis kit desarrollo	Informe	20/09
1	2	Flujo y funcionalidades de la app	Esquema	20/09
3	3	Diseño UI	Diseño	10/12
4	4	Investigar APIs	Informe	02/01
4	5	Información service workers	Informe	02/01
4	6	Benchmarking de apps similares	Informe	02/01
5	7	Desarrollo de vistas y lógica core de la app	Repositorio de código	01/03
6	8	Análisis herramientas Baas	Informe	29/03
6	9	Desarrollo modelo de datos y autenticación	Repositorio de código	29/03
7	10	Adaptación service worker y pruebas de rendimiento	Repositorio de código	18/04
8	11	Redactar la memoria final	Final Report	03/05

Tabla 9. Hitos.

1.3.3. Diagrama de Gantt

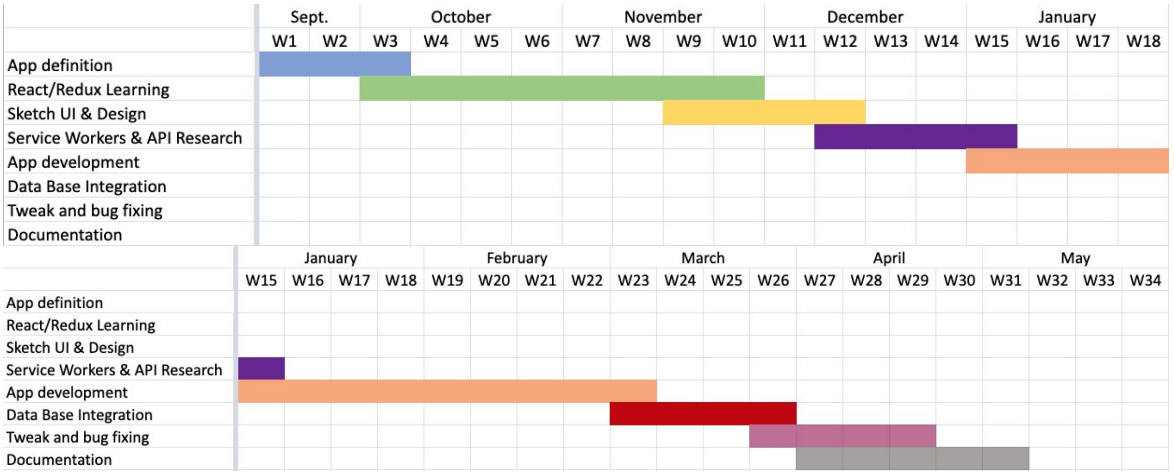


Tabla 10. Diagrama de Gantt.

1.4. Desviaciones e incidencias

En noviembre, mientras estaba con la formación de React y Redux, me di cuenta que había sido demasiado optimista con el tiempo que me costaría entender bien estas librerías para poder aplicarlas a mi proyecto. Por un lado no había programado demasiado en Javascript y el no conocer demasiado bien el lenguaje core limitaba mucho a la hora de desarrollar el código. Por otro lado, React poseía una curva de aprendizaje no tan elevada como otros frameworks, pero aún así, al ser el primero que aprendí requería entender bien su filosofía. Para conseguirlo realicé muchas prácticas y leí mucho sobre este paradigma en Reddit, antes de adoptar esta forma de pensar y poder hacer un desglose general de cómo iría a construir mi aplicación. Además, luego se añadió Redux, un patrón de diseño realmente útil para facilitar el flujo de datos entre los componentes y su estado. Esta librería tampoco fue tan trivial y fácil de adquirir como había estimado inicialmente.

Al acabar la formación, disponía de una fuerte base de React y Redux, pero Facebook había presentado los React Hooks, una nueva forma de enfocar los componentes con funciones facilitando la reusabilidad de la lógica, lo cual era complicado con la arquitectura que tenían hasta el momento, y gracias a esta nuevo enfoque, totalmente compatible con lo existente hasta el momento, se planteaba una potente herramienta que tanto Jose como yo creímos que valía la pena añadir en el trabajo, ya que sería la tendencia que iba a seguir la comunidad en los próximos años.

Estos fueron los factores principales que provocaron la extensión del trabajo y que vimos necesarios para cerrar un proyecto bien justificado a nivel técnico. Por consiguiente, también obtuvimos más tiempo para poder integrar el backend como servicio y el sistema de autenticación, lo cual era una caja negra de la que no sabía qué solución escoger, cuánto tiempo me iba a tomar y si me daría tiempo a integrar con todo el proyecto, ya que se desvinculaba del alcance del frontend pero, a su vez, aportaba un gran valor al producto final, y así ha sido.

Tuve que renunciar a investigar sobre el testing. Al final decidí trabajar más en el producto ya que la idea inicial del proyecto era crear una herramienta de Comunicación

Aumentativa y Alternativa y ya había afrontado muchos retos tecnológicos. Era preferible cerrar la PWA que añadir más tecnología y no poder concluirla.

Analizándolo al final del proyecto con una visión más general creo que tomé las decisiones correctas y necesarias y aunque el proyecto podría ampliarse mucho más a nivel tecnológico creo que se han justificado las incidencias para llegar al producto final del que estoy altamente satisfecho.

2. Estado del arte de la tecnología aplicada en esta tesis

2.1. Aplicaciones Web Progresivas

Como se ha comentado en la introducción, las PWA representan el futuro de las páginas webs. Una definición simple podría ser la siguiente: "*Una PWA utiliza las modernas capacidades web para ofrecer una experiencia de usuario similar a la de una app nativa. Evolucionan de las páginas en las pestañas del navegador hacia aplicaciones inmersivas de alto nivel, manteniendo la baja fricción de la web en todo momento*". Esto define una nueva metodología de desarrollo de software. Las PWA son un nuevo modelo que intenta combinar las herramientas que ofrecen los navegadores modernos con los beneficios de la experiencia en una app móvil.

Hasta este punto se han nombrado tan solo algunas funciones pero se plantean dos preguntas cruciales: ¿Cuáles son realmente los rasgos que caracterizan las aplicaciones web progresivas? ¿Cómo consiguen estas características a partir de una simple web app? Bien, en los siguientes apartados desglosaré las service workers para dar respuesta a estas cuestiones:

2.1.1. Características

Son varias las propiedades que acaban dando forma al concepto de PWA. A continuación los presentaré de forma ordenada.

Un concepto a tener en cuenta es la **progresividad** de las PWA. Muchas de sus funciones todavía no están soportadas en todos los navegadores, además, nuevas serán añadidas en el futuro. Aún así, estas funciones son totalmente independientes entre ellas y no hay necesidad de esperar a que un navegador soporte todas para disfrutar de sus beneficios. Si una app no soporta el *App Manifest*, que es uno de los requerimientos para aportar la instalabilidad, pero en cambio soporta el almacenamiento de caché, esta funcionalidad será válida igualmente, y si posteriormente soportara el *App Manifest* se añadiría la posibilidad de instalarse. De esta forma, vemos cómo todos los navegadores pueden ir soportando las PWA progresivamente, y como éstas a su vez pueden ampliar gradualmente sus funcionalidades sin limitaciones externas.

Con la multitud de dispositivos que existen hoy en día, el diseño web ha tenido que aprender a crear diseños que se adapten automáticamente a todos los tamaños de pantalla sin romper la estructura de la página. Este tipo de diseño se llama **responsive** y claramente se utilizará para las PWA. De esta forma, el código es respetuoso tanto para ordenadores y aplicaciones de escritorio como para móviles y cuando se instale podrá verse como una app nativa. Esta propiedad junto con los atributos que proveerá el App Manifest, que veremos en la siguiente sección, el usuario sentirá que se le ofrece una **experiencia de app nativa**.

Google, quien ha impulsado esta tecnología desde su creación, ha añadido también las PWA en su Google Play Store, lo cual potencia esta tecnología y fuerza a los navegadores competidores de chrome a adoptarla para no perder mercado. Esta opción se suma a la **accesibilidad** y **ligereza** que garantiza encontrar una PWA navegando por

Internet o como es compartirlas enviando su link y al abrirlo directamente sugerirá instalarla. El sistema está realmente enfocado a tentar al usuario para que añada la aplicación web su dispositivo.

La **instalabilidad** de las PWA tienen el poder de atraer al usuario, las estadísticas hablan por sí solas en cuanto el tiempo que pasa un usuario en una página web frente al que pasa en una app nativa. A esto se le añade la facilidad de **actualización**, ya que mientras las apps nativas deben pasar por un proceso de validación y luego reinstalarse en el dispositivo, las aplicaciones web tanto solo basta con cambiar el código en el servidor. Además, aparece un nuevo factor que también lo favorece: las "**Push Notifications**". Estas se usan para notificar al usuario de nuevo contenido incluso cuando la aplicación está cerrada. Si este contenido es por ejemplo la respuesta de un mensaje, una nueva noticia o una petición de amistad, el usuario siente la necesidad de abrir la notificación y vuelve a lanzar la aplicación, por lo tanto el usuario se reengancha con la aplicación web.

Otra propiedad que surge de un requerimiento es que es **seguro** debido a que debe servirse a través de **HTTPS**. A causa de que el service worker tiene la habilidad de interceptar las peticiones de la red y modificar la respuesta desde el navegador (el cliente), se utiliza HTTPS para evitar espionaje y garantizar la integridad del contenido.

Google descubrió que el 53% de los usuarios abandonan una web si la página tarda más de 3 segundos en cargar, por lo que una PWA debe mostrarse **rápida** para no interrumpir la experiencia de usuario. Para solucionar este problema los service workers pueden gestionar realmente bien la velocidad para gestionar el contenido decidiendo entre la memoria caché o la red en función de la conectividad. De esta forma aseguramos **fiabilidad** sin perder rapidez. Si a esto le sumamos la sincronización en segundo plano ("**background sync**"), aseguramos integridad en las acciones que requieran internet, ofreciendo así una experiencia mucho más completa.

Una vez presentadas las características principales de una PWA es momento de describir cómo se consigue esto a nivel técnico. Hay dos componentes técnicos fundamentales para poder agregar este potencial a las aplicaciones web estándar y son: el **App Manifest** y los **Service Workers**. Realmente existe un tercero que es estar servido con **HTTPS**, pero ya he explicado la razón en este punto. A continuación, explicaré detalladamente en que se basan estos dos nuevos componentes y qué nos aportan.

2.1.2. App manifest

El app manifest, en castellano el manifiesto de la aplicación, es simplemente un JSON que contiene información sobre nuestra aplicación que nos servirá para instalarla y definir su formato en los diversos dispositivos para poder tratarla como una app nativa más. Su función principal es informar a los distintos dispositivos que vayan a instalar la aplicación web cuál es el aspecto que debe adoptar, es decir, nos dirá cual es el icono que deberá mostrarse en la pantalla, si ocupará pantalla completa o dejará la barra de navegación y este tipo de variaciones. En el **Anexo 1**, podemos encontrar la explicación más detallada de los campos que se definen en este manifiesto y como afectan al aspecto de la PWA.

Una vez el navegador ha descargado el App Manifest y lo ha podido parsear será capaz de interpretar esta información en cada dispositivo. El formato del JSON, suele tener esta estructura:

```
{
  "short_name": "Weather",
  "name": "Weather: Do I need an umbrella?",
  "description": "Weather forecast information",
  "icons": [
    {
      "src": "/images/icons-192.png",
      "type": "image/png",
      "sizes": "192x192"
    },
    {
      "src": "/images/icons-512.png",
      "type": "image/png",
      "sizes": "512x512"
    }
  ],
  "start_url": "/?source=pwa",
  "background_color": "#3367D6",
  "display": "standalone",
  "scope": "/",
  "theme_color": "#3367D6"
}
```

Figura 4: Captura de un ejemplo de App Manifest

El navegador interpretará la información de la siguiente manera:

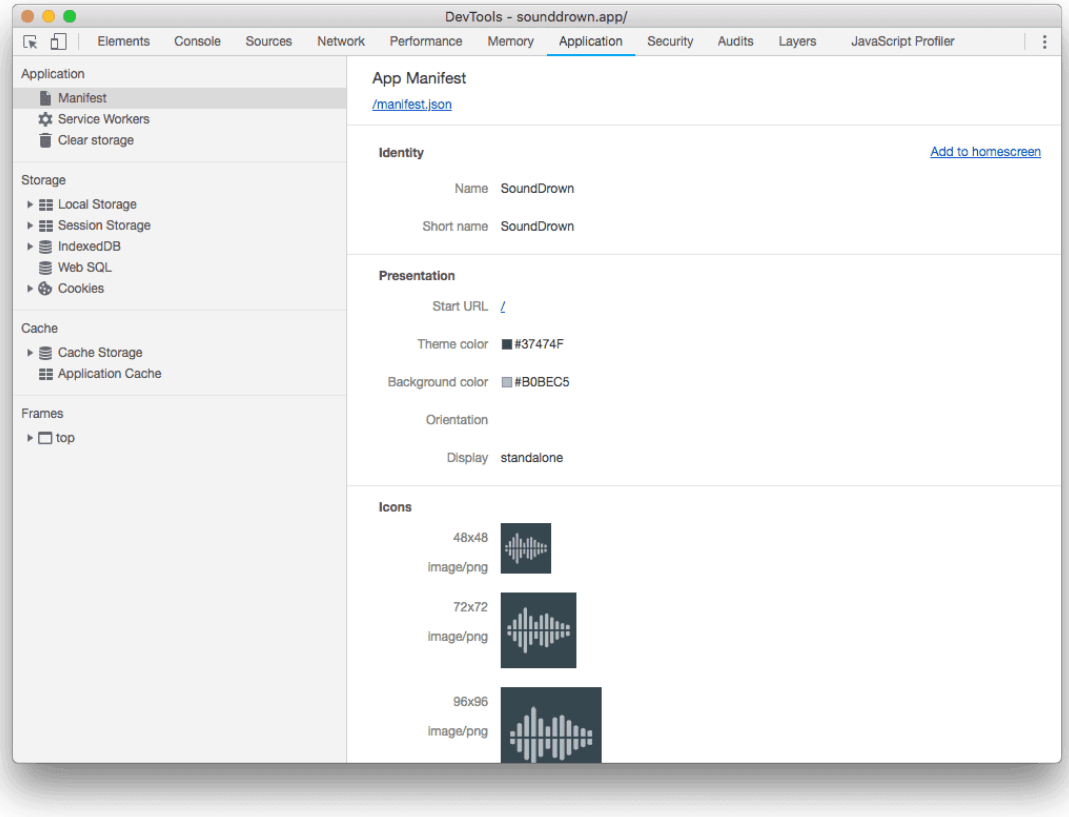


Figura 5: Navegador interpretando App Manifest

2.1.3. Service Workers

Los service workers son un tipo de "Javascript Worker" que permite ejecutar scripts del navegador en segundo plano. Es el encargado de cachear y servir los recursos solicitados por el navegador, incluso cuando está offline. Además, también agrega las funcionalidades de notificaciones "push" y sincronización en segundo plano (background sync). La notificaciones push son aquellas que permiten que los servidores envíen peticiones no solicitadas a los clientes. Mientras que la sincronización en segundo plano consiste en un servicio que permite diferir acciones hasta que el usuario tenga una conexión estable y de esta forma se garantiza que todo lo que el usuario quiere enviar se envía. Para tener una mejor idea sobre estos service workers, podemos imaginarlos como la persona en medio entre nuestro navegador (el cliente) y la red (el servidor), por lo tanto, actúa como un proxy. De esta forma, como todas las peticiones que van hacia la red o hacia la aplicación pasan a través de este worker, es capaz de interpretarlas con el propósito de gestionarlas junto con el caché y conseguir así ser más eficientes, proveyendo una experiencia mucho más amplia y flexible.

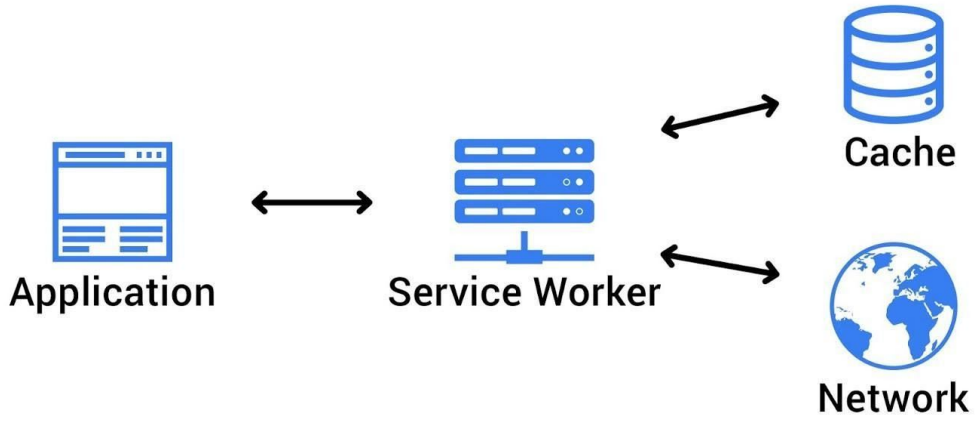


Figura 6: Esquema del ecosistema de un service worker

Con este potencial, nos permite decidir qué peticiones servir con la red o con el caché y poder ofrecer una experiencia de usuario ininterrumpida, independientemente de la conexión a Internet. Como bien sabemos, hasta ahora las aplicaciones web sólo pueden servir contenido si estamos conectados a la red, de lo contrario, el usuario vería el usual mensaje de "Error de la red: La página no ha sido cargada" o el juego del dinosaurio de Chrome sin conexión. Los services workers cambiaron las reglas del juego. A partir de este punto, las web apps tienen la capacidad de cachear las páginas webs y muchos otros recursos que aportan contenido a nuestro sitio más allá de simple texto. Estos recursos cacheados se guardan en un nuevo almacenamiento llamado *"CacheStorage"*, donde los service workers pueden acceder para comprobar los recursos cacheados y servirlos si se requiriera.

El service worker no es el único tipo de worker que se utiliza en la tecnología web. De hecho, existen tres tipos, incluyendo el service worker, y cada uno tiene un alcance y función. En el **Anexo 2** podemos encontrar una explicación sobre qué aporta cada uno de ellos.

2.1.3.1. Service Worker como un thread

Desde el punto de vista de los threads (hilos de ejecución), los service workers corren en uno distinto al del DOM, el cual renderiza la vistas.

Javascript es un lenguaje single-threaded, es decir, corre en un solo hilo de ejecución, por lo tanto, no es posible generar otro hilo como lo hacemos normalmente en Java, C o C++. Esto provoca que si queremos realizar una operación intensiva de la CPU en el hilo del DOM degradará la experiencia de usuario.

En Chrome, el motor de Javascript v8 genera dos hilos independientes: en uno correrá el DOM y en el otro el service worker. Cada uno está aislado y se trata como una instancia de máquina virtual con su propia pila de ejecución. De este modo, estamos creando dos instancias distintas cada una con su pila de ejecución independiente de la otra. Estas se ejecutan concurrentemente y solo pueden comunicarse entre ellos publicando mensajes en un registro de memoria que está compartidos para ambos con permisos de lectura/escritura. Dicho esto, para evitar obstruir la vista del navegador ejecutando operaciones pesadas en el hilo del DOM se aconseja trasladarlas a los workers. Cuando se hace, los dos hilos se ejecutan paralelamente sin entorpecer ni bloquear al otro, y cuando termina la operación, publicará un mensaje en el hilo del DOM, utilizando el método de PostMessage mencionado en el apartado anterior, para informar de que el trabajo ha finalizado y puede actualizar el DOM. El sistema operativo es el encargado de gestionar el multi-paralelismo y decidir cuándo y cómo cambiar los hilos de ejecución.

2.1.3.2. Ciclo de vida de los Service Workers

El ciclo de vida de los service workers es crucial para entender su naturaleza. Cabe mencionar, que los service workers son “event-driven”, es decir, reaccionan a eventos. Esto explicará más adelante su forma de trabajar. Los objetivos que se persiguen a la hora de usar un service worker son cuatro:

1. Habilitar el funcionamiento offline.
2. Permitir que un nuevo service worker se prepare en el navegador sin interrumpir el que funciona actualmente.
3. Asegurarse de que las páginas definidas dentro del alcance estén controladas por el mismo service worker (o por ninguno).
4. Asegurar que solamente hay una versión de la app web ejecutándose a la vez. Este punto es realmente importante, dado que si no se controlara, el usuario podría estar accediendo a dos versiones distintas de service workers que gestionan de manera distinta el almacenamiento y desencadenar en errores e incluso pérdida de información.

Con los objetivos sobre la mesa, ahora es momento de explicar el ciclo de vida de forma ordenada:

1. **Registro.**
   El primer paso del registro es comprobar si el navegador acepta Service Workers y si podemos trabajar con su API, ya que, como ya hemos mencionado previamente, es una tecnología nueva que los navegadores están integrando gradualmente. Otra de las comprobaciones que realiza es si el origen se sirve con

HTTPS que como veremos posteriormente es un requerimiento obligatorio. Si el navegador permite el service worker, el registro es un método que recibe como parámetro la localización del archivo que contiene la configuración de nuestro service worker. Devolverá una promesa que de resolverse el navegador será capaz de descargarlo, parsearlo y estará listo para instalarse. Una **Promise**, o promesa, es un objeto que representa la terminación o el fracaso de una operación asíncrona. En función del resultado, la promise se resolverá si consigue su objetivo o se rechazará si falla o se produce un error". En nuestro caso, si algo fallara en este proceso se pasaría al estado "redundant" y se descartaría el service worker.

2. **Instalación.**
   Una vez registrado, el service worker ya estará trabajando en un hilo aislado al principal y el primer evento que recibirá cuando se ejecute será el de instalación ("*install"*). Este evento solo se lanzará una vez por service worker, pero si el navegador considera que el archivo de configuración del worker ha variado también es capaz de volver a lanzarlo.
   Esta etapa es el momento perfecto para instalar todo el contenido que queramos precachear, como por ejemplo los recursos estáticos. El evento nos da la oportunidad de alargar la instalación tanto como deseemos para que pueda realizar todas las operaciones que consideremos oportunas. Técnicamente, se pasan una serie de promesas a un método del evento que esperará a que todas se resuelvan para dar el visto bueno y continuar. En caso de que al menos una promise fallase, la instalación fallaría y se pasaría al estado "redundant", por lo que una cantidad excesiva de contenido a precachear aumentaría la probabilidad de producirse un error y es aconsejable centrarse en precachear lo estrictamente esencial.
   En este paso solo estamos precacheando contenido, no estamos definiendo cuando tiene que servirse, por lo que todavía no funcionaría el servicio offline.
   Si al finalizar la instalación ya existiera un service worker activo y vinculado a ese origen, el nuevo service worker se mantendrá instalado a la espera de poder entrar en acción. En este punto aún no está en uso, es decir, quien sigue decidiendo el comportamiento del service worker y qué hacer con las peticiones recogidas es el anterior. Cuando se cierren todas las pestañas que utilicen el service worker viejo o si tiene tan solo una abierta la refresque, solo entonces, se activaría el nuevo.

3. **Activación.**
   El evento "activate" se lanza una vez la instalación ha terminado, el nuevo service worker está a la espera para ser activado y el anterior service worker, si lo hay, ya no está funcionando, ha pasado al estado de "redundant" donde ya no podrá volver a ejecutarse. Sin embargo, todo lo que haya cacheado se mantiene en la memoria.
   Esta etapa es ideal para realizar todas aquellas acciones que se nos impedía mientras el anterior service worker estaba en funcionamiento, como por ejemplo migrar las bases de datos y el limpiando caches. El funcionamiento técnico es similar al del paso anterior: el evento ofrece un método que puede recibir promesas y espera a que finalicen. Si resuelven activarán el service worker y

estará listo para controlar eventos como “fetch”, “push” y “sync” entre otros. Si alguna promesa fallara se descartaría el worker pasando al estado “redundant”.

4. **Idle.**
   Este es el estado en el que el service worker espera, ya activado y operativo, a que se lance algún evento y deba actuar.

5. **Terminated.**
   Este es el estado final que el service worker ejecuta automáticamente, pero no significa que sea la última etapa. Cuando el worker se mantiene en el estado de desocupado (“idle”) por un largo periodo, para liberar carga del navegador se mueve a este estado en el que duerme. Además, aquí olvida el caché que ha guardado pero continúa escuchando eventos por si recibe uno nuevo y tiene que reactivarse. Si se quisiera que algunos datos perduraran, deberían migrarse a la IndexedDB para luego volver a guardarse en la memoria caché al despertarse. La IndexedDB es una base de datos que nos permite almacenar cantidades significativas de datos estructurados, incluyendo archivos y blobs.

6. **Fetch/Message.**
   Aunque hay más métodos a los que reaccionará el service worker hablaremos sobre el “fetch” que es una de claves de los service workers.
   Cualquier petición que se realice a la red lanzará este evento. Este evento permitirá que la PWA gestione el contenido en escenarios sin conexión a la red o de mala conectividad. Como he mencionado, el service worker es como un proxy que intercepta todas las peticiones, por ser “event-driven”, y será capaz de decidir cómo proceder. Hay muchas posibilidades: comprobar si tenemos la respuesta en caché, si no está ir a la red a buscarla, quizás ir directo a la red o solo a la caché, etc. Estas posibilidades se han estandarizado y se definirán en el siguiente apartado. En esta etapa, el service worker activado ya tiene poder de decisión y nos brinda la oportunidad de completar la experiencia de usuario sin tener que depender de la red, una característica que nos acerca mucho a las apps nativas. Una vez resuelta la petición volverá al estado de “Idle” a la espera de otro evento.

Para tener una idea más global del recorrido que ha realizado el service worker de inicio a fin el siguiente desglose de estados servirá para esclarecer más el concepto:

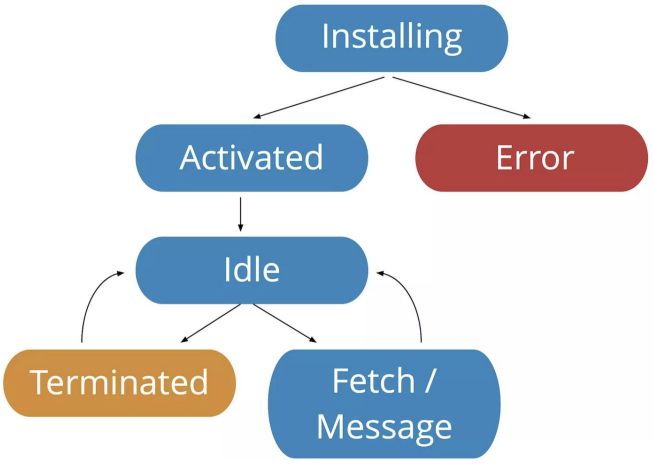


Figura 7: Ciclo de vida del Service Worker

En el **Anexo 3** podemos encontrar el ciclo de vida llevado a la práctica, con el desarrollo a nivel técnico, código, recomendaciones, imágenes y conclusiones de los resultados de un experimento en el que pruebo a bajo nivel que el comportamiento del service worker es como se ha definido en este punto.

2.1.3.3. Workbox

Una vez presentados los service workers, uno se da cuenta de lo potente que es esta herramienta y de la cantidad de posibilidades que nos brinda. Teniendo en cuenta que los service workers son el core tecnológico para potenciar las PWA, era simplemente cuestión de tiempo que aparecieran nuevas librerías e instrumentos con el objetivo de crear un patrón de buenas prácticas y facilitar la tarea de configuración de dichos workers. Además, un enorme número de aplicaciones compartían la misma implementación inicial, por lo que aparecieron también herramientas que proveían una plantilla para potenciar una web app rápidamente con esta tecnología disruptiva, sin tener que empezar de cero y tener que entender qué pasa a bajo nivel.

A medida que los developers empezaron a adoptar las PWA, surgió la discusión sobre las estrategias de cacheo, entendiendolas como patrones que determinan cómo un service worker procede al recibir un evento “fetch”. El paso del tiempo incrementó la experiencia en este tema y se distinguieron unos cuantos patrones. Emergieron varias herramientas como “sw-precache” y “sw-toolbox” y Google finalmente presentó Workbox como el sucesor definitivo.

Workbox es una colección de librerías y módulos para Nodejs que facilita la metodología para cachear recursos y saca partido de herramientas para controlar el precacheo, el enrutamiento y el cacheo en tiempo de ejecución, entre muchas otras, con el objetivo de construir una PWA realmente sólida que aproveche al máximo todas sus funcionalidades. Workbox ha evolucionado hasta organizarse en módulos, donde cada módulo se encarga de gestionar las herramientas sobre una función en concreto. De esta forma, solo es necesario importar los módulos que deseemos a medida que se requieran.

Vamos a analizar las distintas estrategias de cacheo que plantea Workbox:

- **Cache Only.** La primera estrategia presentada es una de las más simples. Cache Only asegura que la respuesta proporcionada al evento fetch provenga de la caché. La primera impresión puede parecer que sea una estrategia extraña o incluso un sinsentido pero en realidad lo tiene, ya que si hemos precacheado recursos en la etapa de instalación, como fuentes, imágenes o cualquier contenido estático, podemos servirlas sin necesidad de pedirlas a la red.

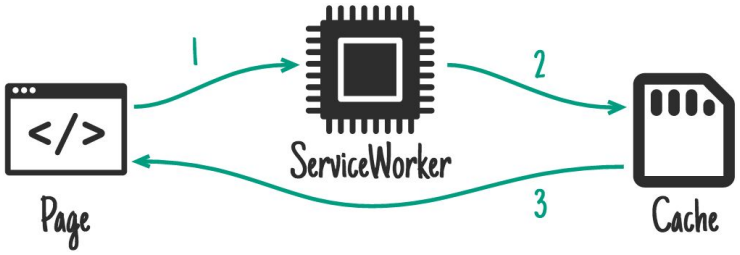


Figura 8: Esquema de la estrategia Cache Only

- **Network Only.** Como se puede deducir, esta estrategia repite el patrón de la anterior pero esta vez asegurando que la respuesta proviene de la red. Esta herramienta es comúnmente utilizada para casos en los que no podemos confiar en los datos cacheados o no existe una equivalencia offline y actualizar la información con la red es una obligación. Por ejemplo, si el estado del usuario de un chat está activo o no, no tiene demasiado sentido servirlo con información cacheada.

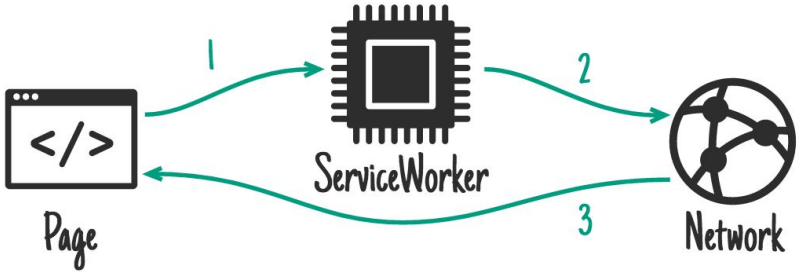


Figura 9: Esquema de la estrategia Network Only

- **Cache First (Cache falling back to network).** Este tipo de estrategia es uno de los que faculta de capacidades offline a las web apps. Básicamente, el service worker recibe el evento “fetch” y primeramente intentará obtener la respuesta del caché. Si es satisfactorio, el service worker podrá contestar la petición sin necesidad de interactuar con la red. De lo contrario, si no encuentra los datos en el caché, se realiza un segundo paso, donde el service worker completa la petición buscando, esta vez, la información en la red. En este caso, la respuesta será cacheada para que las próximas veces pueda servirlas directamente desde la caché. El siguiente esquema ayuda a entender el proceso paso a paso. Es importante mencionar que esta herramienta se recomienda para recursos que no son críticos y pueden ser cacheados gradualmente, mientras que si tenerlos cacheados es una necesidad se debería optar por la estrategia *Cache Only* y precachearlos en la instalación.

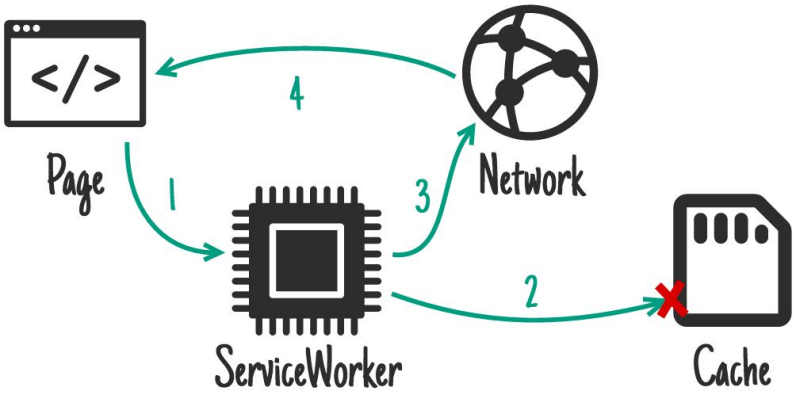


Figura 10: Esquema de la estrategia Cache First

- **Network First (Network falling back to cache).** Esta estrategia se basa exactamente en los mismos principios que la estrategia anterior pero ahora la prioridad es la red, es decir, cuando el service worker recibe un evento reenviará la petición a la red. Si obtenemos respuesta se añadirá al caché y será servido al

cliente. De esta forma, el caché se renueva continuamente siempre que la red aporte contenido actualizado. De lo contrario, si la petición a la red fallase, irá a buscar la respuesta a la memoria caché, pero es relevante saber que esto ocurrirá sólo cuando la red responda y esto puede tomar su tiempo ya que depende de diversos factores como por ejemplo la cobertura del dispositivo móvil. En resumen, esta estrategia es adecuada para recursos que están frecuentemente actualizados, por ejemplo el contenido de una red social como avatares, artículos, posts y demás. Esto significa que ofrecerá a los usuarios online el contenido más actualizado mientras que a los usuarios offline se les servirá el contenido cacheado, idealmente bastante actualizado.

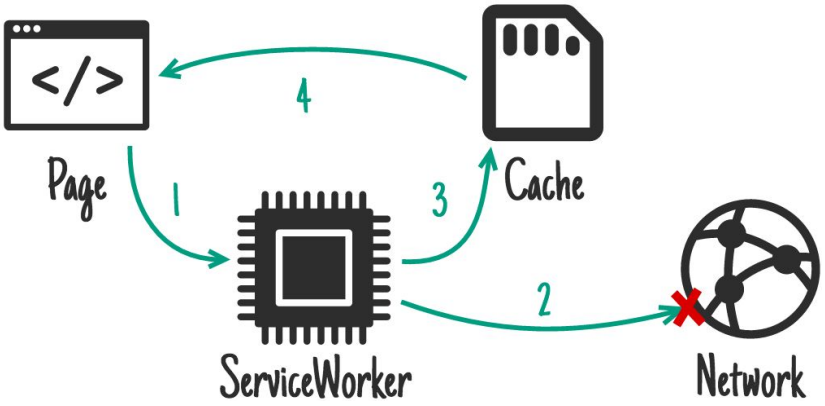


Figura 11: Esquema de la estrategia Network First

Stale-While-Revalidate.

Este patrón construye la solución ideal para cuando deseamos actualizar constantemente los recursos pero tener la última versión no es estrictamente esencial, por ejemplo las imágenes de los avatares. Así que, esta estrategia fuerza al service worker a construir la respuesta con caché si está disponible y mientras tanto irá a la red a actualizar su respuesta en caché para la próxima petición. Por lo tanto, desde el punto de vista de la petición se resuelve rápidamente si está cacheada. En otras palabras, esta estrategia no actualiza la página cada vez que la red retorna información pero sí que renueva el caché para que la nueva información esté disponible al refrescar.

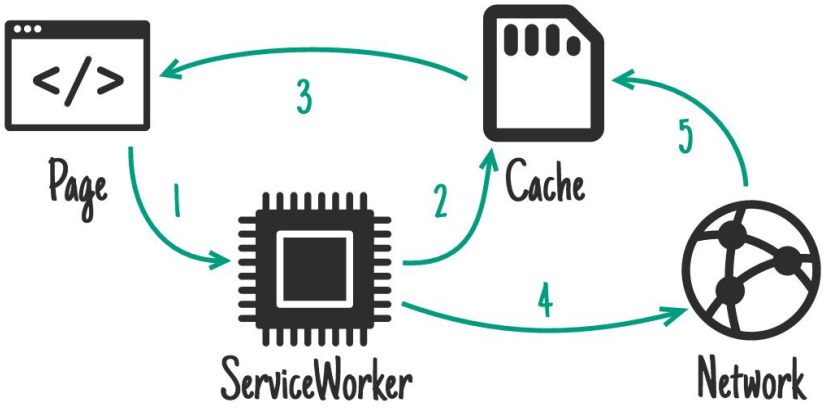


Figura 12: Esquema de la estrategia Stale-While-Revalidate

Como hemos visto hasta ahora, Workbox facilita mucho el desarrollo de PWAs. Además, al estructurarse modularmente permite combinar varias herramientas para el uso que el desarrollador desee, permitiendo también añadir plugins que configuran más aún el service worker. Un caso de uso práctico genérico sería el siguiente:

- Primero se define para qué rutas se aplicará el comportamiento (ya sea con una expresión regular o bien definiendo la extensión de los archivos).
- Después se añade la estrategia y esta nos permite añadir más opciones como el nombre y plugins.

En el ejemplo de abajo veremos cómo con unas pocas líneas podemos definir una estrategia con Workbox y si se ha revisado el **Anexo 1**, también se ve la cantidad de código que ahorra. En el ejemplo se estipula que para todas las rutas que cumplan con la expresión regular "/images/", se aplicará la estrategia de "Cache First". También vemos que el caché se guardará con el nombre "image-cache". Finalmente se ha añadido el plugin de expiración, el cual define el máximo de requests que almacenará y por cuánto tiempo. No tenemos que preocuparnos de cómo lo hará, simplemente el sistema ya está optimizado para realizar esta tarea y utilizará todos los recursos del navegador que disponga para llevarla a cabo, como la memoria caché, IndexedDB, Background Services (explicados a continuación del código), etc.

```
import {registerRoute} from 'workbox-routing';
import {CacheFirst} from 'workbox-strategies';
import {ExpirationPlugin} from 'workbox-expiration';

registerRoute(
  new RegExp('/images/'),
  new CacheFirst({
    cacheName: 'image-cache',
    plugins: [
      new ExpirationPlugin({
        // Only cache requests for a week
        maxAgeSeconds: 7 * 24 * 60 * 60,
        // Only cache 10 requests.
        maxEntries: 10,
      }),
    ]
  })
);
```

Los Background Services son una colección de herramientas que permiten que una página web envíe y reciba actualizaciones incluso cuando esta no está abierta. Funcionalmente se puede interpretar com un proceso en segundo plano. Google los agrupa en cuatros tipos:

- **Background Fetch.** Habilita la posibilidad de descargar recursos pesados, como películas o podcasts, en segundo plano.

- **Background Sync.** Garantiza que enviará todos los datos a la red una vez se haya restablecido la conexión de forma fiable. Mientras, almacena dichos datos en segundo plano. Este es el servicio que se ha implementado en el proyecto.
- **Notifications.** Después de que un service worker reciba un Push Message, utilizará este servicio API para mostrar los datos al usuario.
- **Push Message.** Para mostrar una push notification a un usuario, un service worker debe primero usar la API de Push Message para recibir los datos de un servidor. Cuando se reciben, entonces podrá utilizar la api de notificaciones para mostrar dicha información.
- **Periodic Background Sync.** Esta API permite registrar tareas que un service worker ejecutará en intervalos periódicos con conectividad a la red. Esto permite refrescar el contenido de las aplicaciones web en segundo plano para mostrar a los usuarios contenido actualizado constantemente.
- **Payment Handler.** Este servicio abre un nuevo ecosistema a los proveedores de pagos, ya que permite que una aplicación web utilice un service worker para actuar como método de pago y se integre en las páginas web comerciales.

2.2. React

React es una librería de Javascript creada por Jordan Walke, un ingeniero de software de Facebook, en el año 2011. Rápidamente se implementó en Facebook, al año lo siguió Instagram y desde entonces no ha para expandirse. React tiene una serie de características técnicas que lo hacen realmente diferente del resto de competidores:

- **Reusable.** Se estructura en componentes, por lo que resulta completamente modular y esto facilita mucho el desarrollo web, ya que se repiten constantemente patrones y elementos que si se ha programado con un ejercicio previo de abstracción de la lógica se puede reutilizar una gran cantidad de código, tanto a nivel del UI como funcional.
- **Reactivo.** Cada componente gestiona a lo largo de su ciclo de vida un estado que varía en función del flujo de la aplicación y define su comportamiento. En definitiva, la aplicación reacciona ante la variación del estado de sus componentes.
- **Simple.** A diferencia de frameworks como Angular que utilizan motores de plantilla para reinventar algoritmos básicos de programación como “if” o “for” y incrustarlo con una mezcla de JS en HTML, React utiliza su propia extensión de Javascript, llamada JSX, para hacerlo a la inversa, escribiendo “pseudo HTML” directamente en el Javascript, lo cual le permite juntar lógica y maquetado en un mismo archivo. Con esta potente herramienta y junto a las librerías de terceros que se necesiten a demanda, React se libera del enorme set de herramientas incluidas por defecto en otros framework, permitiendo así que sea mucho más ligero, eficiente y rápido.

La filosofía de React, así como su aplicación en esta tesis, se ha explicado detalladamente en el **Anexo 5**, ya que se ha vinculado con el código del proyecto conllevando una extensió considerándose mejor trasladarla a una anexo.

3. Desarrollo del proyecto

En esta sección desglosaré y explicaré ordenadamente el proceso de construcción de mi PWA, aplicando todas las tecnologías presentadas en el apartado del Estado del arte y, como veremos, otras más que se fueron necesitando a medida que el proyecto avanzaba.

3.1. Flujo de la aplicación

Cuando empecé a pensar cómo debería ser la aplicación me plantee la siguiente experiencia de usuario:

- Primero, el usuario debería encontrarse con una pantalla para autenticarse en la aplicación con un un formulario de correo y contraseña. En caso de no disponer de una cuenta, el usuario debe poder acceder a una ventana para registrarse.
- Una vez autenticado, el usuario será redirigido a la página inicial, donde encontrará sus historias sociales y las de otros usuarios que han hecho públicas. En caso de no haber creado ninguna aún aparecerá una lista vacía. En esta pantalla será posible filtrar las historias sociales según el título o las etiquetas que contenga.
- En la esquina superior derecha encontrará un menú que desplegará un panel lateral para poder navegar a las otras vistas o cerrar sesión.
- La otra vista planteada inicialmente es la de creación de una historia social. En esta vista podremos añadir pictogramas representados por: un pictograma de la API de Arasaac, una fotografía o un dibujo. Iremos añadiendo pictogramas y construyendo las filas y se proporcionará una herramienta para acceder a la cámara del dispositivo, una vista para poder dibujar y buscador de pictogramas. Una vez definida la historia para terminar el primer paso del creador de historias sociales, incluiría un formulario para añadir un texto sobre los pictogramas y se indicaría el número de imágenes que habría por fila.
- El segundo paso de la vista del creador sería un formulario para definir un título y una descripción sobre la historia y se le preguntaría si quiere compartir la historia social con la comunidad de usuarios de la aplicación.
- Por último, el tercer paso nos permite crear etiquetas o usar las que ya hemos creado para asociar las historias a temas concretos, como podrían ser temas familiares, amigos, deportes, etc.
- Al finalizar, el usuario será redireccionado a la vista inicial donde podrá ver su historia para trabajar con ella y, si así se quisiera, borrarla o editarla.

Una vez pensado el flujo, es momento de trasladarlo a un diseño para decidir colores, distribución y empezar la batalla entre ergonomía, utilidad y diseño. Aunque la idea inicial de la app sufrió algunas variaciones, unas a causa de la implementación de soluciones mejores y otras no se llegaron a implementar por falta de tiempo, en el Anexo 4 se ha incluido el diseño inicial de la interfaz de usuario hecho con el programa Sketch.

Mencionar que a la aplicación se la bautizó como “Bilingualy”, nombre con el que también se le referirá a partir de ahora.

3.2. Control de versiones

En el desarrollo de software existe una metodología que resulta imprescindible: el control de versiones. El control de versiones nos permite registrar los cambios realizados sobre un archivo o conjunto de archivos, de modo que establece una versión específica que podremos recuperar más adelante. Así pues, si por error eliminamos un archivo o lo modificamos de tal manera que la aplicación no funciona correctamente, siempre tenemos la opción de volver a una versión anterior estable o incluso con funciones más avanzadas que nos permiten elegir qué archivos queremos restaurar, manteniendo los otros cambios. Luego todas modificaciones se pueden subir a un repositorio remoto que contiene una copia de todos las versiones que hemos subido. Otra herramienta fundamental es el trabajo en ramas, el cual permite trabajar en paralelo en distintas partes o archivos del proyecto para luego poder juntar todo el código en una única versión. Esta es la forma en la que trabajan desde las pequeñas startups hasta las grandes empresas del sector tecnológico.

3.2.1. Git

Existen varios sistemas de control de versiones como Git, SVN, Mercurial, etc. El más utilizado actualmente es Git y también es el que se ha empleado en este proyecto, pues dispone de todas las herramientas necesarias para el desarrollo y su amplia comunidad y madurez permiten aprender a usarlo realmente rápido.

Por otro lado, para poder almacenar el proyecto en la nube se precisa de una plataforma web compatible con este protocolo. Son muchas las soluciones gratuitas y completas, como por ejemplo Bitbucket o GitLab pero mi elección fue Github. Las prestaciones entre ellos eran muy comunes pero ya había utilizado previamente esta web y tenía buenas referencias de otros compañeros.

En cuanto al protocolo para organizar el versionado no hay un patrón único ni una solución absoluta. Consideré que era importante empezar con una buena organización desde el principio y, tras una pequeña investigación, acabé encontrando una publicación de *Vincent Driessen* titulada “A successful Git branching model” en la que decidí basar mi modelo. En el Anexo 5 explico como se plantea este sistema de ramificación y cómo se ha aplicado al proyecto.

Como se describe en el modelo, Bilingualy consta de las ramas principales, **master** y **develop.** Adicionalmente, se crearon cuatro ramas más que agruparon el resto de funcionalidades necesarias para la aplicación:

- **feature/dashboard.** Donde se ha creado las vista principal y origen de la aplicación, así como todos sus componentes necesarios. Además, la mayor parte del desarrollo de Redux se ha creado en esta feature.
- **features/create.** En esta rama se creó toda la vista de creación de una historia social, la integración con la API de Arasaac y el sistema “grid” para poder organizar los pictogramas.
- **feature/firebase.** En esta parte se desarrolló la integración con firebase para gestionar el sistema de autenticación y el sistema de almacenamiento online en firestore. Por lo tanto, también se creó el servicio de llamadas a dicha API.

- **feature/service-worker.** Finalmente, se creó esta rama para configurar el service worker con Workbox y añadir la configuración del App Manifest necesaria.

El código completo se puede encontrar en el siguiente repositorio:

https://github.com/gerardcastell/bilingualy

3.2.2. Netlify

Cuando se desarrolla un proyecto de software se suele desplegar en local para poder ver los resultados de lo que estamos programando al instante. Sin embargo, como hemos visto en el apartado de los service workers, uno de los requerimientos es que la aplicación web se debe servir con HTTPS. Esto no supone un problema ya que existen una gran cantidad de paquetes que puede desplegar el contenido en local sirviéndolo con HTTPS.

A medida que la aplicación crecía, necesitaba que la gente la evaluara, pues no tenía tiempo de realizar los test unitarios, de integración, end-to-end, etc. y, además, aportaba mucho valor que otros usuarios la probarán para ver, por un lado, si la interfaz estaba siendo intuitiva y gustaba y por otro lado detectar “corner cases”, es decir, casuísticas que no había pensada y que la app no implantaba o lo hacía mal.

Dado que lo interesante era que la app se pudiera acceder desde cualquier dispositivo, ya sea ordenador, tablet, móvil… y a su vez, disfrutar de la instalabilidad y las funcionalidades que nos ofrece la Progressive Web App, decidí utilizar Netlify. Netlify, es una plataforma web que permite conectar un repositorio remoto, en mi caso Github, para servir el contenido de la branch que deseemos públicamente en un dominio que nos dará él si no disponemos de uno propio. Entre el kit de herramientas que nos ofrece, encontramos también la posibilidad de desplegar las Pull Request. Las Pull Request son un concepto propio de Git que, en esencia, representa la intención de hacer un merge entre dos ramas, pero que antes de realizarse se solicita que los compañeros accedan a los cambios y la aprueben. De esto modo, no se realiza el merge de ninguna rama sin que un equipo lo revise previamente. Para mi proyecto, la Pull Request no tiene demasiado sentido ya que básicamente el equipo soy solo yo, pero en Netlify tenía un gran uso porque antes de lanzar la versión estable en master, habría una Pull Request desde develop y Netlify me ofrecía un link donde podía comprobar online el resultado del despliegue. De este modo, obtenía como un entorno beta (previo al entorno de producción), donde podía comprobar si las vistas se ajustan a los diferentes dispositivos, la tecnología funcionaba bien o encontraba algún error. Si alguno de estos casos se daba, implementaba la solución y la añadía a la Pull Request hasta que funcionase adecuadamente. En ese momento, disponía de una versión estable y cerraba la Pull Request, se realizaba el merge a master y el de producción ofrecido por Netlify actualizaba su contenido con la versión de master y lo servía con HTTPS. Este link se puede visitar en cualquier momento y es el siguiente: https://bilingualy.netlify.app/

El siguiente esquema representaría la pipeline de CI/CD que más o menos he generado, quitando el bloque de los Unit Test:

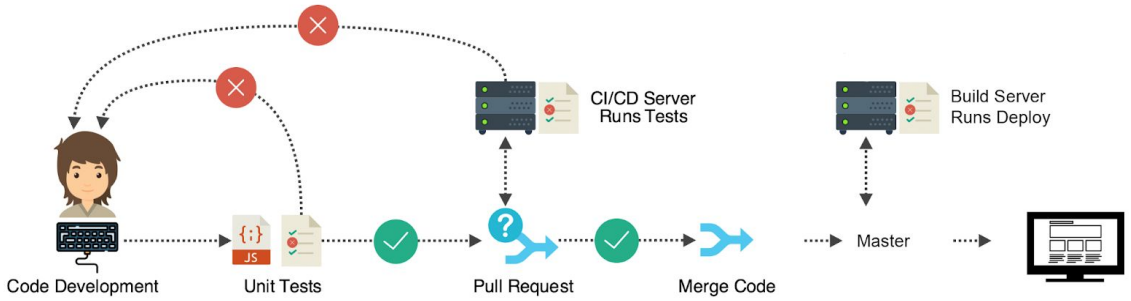


Figura 13: Diagrama de la pipeline de Bilingualy

3.3. Framework7

Cuando se empieza un proyecto con React, se necesitan una serie de configuraciones en el proyecto que no vienen por defecto con su librería. Para ello existen herramientas como *CreateReactApp(CRA)*, el cual nos ofrece un moderno setup inicial para el proyecto sin configuración adicional. Es una de las soluciones más rápidas y sólidas para construir aplicaciones no demasiado complejas y además, es la solución oficial ofrecida por React, el cual da soporte.

Entre los módulos y configuraciones que aportan *CRA* debemos fijarnos en los siguientes:

- **Babel.** Babel es un transpilador o transconspilador que se encargará de compilar el Javascript de estándares superiores a ES5 y React, es decir, JSX. a Javascript ES5.
- **Webpack.** Se encargará de cargar Babel, pero más importante aún es un bundler. Un bundler es un empaquetador de módulos que en este caso se utiliza para el Javascript, el CSS, imágenes, etc. Realmente se trata de una herramienta fundamental cuando se trabaja con apps web que siguen una filosofía modular. Webpack se encargará de gestionar dichos módulos y sus dependencias y será incluso capaz de de optimizar el sistema realizando concatenación de código, minimización y ofuscación, entre muchas otras funciones.

Aún existen muchos más atributos con que *CreateReactApp* facilita, en enorme medida, el desarrollo de una app con React y por las que parece una elección casi trivial para desarrollar un proyecto, aunque existe un pero. Con la expansión de las PWA no tardaron en acoplar esta tecnología a su setup y, más adelante también se integró Workbox pero con su configuración cerrada. El argumento fue que su herramienta pretendía ofrecer una solución sencilla para crear pequeñas aplicaciones con React y ofrecer una configuración abierta de Workbox podría causar problemas que dificultarán a usuarios menos expertos en esta tecnología su avance con React.

Con esto conflicto tuve que realizar una pequeña investigación para ver cuáles eran las alternativas y se plantearon las siguientes:

- Construir el setup con CRA y “eyectarla”, lo cual expondrá toda la configuración y la dejará a manos del desarrollador pero implicaría responsabilizarse del mantenimiento. Una vez eyectada,debería modificarse la configuración de

webpack para inyectar Workbox en su proceso de bundler, lo cual tampoco era tarea fácil y fue descartada.

- Otra opción era realizar un “fork”. Básicamente, consistía en bifurcar la última versión estable de CRA para modificar lo que necesitemos en su configuración y luego fusionar los próximos cambios para no tener que mantenerla por nosotros mismos. Parecía una solución mejor que la anterior, pero las soluciones ya implementadas por otros desarrolladores hacían depender el proyecto de un tercero, que sería muy frágil y no tenía los conocimientos ni el tiempo para implementar la bifurcación yo solo. Por lo tanto, también se descartó.
- Otra solución muy común entre los desarrolladores consistía en instalar tres librerías (“react-app-rewired”, “react-app-rewired-workbox” and “workbox-webpack-plugin”), las cuales permitían crear un archivo que sobrescribía la configuración por defecto para que apuntara a otro archivo a la hora de leer la configuración del service worker, y en éste inyectar Workbox. Existían muchos papers con variaciones sobre este método pero se acabó descartando porque forzaba a utilizar una vieja versión de workbox y no garantizaba todas sus funcionalidades.

Actualmente, existen varias Pull Requests abiertas en el repositorio de CRA que cumplen con casi todos los requerimientos para ser “merged” pero falta que el equipo de el visto bueno. Algunas de ellas son las siguientes:

https://github.com/facebook/create-react-app/pull/5369.

https://github.com/facebook/create-react-app/pull/8485

Esta es la razón por la que acabé utilizando Framework7. Se trata de una librería de componentes de Javascript gratuita y de código abierto para desarrollar aplicaciones móviles, aplicaciones web y de escritorio con una apariencia nativa. Pero lo que realmente es interesante para el proyecto es que provee un setup para React que no llega a ser tan óptimo como el de CRA pero incluye Webpack y Babel, las dos piezas fundamentales, y todavía más importante, incorpora la última versión de Workbox y deja libre su configuración para el desarrollador. Estamos hablando de un solución que contiene algunas limitaciones para React, como que la arquitectura de las vistas está definida y debe seguirse para que la app funcione y, por poner otro ejemplo, el router de navegación también está configurado y no puede modificarse. Aún así, nos aporta la función fundamental a investigar en esta tesis, el desarrollo de una PWA. Por lo tanto, evaluando este “back-and-forth” entre ventajas e inconvenientes acabé razonando que Framework7 era la mejor elección para poder evaluar así el comportamiento de la PWA, aunque tenga menos libertades en cuanto a la arquitectura de React se refiere.

3.4. Mobile First y Responsive UI

Hoy en día, cuando se desarrolla una app web se debe tener en cuenta la gran variedad de dispositivos en los que se servirá y, por lo tanto, la aplicación debe mostrarse adecuadamente en todas ellas para seguir siendo intuitiva, útil y ergonómica. Esta propiedad de las aplicaciones webs para adaptar su distribución y forma en función del tamaño de la pantalla se denomina “**responsive**”. En el caso de las PWA aún debemos incidir más en esto ya que pretendemos que parezcan app nativas en los móviles. Es

importante matizar que cuando se diseña primero el UI en móvil para luego adaptarlo a web se denomina "**mobile first**", y este es el patrón de diseño que se ha seguido para Bilingualy, puesto que se priorizaba su versión en mobile frente a web.

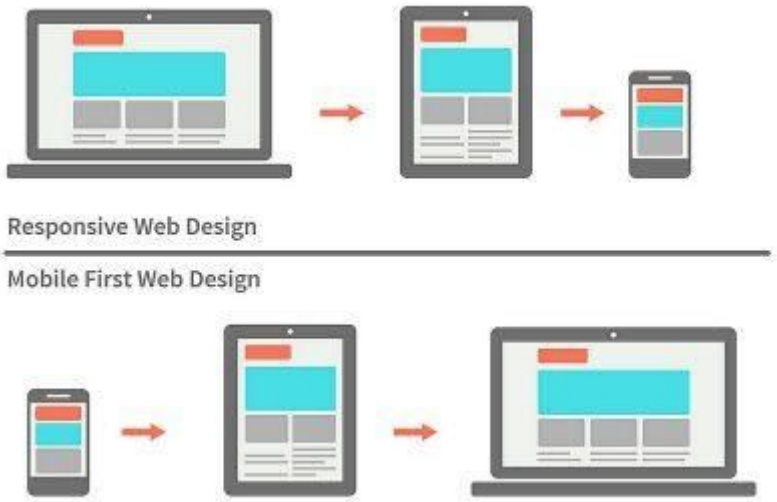


Figura 14: Enfoque del Responsive Design respecto a Mobile First Design

Para Bilingualy se han utilizado en total tres estrategias:

- Los **componentes de Framework7**. Estos componentes están diseñados con una filosofía de "mobile first", lo cual facilita mucho la tarea del diseño y permite centrarse más en la funcionalidad.
- **Sass** es un preprocesador de CSS. Básicamente, se trata de lenguaje de scripting que extiende las funciones de CSS básico y luego lo compila a CSS normal. Nos permite introducir lógica, facilita mantener un código más limpio y crear piezas y variables reusables.
- Combinando las **media queries** y los **breakpoints**, podemos definir distintos rangos de tamaños de pantalla y aplicar para cada rango un estilo de CSS distinto. De esta forma, podemos definir un diseño distinto para cada de pantalla.

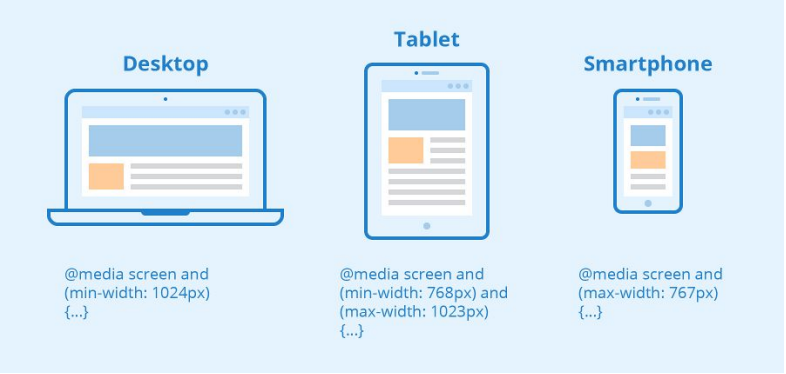


Figura 15: Diseño adaptado a distintos dispositivos con media queries

3.5. Backend y librerías de terceros

Cuando se desarrolla una aplicación web, en muchas ocasiones se suele necesitar una base de datos para almacenar toda la información de usuarios, datos y demás, y un sistema de backend que gestione las llamadas para servir esta información. Además de

esto, también podemos precisar de prestaciones que ofrecen terceros. Dichos servicios suele proveer una API donde encontramos los endpoints para aprovecharnos.

3.5.1. API de Arasaac

Para Bilingualy se ha precisado de la **API de ARASAAC**. Se trata de una organización sin ánimo de lucro del gobierno de Aragón que ofrece muchos servicios gratuitos sobre Comunicación Aumentativa y Alternativa, como una API para sus pictogramas. Con este servicio, Bilingualy es capaz de realizar búsquedas de pictogramas que se relacionan con una palabra clave y ofrecer todas las imágenes vinculadas a esta palabra. Las imágenes permitirán que el usuario pueda construir las historias sociales con las características que el desee.

3.5.2. Firebase

Por lo que respecta al backend, no entraba en el alcance de este proyecto crear un backend desde cero. Aún así, aprovechando que la aplicación web se servía en un dominio público gracias a Netlify, ofrecía un gran valor al producto que tuviera su sistema de autenticación y base de datos para poder almacenar las historias sociales de cada usuario. **Firebase** fue el elegido para este servicio. Firebase es un "Backend as a Service (BaaS)" y nos permite prescindir del desarrollo ya que ofrece un sistema implementado realmente fácil de configurar a través de su plataforma web. Firebase, como otras soluciones BaaS, como "Parse" o "Backendless", nos ofrecerá un sistema de almacenamiento en la nube realtime, servicios de anaítica, sistema de autenticación y notificaciones push entre muchas otras funciones. Además, ofrece un servicio gratuito mientras no se supere un cierto número de llamadas a su API y dado que no se esperaba un alto tráfico para la app en los meses de desarrollo era una solución muy válida. Esto, junto con su amplia documentación y comunidad hicieron que fuera la elección escogida.

Para este proyecto Firebase ofrece dos servicios:

- Por una parte, encontramos el sistema de **autenticación**, el cual nos ofrece los endpoints para poder registrar, identificar y cerrar sesión a un usuario.
- Por otra parte, Firestore es la base datos NoSQL, que básicamente son bases de datos más flexibles con respecto a sus campos, en la que almacenaremos pares clave-valor donde el valor es un JSON. Con este servicio, diseñé e implementé el siguiente modelo de datos:

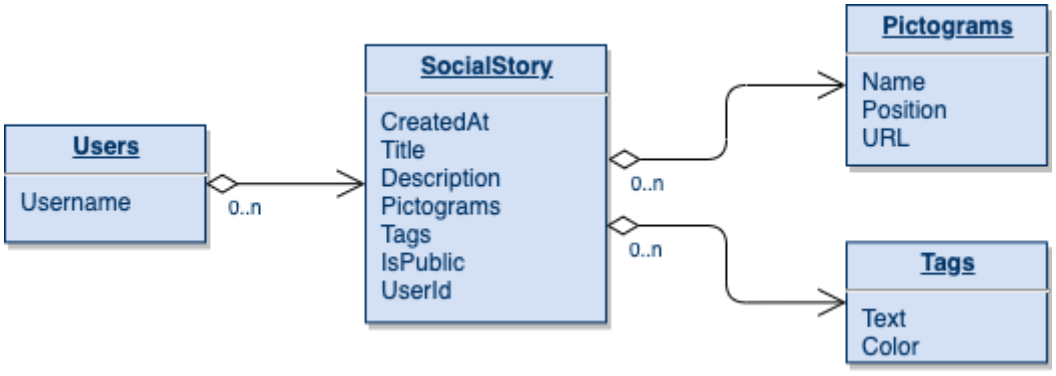


Figura 16: Representación modelo base de datos

En cuanto a la seguridad, Firebase ofrece la posibilidad de configurar reglas para restringir el acceso a sus datos. La configuración que desarrollé es sencilla pero cubre la capa mínima necesaria:

```
service cloud.firestore {
  match /databases/{database}/documents {
    match /socialStories/{story=**} {
      allow read: if request.auth.uid == resource.data.userId || resource.data.isPublic
      allow write: if request.auth.uid != null
      allow delete: if request.auth.uid == resource.data.userId
    }
    match /users/{userId} {
    allow create
    allow read: if request.auth.uid != null
    allow write: if request.auth.uid == userId
    }
  }
}
```

Figura 17: Captura de las reglas habilitadas en Firebase

Con estas reglas configuramos unas condiciones para la tabla de **users** y otras para la tabla de **socialStories**:

- Para *users* se permite la creación para cualquiera, lectura para cualquier usuario que esté autenticado y escritura tan solo para el identificador vinculado a ese usuario.
- Para *socialStories* se permite la lectura de los creadores de dichas historias y de todas las que estén publicadas con el flag *isPublic* activado. La escritura para cualquier usuario autenticado en el sistema, y la eliminación tan solo si el usuario es el que ha creado la historia social.

3.6. Configuración del Service Worker y App Manifest

Como hemos visto en apartados anteriores, con la aplicación web finalizada, solo es necesario configurar el App Manifest y el Service Worker para disfrutar de las prestaciones de la Aplicación Web Progressive.

3.6.1. App Manifest

Para el App Manifest se configuraron las propiedades requeridas: el nombre de la app que aparecerá al instalarse, el modo de pantalla, el logo en varias proporciones (128, 144, 152, 192, 256 y 512 píxeles), el tema de la aplicación, etc. Como resultado se obtuvo el siguiente diseño:

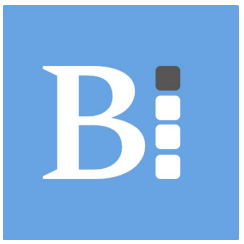


Figura 18: Logo de Bilingualy

El JSON del App Manifest contiene las siguientes características:

```
{
  "name": "Bilingualy",
  "short_name": "Bilingualy",
  "description": "Bilingualy PWA",
  "lang": "en-US",
  "start_url": "/",
  "display": "standalone",
  "background_color": "#68a3e3",
  "theme_color": "#468edd",
  "icons": [
    {
      "src": "/static/icons/128x128.png",
      "sizes": "128x128",
      "type": "image/png"
    },
    {
      "src": "/static/icons/144x144.png",
      "sizes": "144x144",
      "type": "image/png"
    },
    {
      "src": "/static/icons/152x152.png",
      "sizes": "152x152",
      "type": "image/png"
    },
    {
      "src": "/static/icons/192x192.png",
      "sizes": "192x192",
      "type": "image/png"
    },
    {
      "src": "/static/icons/256x256.png",
      "sizes": "256x256",
      "type": "image/png"
    },
    {
      "src": "/static/icons/512x512.png",
      "sizes": "512x512",
      "type": "image/png"
    }
  ]
}
```

Cuando la aplicación capta el manifiesto y lo parsea, podemos ver en la aplicación web cómo la interpretación ha salido exitosa:

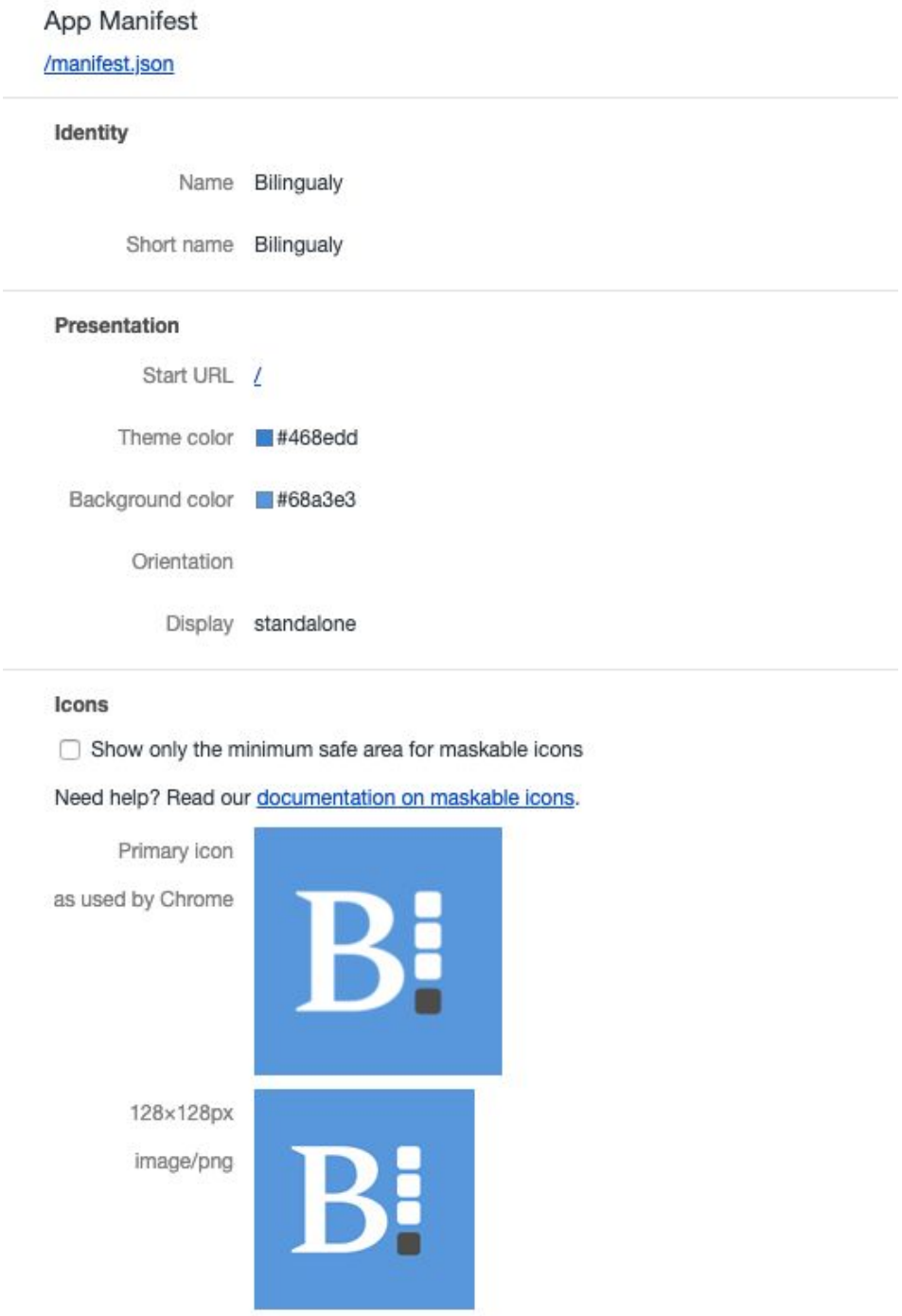


Figura 19: Representación de App Manifest de Bilingualy parseado en el navegador

3.6.2. Service Worker

La configuración del service worker he intentado que sea coherente con lo que la app precisaba. Mostraré las distintas funcionalidades del código y las explicaré seguidamente:

```
registerRoute(
  /\.(?:js|css|scss)$/,
  new NetworkFirst({
    cacheName: "static-resources",
    plugins: [
      new ExpirationPlugin({
        maxEntries: 60,
        maxAgeSeconds: 20 * 24 * 60 * 60, // 20 Days
      }),
    ],
  })
);
```

Esta primera parte se encarga de cachear todos los archivos Javascript, CSS y SCSS que solicite el navegador para montar el DOM. De esta forma, almacenaremos la estructura de la app en el caché por un tiempo máximo de 20 días con la estrategia NetworkFirst. Recordemos que esta estrategia solicita primero a la red y si falla tirará de caché. Se ha elegido esta estrategia para que el usuario siempre disponga de la versión más actualizada posible, si su conexión a Internet lo permite, de lo contrario utilizaría una versión cacheada bastante reciente.

```
registerRoute(
  /\.(?:png|jpg|jpeg|svg|gif)$/,
  new CacheFirst({
    cacheName: "image-cache",
    plugins: [
      new ExpirationPlugin({
        // Cache only 20 images.
        maxEntries: 20,
        // Cache for a maximum of a week.
        maxAgeSeconds: 7 * 24 * 60 * 60,
      }),
    ],
  })
);
```

El siguiente bloque cacheará los formatos png, jpg, jpeg, svg y gif que encuentre en el bundler de la app, es decir los recursos estáticos de la aplicación, que no se deben

confundir con los pictogramas que solicitarán los usuarios. Estos recursos se cachearán con una estrategia de CacheFirst, primero consulta el caché y si no obtiene resultado va a la red, y almacenará hasta 20 imágenes por un máximo de una semana.

```
registerRoute(
  /https:\/\/api\.arasaac\.org\/api\/pictograms\/\d+/,
  new CacheFirst({
    cacheName: "pictogram-cache",
    plugins: [
      new ExpirationPlugin({
        maxEntries: 200,
        maxAgeSeconds: 60 * 60, // 60 minutes
      }),
      new CacheableResponsePlugin({
        statuses: [0, 200],
      }),
    ],
  })
);
```

Aquí empieza el bloque de la búsqueda de pictogramas. Esta entrada se encarga de almacenar todas las respuestas de la API de ARASAAC que nos devuelven una lista de URLs con los pictogramas que se asocian a la palabra clave que le hemos enviado como parámetro. Se utiliza una estrategia de CacheFirst ya que estas bases de datos ya están populadas y por lo general no suelen variar demasiado. Almacenará hasta 200 búsquedas, siempre que el código de respuesta haya sido 0 (opaco) o 200, s por un máximo de 60 minutos.

```
registerRoute(
  new
RegExp("https://api\\.arasaac\\.org/api/pictograms/(es||en)/search/*")
,
  new CacheFirst({
    cacheName: "search-cache",
    plugins: [
      new ExpirationPlugin({
        maxEntries: 100,
        maxAgeSeconds: 60 * 60, // 60 minutes
      }),
      new CacheableResponsePlugin({
        statuses: [0, 200],
      }),
    ],
  })
);
```

Esta sección está totalmente vinculada a la anterior puesto que las peticiones que captará se harán a partir de los resultados de la anterior estrategia. En ésta repetimos CacheFirst, las respuesta con código 0 o 200 y hasta un máximo de una hora, ya que, como hemos dicho previamente, las imágenes no suelen modificar su origen. El número de entradas máximo se ha reducido a 100, puesto que ahora almacenaremos imágenes que pesan muchísimo más que simples strings y los dispositivos móviles no pueden ofrecer tanto espacio libre, por lo que debe controlarse el peso máximo que podemos alcanzar. En Bilingualy con todo el caché lleno rondaría los 1,2 GB. En el siguiente esquema podemos ver el flujo de la información en las dos casuísticas que se pueden dar cuándo un usuario envía una petición y hay conexión a Internet y cuándo no, y cómo se comportará Bilingualy con la configuración que hemos definido para Workbox:

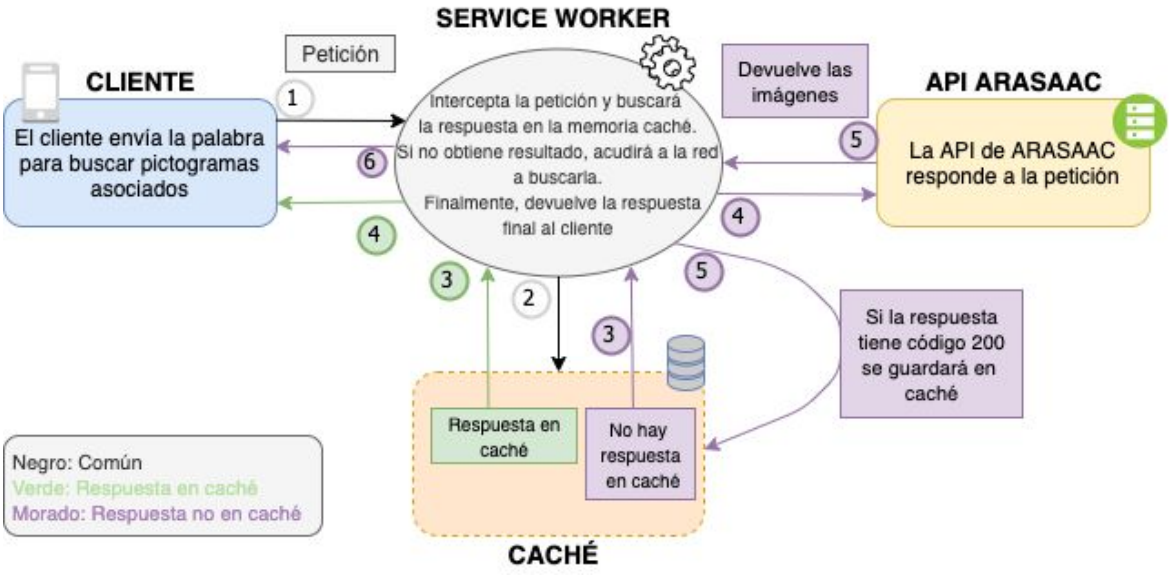


Figura 20: Esquema del funcionamiento de service worker de Bilingualy

```
const bgSyncPlugin = new BackgroundSyncPlugin("postedStoriesQueue", {
  maxRetentionTime: 24 * 60 * 7// Retry for max of one week
});
registerRoute(

/https:\/\firestore\.googleapis\.com\/google\.firestore\.v1\.Firestore
\/Write\/*/,
  new NetworkOnly({
    plugins: [bgSyncPlugin],
  }),
  "POST"
);
```

Finalmente, se ha configurado el Background Synchronization. Con esta funcionalidad, el service worker capta todas peticiones de tipo POST que vayan dirigidas al Firestore,

como son la creación o la eliminación de historias sociales. Se almacenan hasta un máximo de una semana, puesto que esta información es un simple JSON que ocupa muy poco espacio y podemos garantizar que la actividad online que ha realizado el usuario se registrará en la base de datos cuando vuelva a tener conexión a Internet. De la misma forma que en ejemplo anterior, a continuación se ha añadido un diagrama que explica las casuísticas cuando un usuario intenta guardar en el servidor una historia socia. En consecuencia, se evalúa cuándo tiene conexión a Internet o cuando no, y cómo el background sync interviene en este proceso:

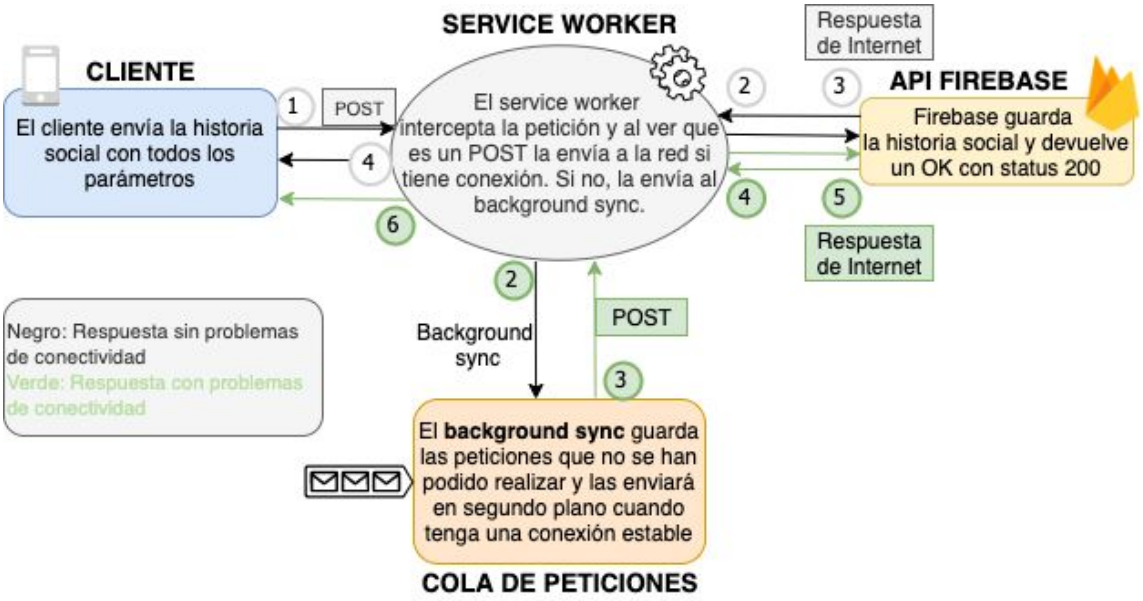


Figura 21: Esquema del funcionamiento del servicio de background sync de Bilingualy

Con esta configuración, una vez se ha instalado y activado el service worker, podemos ver como Workbox se ha encargado de generar las respectivos memorias cachés:

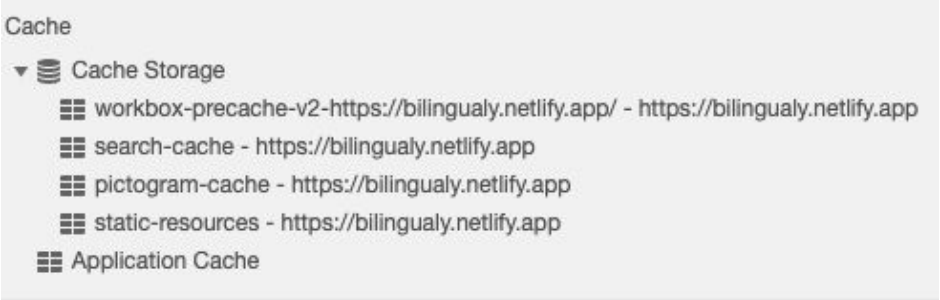


Figura 22: Captura de las cachés generadas por el service worker de Bilingualy

4. Resultados

Tras el desarrollo tecnológico se consiguió cerrar una versión estable y pública de Bilingualy. Como hemos visto en la sección anterior, esta aplicación web ha adoptado los requerimientos necesarios para ser considerada una Aplicación Web Progresiva, así que el objetivo principal de la tesis se ha cumplido satisfactoriamente. Existe una herramienta llamada *Lighthouse* que realiza una auditoría para ejecutar una serie de pruebas y evaluar así la calidad de una aplicación web. Lighthouse tiene un gran enfoque sobre las funciones de las PWA, como la instalabilidad y el soporte sin conexión y presenta una lista de validaciones que una app debe cumplir para que sea considerada PWA. Su objetivo principal es ofrecer una auditoría de extremo a extremo de todos los aspectos del rendimiento de la web y generar un informe con los resultados de las pruebas y los indicadores que podemos utilizar para mejorar estos aspectos. A continuación adjunto el informe hecho para Bilingualy en https://bilingualy.netlify.app/:

Figura 23: Resultados del informe de Lighthouse sobre Bilingualy

Como podemos observar, los resultados negativos son los campos de *Performance* y *Accessibility*. Por un lado, la Performance refleja la velocidad de carga de la página en su primera visita, cuando el service worker aún no sirve el contenido. Esta velocidad se ve muy influenciada por el servicio de Netlify, que al ser gratuito no ofrece sus mejores prestaciones, como podría ser el uso de CDN. Además, se debe tener en cuenta que el bundler configurado por Framework7 no es tan óptimo como el de CRA, el cual hace que la compresión no esté quizás tan bien planteada. Además, la arquitectura base de los componentes de los que nos obliga a partir Framework7 pueden haber generado lentitud en la renderización. Si a ello le añadimos mi escasa experiencia con React, seguro que la arquitectura era muy mejorable en lo que a arquitectura y optimización del código se refiere.

Por otro lado, la Accessibility nos habla de la adaptación de la web para recibir usuarios que pueden estar fuera del rango del usuario "típico", que pueden acceder o interactuar con elementos de manera diferente a lo que el desarrollador espera. Estos errores son más vinculados a mi falta de experiencia como desarrollador web, ya que se trata de problemas relacionados a buenas prácticas con HTML y la definición del “index.html”, elección de colores para mejorar el contraste con el fin de que el texto se lea fácilmente y más problemas de este estilo.

Por el lado positivo encontramos las *Best Practices,* el *SEO* y la validación de PWA. Las buenas prácticas validan algunos aspectos generales como que las imágenes se

muestren de una forma correcta, que la página sea rápida a través de HTTP/2, que sea segura con HTTPS, que se utilizan APIs y librerías no obsoletas, etc.

El SEO garantiza que la página está optimizada para aparecer en los rankings altos del motor de búsqueda. Existen muchos factores que regulan esta búsqueda pero aquí se validan algunos que son fácilmente mejorables. En nuestro caso los resultados muy buenos.

Finalmente tenemos la verificación de PWA que se divide en tres partes:

- La primera comprueba la velocidad y fiabilidad. En este campo solo falta una validación y es que al servir la página la primera vez tarda más de diez segundos en cargarse, como ya ha resaltado el apartado de Performance.
- El segundo verifica la instalabilidad, si se sirve con HTTPS, si el service worker actúa desde la url donde se precisa y si el App manifest cumple los requerimientos.
- El tercero simplemente válida si la PWA está optimizada. Lo único que falta es que la web no renderiza contenido cuando el navegador no acepta Javascript. Por lo demás, está preparada para servirse en dispositivos de Apple, redirigir el tráfico HTTP a HTTPS, el contenido se adapta al tamaño del viewport...

He considerado relevante hablar de esta auditoría en este apartado porque considero que reflejan realmente bien los resultados. Además la he tomado como una crítica constructiva hacia mí como desarrollador. Es decir, por una lado encontramos la tecnología PWA, la cual se ha investigado profundamente y aplicado con éxito en el proyecto. En cambio, por el otro lado encontramos React en el desarrollo más puro web, el cual también se ha completado pero podría mejorarse. Creo que es importante destacar que programar React y el desarrollo frontend mejora a medida que cubres nuevas necesidades en proyectos distinto. En este proyecto se partió cero, esto significa que el proyecto ha sufrido constantes refactorizaciones debido a que cuando descubría una forma mejor de realizar u organizar una tarea la aplicaba y realmente he disfrutado de aprenderlo, lo cual también me impulsaba a intentar hacerlo un poquito mejor cada vez. Esta constante evolución se ve reflejada en una aplicación limpia y ordenada, pero, como indica el informe, todavía existen muchos factores que podrían mejorar y que sin duda mejorarán progresivamente mientras sigo aprendiendo desarrollo web, que pretendo no dejar en los próximos años.

No ha sido posible llegar a conectar la cámara ni las push notifications. Con la cámara faltaría adaptar el sistema para subir imágenes a Firebase y luego que la aplicación sea capaz de recuperar los pictogramas por una parte y las imágenes por otra para montar la historia social. En cambio, con las push notifications se debería configurar también la parte de Firebase que generaría estas notificaciones para enviarlas a los clientes, y además, configurar un Push Manager para escuchar dichas notificaciones y actuar al recibirlas. Firebase ofrece las herramientas para poder realizar tanto el almacenaje de imágenes como las notificaciones, aún así su implementación no es trivial y al ser un backend cerrado debemos adaptarnos a sus sistema, por lo que finalmente, no ha habido tiempo suficiente para desarrollarlas.

En el Anexo 6 podemos encontrar capturas de la interfaz de usuario de Bilingualy en su versión final.

5. Presupuesto

La mayoría de tecnologías utilizadas en este proyecto son de código abierto (React, Firebase, Netlify… así que no se ha precisado de un presupuesto inicial para estas. El ordenador utilizado para todas las tareas de este proyecto ha sido un MacBook Pro comprado en 2018, hace año y media aproximadamente, aplicando una amortización lineal del 20% obtenemos un precio actual de 1820-(1820*0,2*1,5)=1274€. Por otra parte, lo que más eleva el coste es la contratación de un Ingeniero Junior de Software. En este proyecto se ha trabajado 30 semanas con una media de 20 horas semanales son 600 horas en total. Un sueldo para un ingeniero casi licenciado justo sería de unos 12€/hora, que con la contribución de la seguridad social (30%) estaríamos hablando de 15,6€/hora. Por lo tanto el salario saldría por 15,6*600=9360€. Finalmente, se realizó el curso 'Modern React with Redux' en la plataforma online de Udemy para aprender los principios de React y Redux, valorado en 199€.

CONCEPTO	PRECIO
Macbook Pro 2018	1274€
Salario de ingeniero junior	9360€
Curso online Udemy	199€
TOTAL	**10833€**

Tabla 11. Presupuesto.

En cuanto a la viabilidad financiera para el futuro, los dos servicios que podrían modificar su coste y dejar de ser gratuitos podrían ser Firebase y Netlify. Por la parte de Netlify con el plan actual tenemos hasta 300 minutos mensuales de compilación del código para desplegarlo en el dominio online y un ancho de banda de 100 GB mensuales. En el momento que se requieran más minutos para compilar o más ancho de banda tendríamos que pasar al plan Profesional que costaría 45€/mes. Por la parte de Firebase, con el plan gratuito tenemos autenticación ilimitada y la base de datos Firestore permite 1GB de datos almacenados y 10GB de bajada mensuales. Además ofrece hasta un máximo de operaciones de diarias que nos llega de sobra para el uso actual de la app. Si quisiéramos aumentarlos para una versión de producción con más usuarios, se cobraría 0,18$ por GB de dato almacenado y la salida de red se tendría que gestionar con Google Cloud Platform que ofrece un abanico de precios más personalizado en función del número de operaciones diarias de escritura, lectura y eliminación de documentos. Una aproximación sería unos 0,18$ por cada 100.000 operaciones.

6. Conclusiones y líneas futuras

En esta tesis han coexistido dos claras vertientes tecnológicas del mundo Web: las PWA y React. Por un lado, las aplicaciones web progresivas han representado la mayoría de la investigación, mientras que, por otro lado, React ha ocupado la mayor parte el desarrollo práctico. Para hacernos una idea, cuando acabé de investigar el funcionamiento de los service workers y Workbox me tomó solo el 10% del tiempo de desarrollo aplicar esta tecnología al proyecto. El otro 90% fue React, Redux, Firebase, Netlify, conectar con la API de ARASAAC, etc. He de confesar que desde mi nula experiencia como desarrollador web había estimado realmente mal el tiempo que me ocuparía dicho desarrollado, sin embargo, con la extensión de la tesis se consiguieron los objetivos principales.

Respecto a la tecnología PWA, personalmente tengo claro que el mundo tiende al móvil cada vez más: el Mobile First es una tendencia creciente, sobre todo desde que Google anunció que iba a actualizar su algoritmo de búsqueda con el Mobile First Index, el cual sitúa por delante en la indexación las páginas webs cuya versión móvil y experiencia de usuario que obtenga en esta sea mejor. Esto también forzó a Apple a tener que aceptar la tecnología PWA aunque para ellos representa un claro inconveniente, puesto que arrebata mercado a su App Store. Una PWA es accesible, adaptable, instalable, actualizable, segura y ligera, y con estas cualidades estamos obligados a tenerlas en cuenta para todos nuestros nuevos desarrollos futuros.

En mi opinión, las PWA serán la piedra angular en un cambio de paradigma dentro las tecnologías frontend y el mercado tendrá que adaptarse a ellas tarde o temprano. Falta por ver hasta dónde llegarán, puesto que su progresividad en los navegadores no conoce límites e incluso me aventuraria a decir que realmente podrían consolidar en un futuro un digno rival a las aplicaciones nativas, dado que sus prestaciones podrían llegar a cubrir todas su funciones y otras más.

Por lo que respecta a React, he tenido la oportunidad de comprarlo de primera mano con Angular, ya que actualmente trabajo como ingeniero de frontend en una startup donde es una de las tecnologías que utilizamos. Desde mi punto de vista, ambos tienen pros y contras pero, personalmente prefiero React. El único inconveniente es que al ser una librería se deben conocer muchos conceptos más para desarrollar una aplicación sólida, mientras que con Angular y otros framework digamos que puedes obviarlos. Aún así, el conocimiento es poder y considero que conocer las piezas que se necesitan puede aportar rendimiento y personalidad al proyecto, lo cual abre un mundo de posibilidades que frameworks, como Angular, cierran de antemano. Por otro lado, creo que es mucho más intuitivo a la larga crear componentes con JSX y utilizar Hooks para gestionar la lógica que crearlos con Angular y tener que aprender sus métodos para inyectar Javascript en el HTML. Con Angular, el código queda mucho más verboso e incluso menos entendible. Mi experiencia me dice que mejorar con React implica mejorar con Javascript, en cambio, mejorar con Angular no es tan extrapolable a otras tecnologías. En resumen, creo que React tiene mucho futuro por delante y estará en el podio del frontend por muchos años.

Por lo que respecta a la parte funcional de Bilingualy, evaluándola con mi madre, que es quien me inspiró a realizar esta idea y tiene una amplia visión sobre las necesidades que quería cubrir, consideramos que es una herramienta realmente útil. Mi madre ha

defendido que a ella le hubiera ahorrado mucho trabajo, y que sin duda, facilitará muchas tareas a familias y profesionales que trabajen con comunicación aumentativa y/o comunicación alternativa, por lo que personalmente me produce una enorme satisfacción. He decidido continuar con ella para terminar una versión más completa, con la cámara, push notifications, y otras funciones, y repartirlo a varios logopedas y profesionales que conocemos para que la utilicen. Además, el código es público y estará subido para cualquier persona u organización que quiera utilizarlo. Así como ARASAAC, publica gratuitamente muchísimo material y contenido para ayudar a este sector sin ánimo de lucro, también quiero contribuir y aportar mi granito de arena con Bilingualy y ojalá pueda ayudar a alguien más.

Finalmente, me gustaría comentar las líneas futuras a partir de lo que he aprendido:

- Uno de los puntos más importantes sería cambiar todo el código de Javascript a Typescript. Este superset de Javascript, añade muchas ventajas pero destacaré una: convierte el lenguaje en tipado. Tipar el código añadirá una gran solidez al código y prevendrá muchos errores.
- Vinculado con la solidez del código, debería añadirse tests. Ahora que trabajo en este sector he visto la fragilidad de un código sin testear y la dificultad para localizar los errores. Cuando se realizan test unitarios, de integración, end-to-end, etc. el código consigue una madurez inmediata, y a medida que el proyecto crece se agradece mucho. Algunas de las herramientas más interesantes actualmente y sin duda a ojear para el testing son Jest, Cypress e Storybook.
- A nivel de diseño, la app se implementó siguiendo el patrón de mobile first, pero la vista en web debería adoptar muchos cambios con respecto a la vista desde web, por ejemplo el sidebar dejar de esconderse y aparecer fijo en un lateral, suprimir los botones en los extremos de la pantalla, puesto que en web carecen de sentido en un lugar donde al usuario le costaría de encontrar, etc.
- El proyecto ganaría muchas virtudes si adoptase CreateReactApp cuando lance una nueva versión que soporte una configurable de Workbox. Posteriormente, podría elegirse la librería de componentes que se eligiera Material UI, Semantic UI… Aunque Framework7 como librería solo para componentes creo que es una opción muy recomendable igualmente.
- A nivel de la PWA, como he mencionado previamente, sería muy interesante integrar dos funciones más que no han dado tiempo a implementar pero estaban en el plan inicial: por una lado la cámara, que en este proyecto no ha dado tiempo de implementarse debido a que implicaba investigar cómo integrar el almacenaje de imágenes en Firestore y, posteriormente, adaptar el sistema que monta las historias sociales para recoger las imágenes junto con los pictogramas. Por otro lado las push notifications,, que debería evaluarse también como lanzar estos mensajes desde Firebase y configurar un Push Manager con el service worker. De esta forma habríamos cerrado las herramientas más core que ofrecen este tipo de workers.

Para concluir, quería mencionar que ahora que veo todo el proyecto en perspectiva creo que ha resultado una investigación muy interesante y gratificante, realmente he disfrutado de todo lo que he aprendido, además lo considero muy útil y me ha servido considerablemente en el mundo laboral, al que creo que estaré vinculado con el desarrollo frontend en los próximos años. También considero interesante que otro

proyectista continuase con esta línea de investigación puesto que se ha presentado una tecnología, las PWA, que sin duda estará muy presente en el desarrollo web a partir de ahora. A su favor, esta tecnología sigue en continuo crecimiento y seguro que habrá nuevas funciones y objetivos a investigar. Con la base teórica y práctica que he presentado, considero que puede abrir muchas puertas para que otras nuevas ideas se lleven a cabo.

Bibliografía

[1] S. Richard, P. LePage. "What are Progressive Web Apps" [Online] Available: https://web.dev/what-are-pwas/ [Accessed: 5 Septiembre 2019]

[2] Thinkwik "Why ReactJs is gaining so much popularity these days?" [Online] Available: https://medium.com/@thinkwik/why-reactjs-is-gaining-so-much-popularity-these-days-c3aa686ec0b3 [Accessed: 5 Septiembre 2019]

[3] D. Triguero "Progressive Web Apps (PWA): la nueva generación de aplicaciones móviles" [Online] Available:https://profile.es/blog/progressive-web-apps-la-nueva-generacion-de-aplicaciones/ [Accessed: 5 Septiembre 2019]

[4] M. Haldar "What is a PWA and why should you care?" [Online] Available: https://blog.bitsrc.io/what-is-a-pwa-and-why-should-you-care-388afb6c0bad [Accessed: 6 Septiembre 2019]

[5] J. Archibald "The Service Worker LifeCycle?" [Online] Available: https://developers.google.com/web/fundamentals/primers/service-workers/lifecycle [Accessed: 12 Septiembre 2019]

[6] I. Aderinokun "The Service Worker Lifecycle" [Online] Available: https://bitsofco.de/the-service-worker-lifecycle/ [Accessed: 18 Septiembre 2019]

[7] W3schools "React Lifecycle" [Online] Available: https://www.w3schools.com/react/react_lifecycle.asp [Accessed: 27 Septiembre 2019]

[8] P. Lin "Types of React Components" [Online] Available: https://levelup.gitconnected.com/types-of-react-components-a38ce18e35ab [Accessed: 23 Septiembre 2019]

[9] . Matek "Why Use React JS" [Online] Available: https://railsware.com/blog/why-use-react/ [Accessed: 27 Septiembre 2019]

[10] Vincent Driessen "A successful Git branching model" [Online] Available:https://nvie.com/posts/a-successful-git-branching-model/ [Accessed: 20 Noviembre 2019]

[11] M. Anastasov "CI/CD Pipeline: A Gentle Introduction" [Online] Available:https://semaphoreci.com/blog/cicd-pipeline [Accessed: 22 Diciembre 2019]

[12] Denis Zhbankov "Maintaining a fork of create-react-app as an alternative to ejecting" [Online] Available: https://medium.com/@denis.zhbankov/maintaining-a-fork-of-create-react-app-as-an-alternative-to-ejecting-c555e8eb2b63 [Accessed: 22 Diciembre 2019]

[13] Kevin Hernandez "Using Workbox with Create React App (Without Ejecting)" [Online] Available: https://medium.com/la-creativer%C3%ADa/using-workbox-with-create-react-app-without-ejecting-b02b804854b [Accessed: 22 Diciembre 2019]

[14] J. Manuel Alarcón "Webpack: qué es, para qué sirve y sus ventajas e inconvenientes" [Online] Available: https://www.campusmvp.es/recursos/post/webpack-que-es-para-que-sirve-y-sus-ventajas-e-inconvenientes.aspx [Accessed: 2 Enero 2020]

[15] Amazon Web Services "¿Qué son las bases de datos NoSQL?" [Online] Available: https://aws.amazon.com/es/nosql/ [Accessed: 2 Enero 2020]

[16] Página oficial de React"Presentando los Hooks" [Online] Available: https://es.reactjs.org/docs/hooks-intro.html [Accessed: 8 Enero 2020]

[17] Página oficial de Framework7 "https://framework7.io/react/" [Online] Available: [Accessed: 14 Enero 2020]

Anexos

Anexo 1: Configuración del App Manifest

El app manifest, como hemos comentado, tiene un aspecto similar al siguiente:

```
{
  "short_name": "Weather",
  "name": "Weather: Do I need an umbrella?",
  "description": "Weather forecast information",
  "icons": [
    {
      "src": "/images/icons-192.png",
      "type": "image/png",
      "sizes": "192x192"
    },
    {
      "src": "/images/icons-512.png",
      "type": "image/png",
      "sizes": "512x512"
    }
  ],
  "start_url": "/?source=pwa",
  "background_color": "#3367D6",
  "display": "standalone",
  "scope": "/",
  "theme_color": "#3367D6"
}
```

Como se puede apreciar, presenta una serie de campos. Algunos son obligatorios para que la instalabilidad sea posible y otros son optativos, pero cada uno proporciona una parte de información distinta. El propósito de cada campo es el siguiente:

- **short_name**: Especifica el nombre que la app tomará al instalarse en el dispositivo.

- **name**: Este es el nombre que aparecerá en Chrome con el banner que sugerirá añadir la aplicación a la pantalla de inicio. En caso de que no esté presente se utilizará el campo *short_name*.

- **icons:** Define los iconos disponibles con sus respectivos tamaños para que el navegador los utilice en la pantalla de inicio, el lanzador de aplicación, el task switcher, la vista previa, etc. De este modo, dispone de un set de imágenes para poder presentar la PWA de las misma forma que una app nativa. En función del

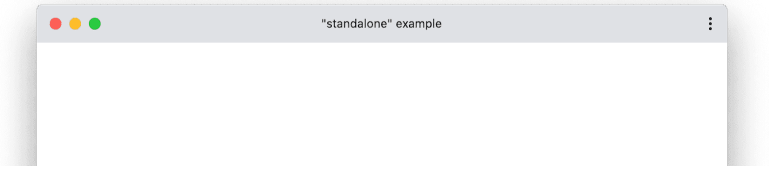
navegador, necesitará que se le proporciona más o menos tamaños o con un par será capaz de autoescalar por él mismo.

- **start_url**: Se trata de la URL donde la app empieza para que una vez instalada la aplicación sepa qué página debe mostrar primero.

- **display**: Sirve para personalizar la vista del interfaz de usuario del navegador. Existen 3 modos:

  - **fullscreen**: Abre la app web sin ningún UI de navegador y utiliza toda el área disponible de la pantalla. Es el modo estándar que se usa para videojuegos.

  - **standalone:** Abre la app web como si fuera un app nativa independiente. La app corre en su propia ventana, separada del navegador y esconde los típicos elementos del navegador como por ejemplo, la barra de búsqueda.



  - **minimal-ui**: Este modo es similar al anterior, pero este adhiere el mínimo grupo de elementos del la interfaz de usuario del browser que controlan la navegación, como por ejemplo el botón de “Volver a la página anterior” o “Recargar”.

  - **browser**: Utiliza la vista usual del navegador, sin modificaciones.

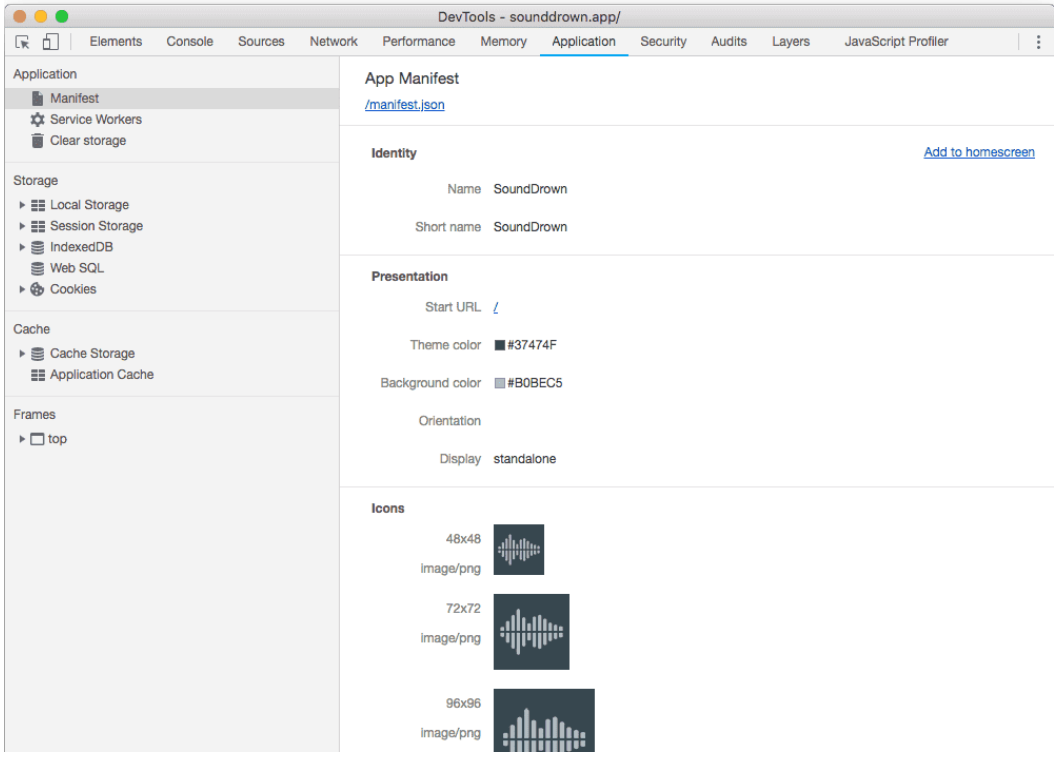
- **scope**: Define el alcance de URLs que el navegador debe considerar parte de la aplicación y se utiliza para definir cuando el usuario sale de aplicación. Es importante que la *start_url* resida dentro del *scope*.
- **theme_color**: Define el color del toolbar del navegador y puede reflejarse en la vista previa.

- **background_color**: Define el color de la pantalla que aparece al lanzar la app en la pantalla de inicio mientras carga el contenido, conocida como “splash screen”.

Si el manifiesto está presente en la raíz del proyecto y debe ser enlazado a su archivo correspondiente. Chrome entre otros navegadores automáticamente activará un banner en la parte inferior de la pantalla para notificar al usuario de que puede instalar la aplicación. Si el usuario acepta, el icono que hemos definido en el manifiesto aparecerá en la pantalla de inicio el teléfono como si de una aplicación nativa se tratase.

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

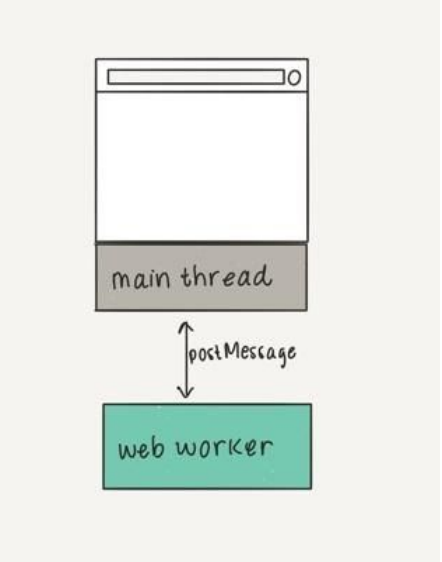
Al seguir todos los pasos correctamente podemos ver cómo aparece la configuración de nuestro manifiesto en las herramientas para desarrolladores de nuestro navegador. Este es un ejemplo de cómo se mostraría en el DevTools de Google Chrome:


DevTools - sounddrown.app/
Elements
Console
Sources
Network
Performance
Memory
Application
Security
Audits
Layers
JavaScript Profiler
Application
Manifest
Service Workers
Clear storage
Storage
Local Storage
Session Storage
IndexedDB
Web SQL
Cookies
Cache
Cache Storage
Application Cache
Frames
top
App Manifest
/manifest.json
Identity
Add to homescreen
Name SoundDrown
Short name SoundDrown
Presentation
Start URL /
Theme color #37474F
Background color #B0BEC5
Orientation
Display standalone
Icons
48x48
image/png
72x72
image/png
96x96
image/png

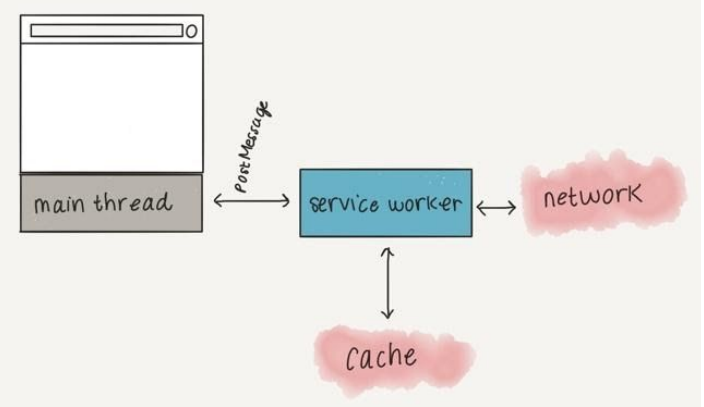
Anexos 2: Javascript Workers

Existen tres tipos de Javascript Workers: Web Workers, Service Workers y Worklets. Los tres comparten características similares en su forma de trabajar pero difieren en su finalidad. Como he explicado en los service workers, un worker es un script que se ejecuta en un hilo distinto del hilo principal del navegador. Si hubiera demasiada actividad en este hilo principal podría ralentizar la página. Para que esto no ocurra se utiliza los workers. A continuación explicaré en qué difiere cada uno de ellos:

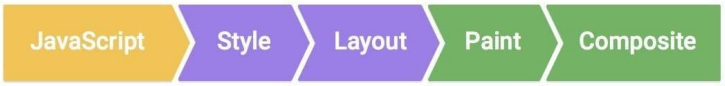
- **Web Workers:** Son los workers que se usan en los casos más generales ya que no tiene casos de uso específico. Su único propósito es descargar del thread principal de cualquier proceso pesado. Se crean con la Web Worker API y deben ser referenciados en el archivo Javascript principal. Al igual, que el resto de workers no tienen acceso al DOM, pero se comunican usando el método "Post Message", el cual les permite intercambiar información entre los distintos procesos.



- **Service Workers:** Son el tipo de workers destinados con el explícito propósito de servir como proxy entre el navegador y la red y/o el cache. Como los demás workers, deben ser referenciados en el archivo de Javascript principal. A diferencia de sus hermanos, esta clase posee las herramientas necesarios para cumplir su propósito como proxy. De este modo, es capaz de interceptar las peticiones de red generadas por el archivo principal y gestionarla, o bien, responderlas adecuadamente ya que tiene agregado el acceso a la memoria caché.



- **Worklets:** Los worklets son los workers más ligeros y específicos. Estos permiten a los desarrolladores desglosar y conectar las etapas con las operaciones más pesadas del proceso de renderización del navegador. De esta forma, los Worklets nos permiten tener acceso a bajo nivel al canal donde se renderizan dichas etapas. Por ejemplo, podríamos utilizar el *Paint Worktlet*, que se encarga de conectar con la etapa de "Paint", para generar un gradiente que calcula su distribución en función de la altura y anchura de la ventana disponible.



Los workers, como hemos visto, permiten liberar carga del hilo principal del navegador y aunque cada uno tiene un propósito distinto, todos comparten la metodología para comunicarse: "Post Message". Con este método y el MessageChannel, los service workers son capaces de hablar con todos los clientes (broadcast), solo con uno (unicast) o incluso con otro worker, y además, transmitir información al DOM y comunicar que puede actualizarse.

```
/* main.js */

// Create worker
const myWorker = new Worker('worker.js');

// Send message to worker
myWorker.postMessage('Hello!');

// Receive message from worker
myWorker.onmessage = function(e) {
  console.log(e.data);
}
```

```
/* worker.js */

// Receive message from main file
self.onmessage = function(e) {
  console.log(e.data);

  // Send message to main file
  self.postMessage(workerResult);
}
```

Anexo 3: Experimento sobre el ciclo de vida de los service workers

Para entender los principios y beneficios de los Service Workers, presentaré un simple escenario donde evaluaré su ciclo de vida.
Vamos a imaginar, como desarrolladores de frontend, que tenemos una carpeta con el usual index.html y un segundo archivo llamado about.html, para embellecer el contenido, añadimos también un archivo “style.css” y finalmente, un archivo Javascript llamado “main.js”. Si la explicación se ha seguido deberíamos tener la siguiente estructura de archivos:

El contenido del HTML y su estilo con CSS lo dejo a libre elección ya que su variación es irrelevante para el experimento. Lo que realmente importa es invocar el archivo “main.js” en los archivos html, ya que si un usuario visita cualquiera de estas páginas queremos que invoque al archivo Javascript para lanzar el service worker. Básicamente, añadiendo la siguiente línea de código ya nos servirá:

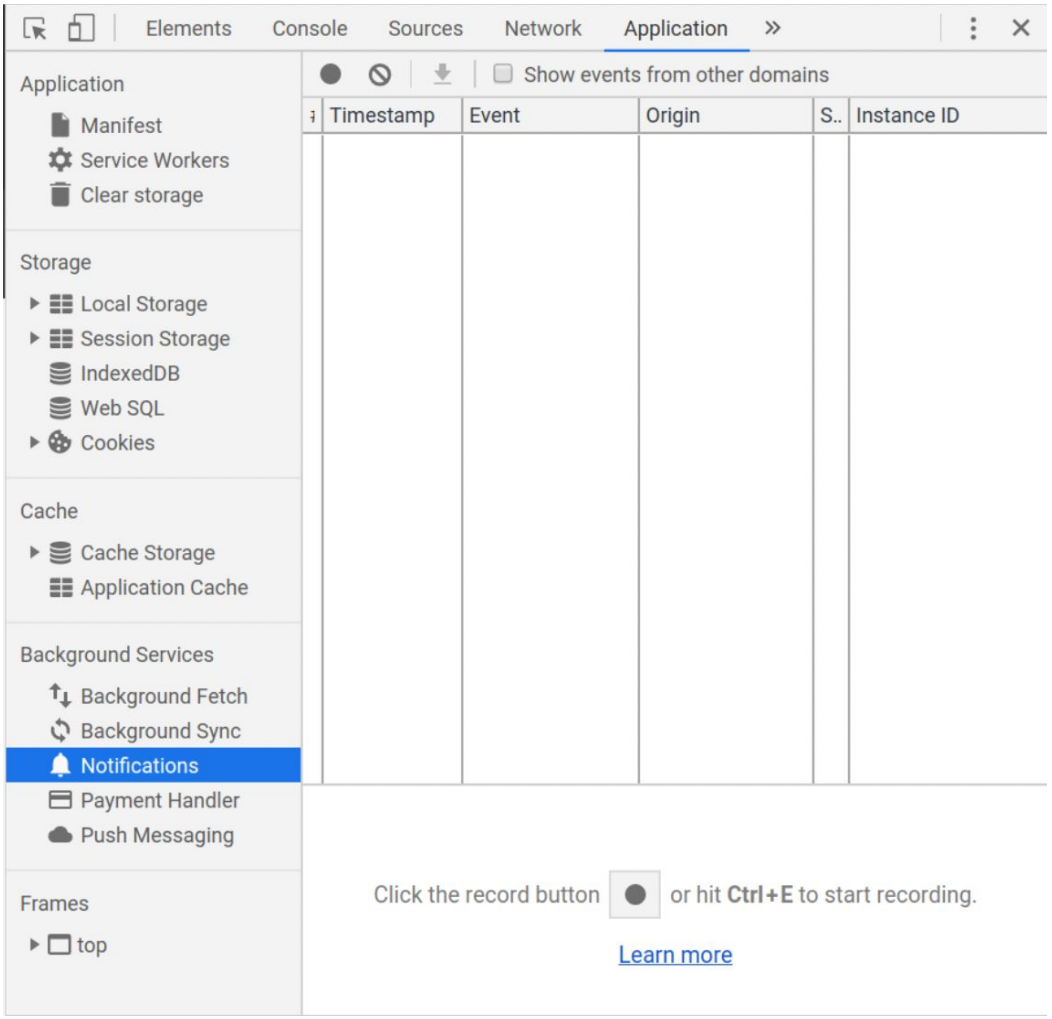
```
<script src="js/main.js"></script>
```

Dado que de momento no vamos con a contar con ningún sistema de routing, tenemos que añadir esta línea de código en cada archivo HTML, de lo contrario el service worker podría no iniciarse en función de la página a la que accedemos primero.
Otro paso importante es adaptar nuestro entorno para que pueda servir con HTTPS el service worker, dado que es un requerimiento obligatorio. Hay muchas formas de hacerlo pero yo he usado una extensión del VS Code llamada “Live Server” el cual puedes encontrar aquí:
https://marketplace.visualstudio.com/items?itemName=ritwickdey.LiveServer

Una vez los archivos web y el entorno están configurados, podemos empezar a analizar el comportamiento de los service workers:

1. Primero de todo, abrir las herramientas de desarrollo del navegador que vayáis a utilizar, yo recomiendo Chrome que es el más adelantado para la tecnología de

PWA. Utilizaremos fundamentalmente dos pestañas: la Consola y la pestaña de Aplicación. La aplicación contiene toda la información referente al service worker y la consola la utilizaremos para reflejar con logs cuando se lanzan los eventos a los que reacciona el service worker.



2. **Registro**: Antes de empezar con el registro tenéis que tener en cuenta que estáis trabajando con PWA y eso implica que no todos los navegadores están preparados para soportar service worker. Por lo tanto, el primer paso consiste en comprobar si el navegador donde se sirve el contenido lo soporta. Si es así. podemos llamar al método “register”. El mínimo código para llevar a cabo este proceso es el siguiente:

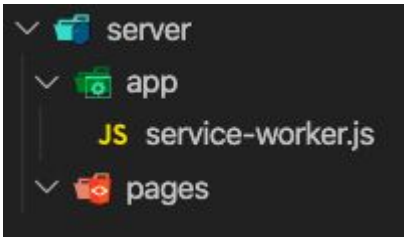
```
if ('serviceWorker' in navigator) {
  navigator.serviceWorker.register('/service-worker.js');
}
```

Este código es el registro básico, a partir del cual podemos encontrar muchas variaciones. Por ejemplo, en el siguiente código muestro un registro más avanzado y quizás adecuado:

```
if ("serviceWorker" in navigator) {
  window.addEventListener("load", () => {
    navigator.serviceWorker
      .register("../sw_cached_pages.js")
      .then(reg => console.log("Service worker registered"))
      .catch(err => console.log(`Service worker registration
failed: ${err}`));
  });
}else{
  console.log("Service worker unavailable");
}
```

Como podemos apreciar, el método “register” recibe como parámetro la ruta donde se encuentra el archivo Javascript con la configuración del service worker. Si no habéis trabajado demasiado con Service Workers os recomiendo localizar este archivo en la raíz del proyecto para que su alcance llegue a todas las rutas (archivos) del proyecto. Por otro lado, el método “register” devuelve una promesa que se resuelva si el registro es satisfactorio, mostrando el log “Service worker registered”. De lo contrario, la promesa se rechazará y el registro fallará, mostrando el log “Service Worker registration failed”.

Mencionando antes el alcance de los service workers: el Service Worker tiene un alcance limitado que define el dominio desde donde se servirán los archivos. Por defecto, dicho alcance empieza a partir de donde se encuentra localizado este archivo. A qué me refiero con el dominio?Simplemente indica la ruta o carpeta origen del proyecto de nuestro sitio web a partir del cual las solicitudes de red serán interceptadas y los archivos serán atendidos. Para entenderlo mejor, si organizamos nuestro proyecto así:



Podemos ver que el service worker está localizado en “server/app/service-worker.js”. Cualquier petición hecha a la URL relativa “/server/app/...” será interceptada por el service worker. Las peticiones que vaya a “/server/pages/…” serán ignoradas por el service worker. Para definir nuestro propio alcance podemos pasar al método “register” la ruta como segundo

parámetro dentro de un objeto que permite varias configuraciones más:

```
if ("serviceWorker" in navigator) {
  window.addEventListener("load", () => {
    navigator.serviceWorker
      .register("../sw_cached_pages.js",{
        scope: '/app/'
       }))
      .then(reg => console.log("Service worker registered"))
      .catch(err => console.log(`Service worker registration
failed: ${err}`));
  });
}else{
  console.log("Service worker unavailable");
}
```

Como he dicho antes, los service workers son “event-driven”, por lo que trabajaremos bastante con eventos. En este último código hemos visto el primer ejemplo: primero comprueba que la API de service worker esté disponible en el navegador. Si lo está, el service worker que está en “../sw_cached_pages.js” se registrará una vez la página cargue y lancé el evento “load”. Por que? Bien, si nos fijamos en el punto de vista de la experiencia de usuario nos daremos cuenta que la primera vez que visitamos la web el service worker no funciona aún y el navegador no tiene ya ninguna forma de saber de antemano si un service worker será finalmente instalado.

Otra pregunta que aparece es qué momento es el adecuado para lanzar el registro del service worker. Desde el punto de vista de un desarrollador, su prioridad debería ser asegurar que el navegador rápidamente carga el mínimo conjunto de recursos críticos para mostrar la página funcional. Cualquier elemento que ralentice la recuperación de los componentes clave es enemiga de una rápida experiencia de carga.

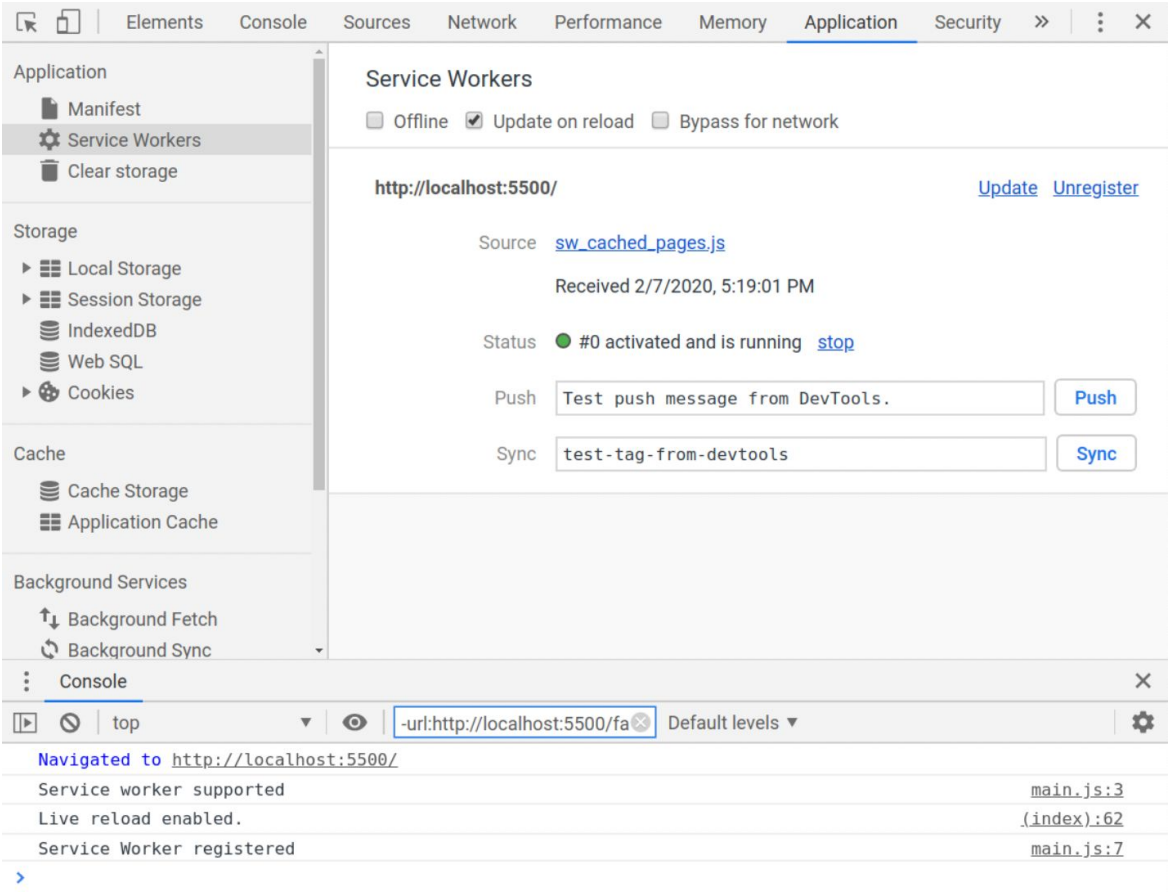
Ahora imagina que estamos utilizando cualquier tipo de teléfono móvil común de baja potencia y en el momento de abrir una web, se inicia un proceso que descarga el JavaScript o las imágenes que la página necesita para renderizar y de repente, el navegador decide iniciar otro hilo en segundo plano. Este subproceso adicional agrega contención al tiempo de CPU y a la memoria que el navegador podría gastar en la presentación de una página web interactiva. Imaginemos ahora el peor escenario, ¿qué pasa si este hilo comienza a descargar recursos de la red? Aparece otro problema grave cuando reducimos aún más el limitado ancho de banda móvil si descarga recursos secundarios al mismo tiempo. Lo que pretendo concienciar es que lanzar el service worker agrega un nuevo subproceso que intente descargar y almacenar en caché los

recursos en segundo plano puede jugar en contra del objetivo principal, que es proporcionar una experiencia de tiempo de interacción más rápida y ligera la primera vez que un usuario visita su sitio web. Una posible solución es controlar el inicio del service worker seleccionando cuándo llamar al registro. Para eso, podemos retrasar el registro hasta después de que el evento “load”, que indica la carga de la ventana, como podemos comprobar en el ejemplo anterior. Esta no es la solución única ni el mejor lugar para hacer siempre el registro, existen muchos escenarios diferentes en los que se tenga que evaluar cómo es más adecuado ejecutar este hilo.
Este proceso ya no tiene impacto en las siguientes visitas al sitio web ya que service worker estará ya operativo y es irrelevante dónde se llame al registro pues una vez instalado y activado, probablemente, no se lanzará otra vez a menos que cambie la ubicación o la configuración del service worker. Una situación perfecta para considerar un registro temprano sería cuando un service worker utiliza la función “clients.claim()”, la cual es utilizada para forzar que el cliente tome el control de la página en la primera visita. Ciertamente, en ese escenario tiene sentido activar el service worker tan pronto como sea posible e intentar popular el caché en tiempo de ejecución. En este caso, deberíamos comprobar los recursos solicitados en la instalación teniendo en cuenta el ancho de banda como he explicado anteriormente.

Para poder simular la primera visita a la web, os recomiendo usar la pestaña de incógnito de Google Chrome mientras visualizas lo que está ocurriendo en la ventana del tráfico del kit de herramientas para desarrolladores del navegador. Además, se pueden usar herramientas de depuración remotas para conectar el móvil al ordenador vía USB y testear una red lenta simulada en un móvil real.
En resumen, como buen desarrollador os animo a priorizar tener la mejor experiencia en una primera visita a tu web. Investigando donde se debería situar el registro del service worker es una gran elección que se debe premeditar pero para situaciones simples suele valer con retrasar el registro hasta se lance el evento de “load”.

Una vez explicado el registro, podemos seguir con el ejemplo. Vamos abrir la herramienta “Live Server”. Normalmente servirá nuestro proyecto en el puerto 5500 por defecto, así que podemos abrir el navegador con el set de herramientas para empezar a evaluarlo. Utilizaremos el registro que espera al evento “load”. Recordad crear el archivo “sw_cached_pages.js” en la ruta correcta para que pueda ser registrado.
Como veréis en el ejemplo de abajo, en este punto, si vamos a la consola veremos que han aparecido varios logs incluyendo “Service worker supported” y “Service worker registered”. Bien, está soportado, pero iremos más allá. Podemos verificar más información sobre nuestro service worker en la pestaña de Aplicación. Allí encontraremos una sección llamada Service Worker, como veréis en la imagen de abajo. Allí, encontrarás la configuración del service worker donde indica el archivo del que está sacando su configuración (“sw_cached_pages.js”) o

el estados del worker, del cual hablaremos en los siguientes pasos. Podréis ver que el estado es “Activated and running”. Nos informa que está operativo y que ha sido capaz de registrarse, instalarse y activarse.
Otro punto importante es activar el checkbox de “Update on reload” para actualizar el service worker cada vez que recarguemos la pággina, y tambiñen, activar el checkbox de “Preserve log” en la consola para que no perdamos los logs al recargar y a su vez podamos ver los logs del hilo en segundo plano.



3. **Instalación:** Antes de empezar con una explicación más compleja, me gustaría añadir un par de líneas de código en el “sw_cached_pages.js” para verificar que el ciclo de vida del worker se ejecuta como esperamos.

```
//Call install event
self.addEventListener("install", event => {
      console.log("Service Worker installed");
});
//Call activate event
self.addEventListener("activate", event => {
      console.log("Service Worker activated");
});
```

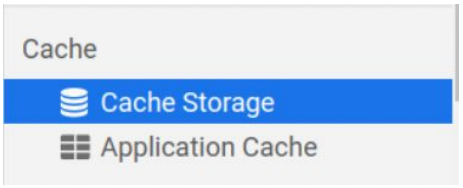
Al guardar esta configuración del archivo del service worker, el navegador

detectará una diferencia a nivel de byte del service worker y procederá a actualizar su service worker, pero esta vez veremos los cuando se lanzan los eventos de “install” y “activate” gracias a los logs que aparecerán en la consola:

```
Navigated to http://localhost:5500/
Live reload enabled.                                    (index):62
Service Worker installed                                sw_cached_pages.js:13
Service Worker registered                               main.js:6
Service Worker activated                                sw_cached_pages.js:18
>
```

De un primer vistazo, podemos pensar que los logs se han mezclado pero no es así. Debemos tener en cuenta que el registro se lanza en un hilo distinto al que correrá el service worker y son totalmente asíncronos, así que el orden de los logs puede cambiar en diferentes intentos pero siempre aparecerá antes el de instalación que el de activación porque estos dos sí se ejecutan en el mismo.
Aún habiendo cambiado la configuración del service worker, como teníamos activado el “Update on Reload” al recargar la página se iba a montar igualmente de nuevo el worker como si fuera una nueva versión.

Ahora, vamos a hablar de la etapa de instalación. Una vez el registro del service worker se ha completado y ha tenido éxito, se lanza automáticamente el evento de “install” tan pronto como sea posible y solo lo hace una vez. En esta etapa, cualquier service worker previamente instalado estará al cargo y el nuevo no interceptará aún ninguna petición.
Como he dicho antes, si alteramos el script del service worker el navegador considerará que es uno distinto y intentará registrarlo e instalarlo. El evento “install” es la oportunidad perfecta para cachear todo lo que necesitemos antes de tomar el control de los clientes, es decir, las páginas a las que serviremos.
Gracias a la promesa que se le pasa al método “waitUntil()”, proporcionado por el evento, el navegador sabe cuando la instalación se da por completa y si ha salido bien. De lo contrario, si la promesa se rechaza, en consecuencia la instalación fallará y el navegador expulsará el service worker, enviándolo a la etapa de “redundant” donde nunca controlará los clientes.
Aquí aparece otro personaje importante, el caché. Podemos encontrar su sección en el kit de desarrolladores y allí encontraremos el “Cache Storage”, el cual prestaremos especialmente atención a partir de ahora porque es donde almacenaremos todo el contenido que queramos servir cuando estemos offline o cuando el worker lo requiera.



Vamos a indagar en esta memoria cache. A continuación, he pegado un bloque de código con el ejemplo más básico para entender la actuación del caché.

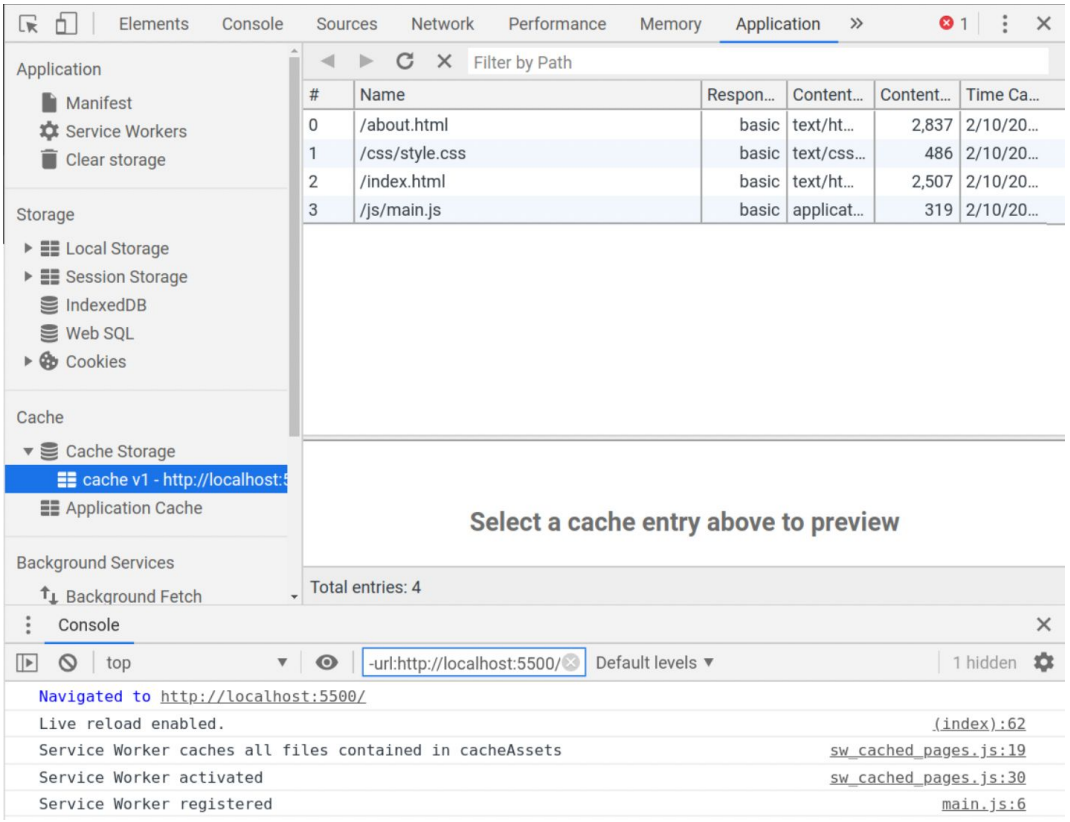
```
const cacheName = "cache v1";
const cacheAssets = [
  "index.html",
  "about.html",
  "/css/style.css",
  "/js/main.js"
];

// Call Install Event
self.addEventListener("install", event => {
  //Tells the browser to wait until the promise is finished
  event.waitUntil(
    //Use Cache Storage API: "caches"
    caches.open(cacheName)
    .then(cache => {
      console.log(
        "Service Worker caches all files contained in
cacheAssets"
      );
      cache.addAll(cacheAssets);
    })
    //At this point we can stop waiting
    .then(() => self.skipWaiting())
  );
});
```

Dentro del “event listener”, que reaccionará al lanzarse el evento “install”, podemos encontrar el *waitUntil()* la cual comunica al navegador que espere hasta que la promesa o promesas que se le pasa como argumento finalice. Esta promesa que vemos en el código utiliza la API de Cache Storage para crear el caché llamado como le hemos indicado en la declaración del cacheName. Esto devolverá un objeto caché con el que llamaremos a su método addAll() y pasaremos una lista de argumentos. Cuando esta promesa se resuelva, llamaremos al metodo *self.skipWaiting()* para indicar que la instalación ha terminado y puede proceder con la activación.

Al guardar el archivo podréis ver que en la consola aparece el log “Service Worker cachés all files container in cacheAssets” y además, el panel de “Cache Storage” será ahora desplegable. Si lo abrimos veremos una sección llamada “cache v1” y al clicar encontraremos todos los recursos que le habíamos indicado que guardase. Abajo dejo una imagen que muestra el contenido cacheado.

Si todos los archivos se almacenan en caché correctamente, se instalará el service worker. En cambio, si alguno de los archivos no se descarga, la instalación fallará. Esto nos permite contar con tener todos los recursos que

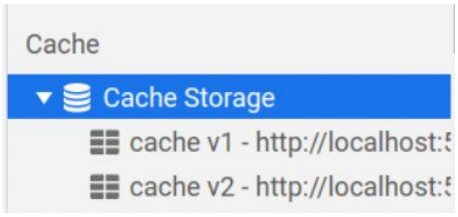
definimos en la instalación si está funciona, pero también implica que debemos tener cuidado con la lista de archivos ya que si definimos una lista larga aumentará la posibilidad de que un archivo no pueda almacenarse en caché por algún error o por simplemente por capacidad, lo que hará que el worker no se instale.



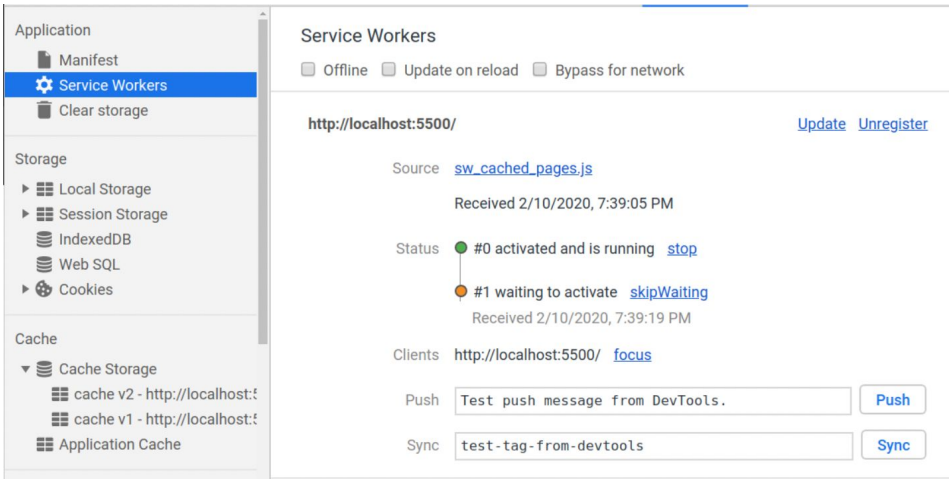
Si intentamos navegar offline ahora mismo, todavía no se servirá ningún contenido y aparecerá el típico juego del dinosaurio de Google para matar el aburrimiento. Esto es debido a que el service worker aún no ha sido activado, y por lo tanto, aún no estamos interceptando las peticiones para gestionarlas. No estamos todavía en la capa de red.

Una vez activado, puedes realizar la prueba offline para asegurarte de que está cachando adecuadamente. Para ello debes activar la opción de “Offline” en la parte superior de la pestaña del Service Worker y recargar la página. Si intentas navegar del index.html al “about.html” verás como todo funciona correctamente. ¡Hemos conseguido un gran objetivo! Este era uno de los grandes pasos que llevábamos persiguiendo para acercar las web apps a la aplicaciones nativas.

¿Qué crees que ocurriría si cambiamos el nombre del cache en el archivo de configuración del service worker a “cache v2”? Si lo hacemos y recargamos la página nos daremos cuenta que ambas caché coexisten en el “Cache Storage”, así que aunque no utilicemos la versión 1 más, todavía sigue almacenada allí.



Este es un escenario típico que se afronta cada vez que actualicemos el service worker. Cuando lo actualizamos, el esperará a que todas las pestañas que tengan abierto el sitio web (clientes) se cierren. Estos clientes están servidos por el viejo worker y mientras el nuevo estará esperando para activarse, como veremos en esta imagen:



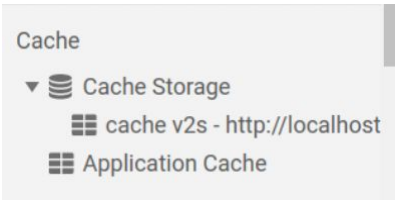
En este experimento hemos obviado este problema, ya que al resolver la promesa lanzamos *self.skipWaiting()* que forzará al nuevo service worker a pasar a la activación sin esperar y parando el viejo.
Por otro lado, para solucionar el problema de la limpieza de la versión 1 del caché debemos proceder al paso de activación.

4. **Activación:** Una vez el nuevo service worker ha sido instalado y la nueva versión deja de estar en uso, se lanza el evento “activate” que recogerá el nuevo worker. En esta etapa el service worker se convertirá en operativo y capaz de interceptar peticiones y servir recursos. Pero antes de activarse, es un buen momento para borrar el caché viejo existente que ya no vamos a utilizar. Con el siguiente código podemos solventarlo:

```
//Activate Event
self.addEventListener("activate", event => {
  event.waitUntil(
    caches.keys().then(cacheNames => {
      return Promise.all(
        cacheNames.map(cache => {
          if (cache !== cacheName) {
            console.log("Service Worker: Clearing old cache");
```

```
            return caches.delete(cache);
          }
        })
      );
    })
  );
});
```

Estás líneas de código, primero de todo, llaman a la función *waitUntil()* del evento como ya hemos visto en otras ocasiones, el cual esperará a que finalice la promesa que se le pasa como argumento. Esta promesa, pregunta por las claves de referencia de la API Cache, la cual le devolverá los nombres de todos los cachés almacenados. Con el *Promise.all()* conseguimos a que espere todas las promesas acaben y solo resolverá cuando todas lo hagan. Con toda la lista de cachés obtenida, filtraremos todas las que tengan un nombre de caché distinto al que hemos añadido en la etapa de instalación de este nuevo service worker. Si los nombres son distintos se eliminarán de la memoria caché.
Si hemos realizado los pasos correctamente, al guardar y recargar la página veremos como en la caché ha desaparecido la versión 1 y solo encontramos la versión 2:



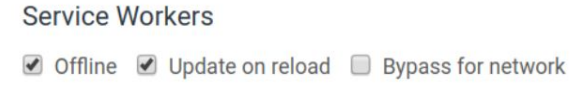
4. **Idle:** No hay mucha parte técnica en esta etapa. Simplemente, el service worker ya activo esperará a recibir algún evento como “fetch” o “message”, por ejemplo, para interceptar la petición y servir los recursos para al finalizar volver a este estado.
5. **Fetch:** Finalmente, hablaremos sobre una de las claves de los service workers, el evento “fetch”. Cuando se haga una petición inmediatamente se lanza este evento. Este evento dará el soporte que necesitamos para cubrir una experiencia de usuario continua en situaciones offline y de mala conexión. El service worker funciona como un servidor proxy, ya que todas las peticiones pasan a través de él. Cuando intercepta una, podemos configurar cómo procederá a responder, si con caché o a través de la red. Como se explica en esta tesis, existen muchas estrategias de cacheo y debemos elegir la que se adapte mejor a lo que requiera la PWA que queramos construir. A continuación introduciré el ejemplo más simple que se me ha ocurrido para entenderlo y después ya introduciré otras estrategias más complejas. En el siguiente código, devolveré los recursos que haya precacheado.

```
//Fetch Event
self.addEventListener("fetch", event => {
```

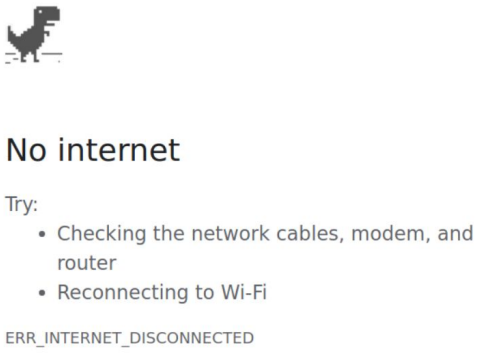
```
  console.log("Service Worker fetching");
  //Here we handle the requests in offline scenarios
  event.respondWith(
    // Since there isn't any connection
    //'fetch' this is going to fail,
    // so as this returns a promise we will 'catch' it.
    // Therefore, we will perform a function
    // that matches the request in our caches
    fetch(event.request)
      .catch(() => caches.match(event.request))
  );
});
```

Lo que podemos encontrar en este código es, primeramente, el *event.respondWith* para interactuar como la capa de red que responde a la solicitud. Aquí, tendremos que devolver la respuesta que deseamos, es por eso que utilizo el *fetch* con la petición que ha interceptado para obtener la respuesta de Internet. Hay que tener en cuenta que esto es una promesa, por lo que devolverá la respuesta de la red si se resuelve, mientras que si falla, lo gestionará buscando en la API de caché algún recurso que coincida con la petición. Si es así, se devuelve la respuesta en caché; de lo contrario, devolvería la respuesta fallida.

Ahora vamos a interactuar con el service worker para evaluar su comportamiento:
Si guardamos el código presentado arriba en el archivo del service worker podremos empezar a testear como trabaja.
En el primer escenario, recomiendo activar la opción offline del kit de desarrolladores del navegador antes de recargar la página.

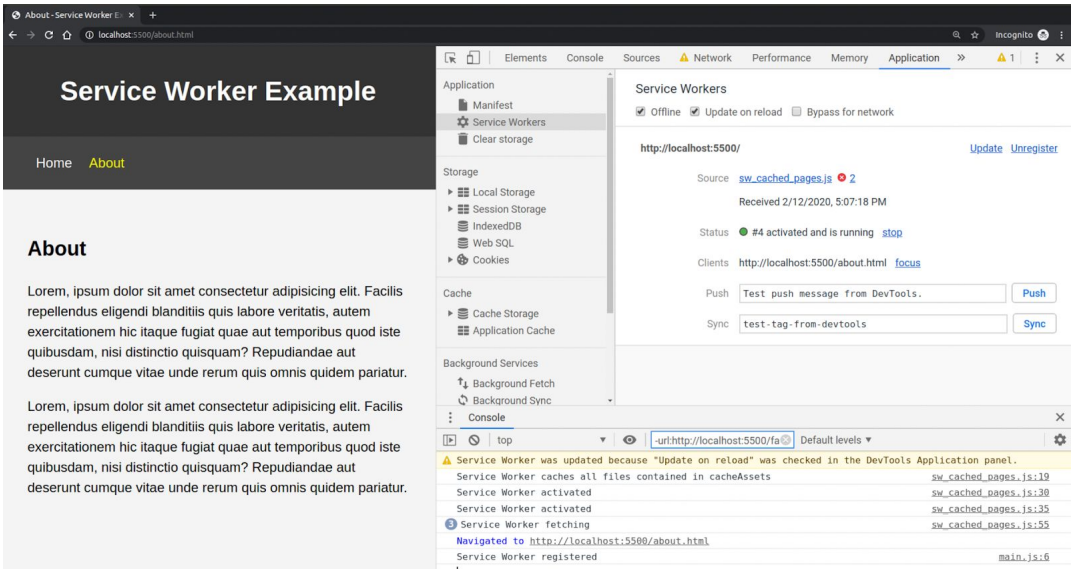


Podréis apreciar entonces que no podemos visitar la página ya que el service worker no ha sido registrado ni ha cacheado nada aún:



El siguiente escenario consiste en cargar primero la página, solo la de vinculada

al “index.html” y después activar la opción offline. Si ahora recargamos la página podremos ver como se refresca exitosamente y aparecen los log en la consola adecuadamente, de este modo, vemos que el service worker funciona como esperamos. Además, si visitamos la página que sirve el archivo “about.html” funcionará también. Esto es debido a que en la etapa de instalación hemos precacheado esta página, dotando al service worker de poder servirla en este momento a partir de la memoria caché. Esto puede variar, si se decidiese no cachear todas las páginas solo funcionarían los que estuvieran precacheadas.



6. **Terminated:** Este es el estado final que el service worker ejecuta automáticamente, pero no significa que sea la última etapa. Cuando el worker se mantiene en el estado de desocupado (“idle”) por un largo periodo, para liberar carga del navegador se mueve a este estado en el que duerme. A nivel técnico no hay nada que cubrir puesto que este proceso se realiza automáticamente por el service worker.

Anexo 4: Workbox y estrategias de cacheo

En este apartado evaluaremos el código de Workbox con lo que está pasando a bajo nivel por debajo en el service worker sin repetir las explicaciones que ya se han hecho en apartado del Estado del Arte de esta tesis. Las estrategias vistas hasta el momento son las siguientes:

- **Cache Only**

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

//Workbox
import {registerRoute} from 'workbox-routing';
import {CacheOnly} from 'workbox-strategies';

registerRoute(
  new RegExp('/app/v2/'),
  new CacheOnly()
);
```

- **Network Only**

```
//Plain JS
self.addEventListener('fetch', event => {
  event.respondWith(fetch(event.request));
  //or simply don’t call event.respondWith,
  // which will result in default browser behaviour
});

//Workbox
import {registerRoute} from 'workbox-routing';
import {NetworkOnly} from 'workbox-strategies';

registerRoute(
  new RegExp('/admin/'),
```

```
  new NetworkOnly()
);
```

- **Cache First**

```
//Plain JS
self.addEventListener('fetch', event => {
  event.respondWith(
    caches.match(event.request).then(response => {
      return response || fetch(event.request);
    })
  );
});

//Workbox
import {registerRoute} from 'workbox-routing';
import {CacheFirst} from 'workbox-strategies';

registerRoute(
  new RegExp('/styles/'),
  new CacheFirst()
);
```

- **Network First**

```
//Plain JS
self.addEventListener('fetch', event => {
  event.respondWith(
   fetch(event.request).catch(() => {
      return caches.match(event.request);
    })
  );
});

//Workbox
import {registerRoute} from 'workbox-routing';
import {NetworkFirst} from 'workbox-strategies';

registerRoute(
  new RegExp('/social-timeline/'),
  new NetworkFirst()
```

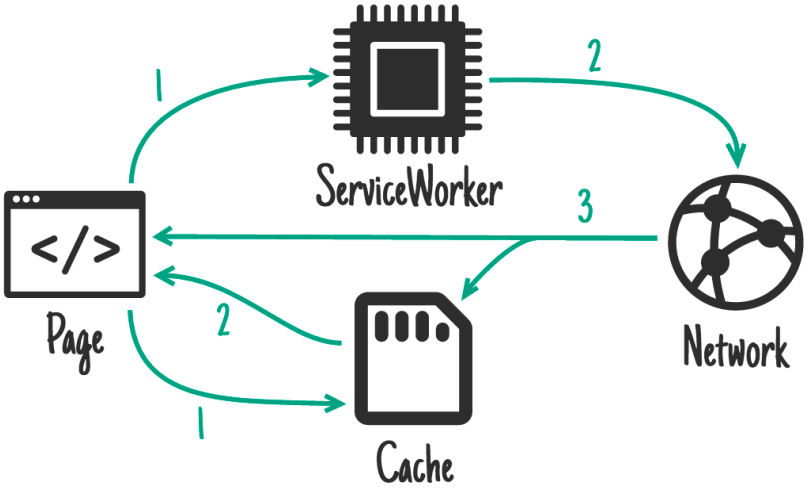
```
);
```

- **Stale-While-Revalidate**

```
//Plain JS
self.addEventListener('fetch', function(event) {
  event.respondWith(
     caches.open('mysite-dynamic')
       .then(function(cache) {
          return cache.match(event.request)
           .then(function(response) {
             var fetchPromise = fetch(event.request)
               .then(function(networkResponse) {
                 cache.put(
                   event.request,
                   networkResponse.clone()
                 );
                 return networkResponse;
                })
             return response || fetchPromise;
          })
       })
   );
});

//Workbox
import {registerRoute} from 'workbox-routing';
import {StaleWhileRevalidate} from
'workbox-strategies';

registerRoute(
  new RegExp('/images/avatars/'),
  new StaleWhileRevalidate()
);
```

Existen más estrategias de cacheo además de las presentadas hasta ahora, que son las que cubre Workbox por defecto. A continuación, presentaré algunas extensiones y variaciones que no cubre Workbox pero que también pueden ser de utilidad. De esta forma se entenderá que Workbox ni tan solo limita el número de estrategias sino que también permite añadir modificaciones para un uso más avanzado. Algunas de esas estrategias son las siguientes:

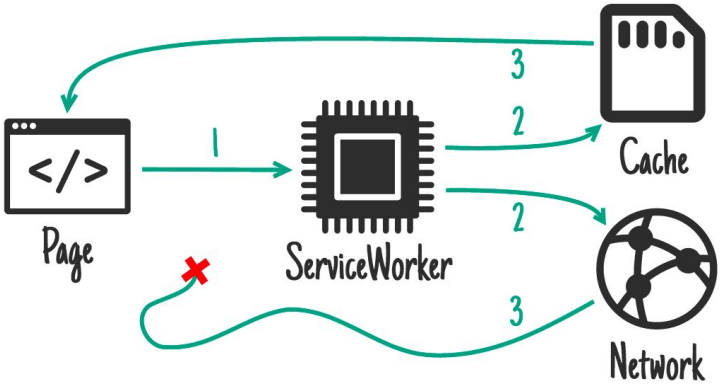
- **Cache Then Network.**
  En este enfoque la página requiere realizar dos peticiones, una a la caché y otra a la red, con el objetivo de usar la respuesta cache primer y cuando llegue la respuesta de la red, que suele ser más lenta, actualizar la página y el caché con esta respuesta. Este patrón es como una mezcla o variación de patrones anteriores. Funciona realmente bien para contenido que se actualiza frecuentemente pero se debe tener en cuenta que cuando la respuesta de la red llega, a veces podría reemplazar el contenido actual que se está mostrando en la web lo cual podría resultar demasiado brusco para el usuario si desaparece o cambia algún contenido que esté leyendo o con el que esté interactuando en ese momento.




```
//Plain JS
var networkDataReceived = false;

startSpinner();

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

caches.match('/data.json').then(function(response) {
  if (!response) throw Error("No data");
  return response.json();
}).then(function(data) {
  if (!networkDataReceived) {
    updatePage(data);
```

```
  }
}).catch(function() {
  return networkUpdate;
}).catch(showErrorMessage).then(stopSpinner);

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

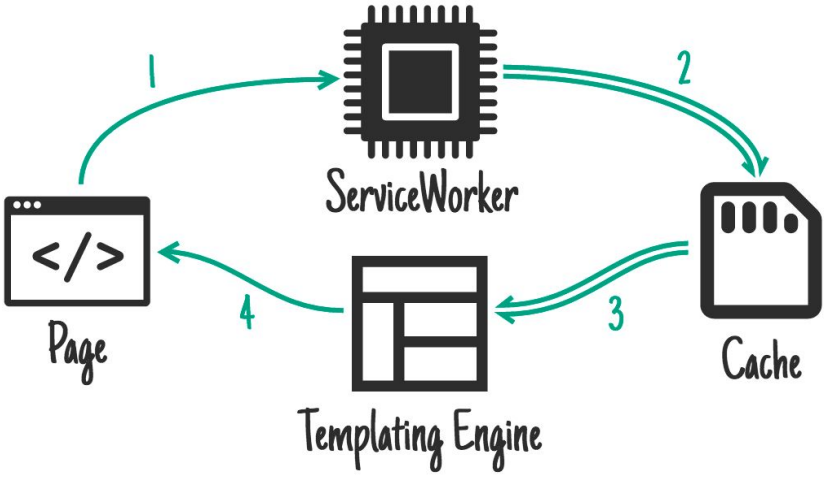
- **Cache and Network race.**
  Existe una incontable cantidad de combinaciones que podría variar en un escenario qué respuesta es más rápida, la cacheada o la de red, ya que dependen de muchos factores como el tipo disco duro, escáneres de virus o la conectividad a Internet. Sin embargo, ir a la red cuando el usuario ya tiene el contenido en el dispositivo puede ser una pérdida de tiempo, así que es importante tenerlo en cuenta.




```
function promiseAny(promises) {
  return new Promise((resolve, reject) => {
    promises = promises.map(p => Promise.resolve(p));
    promises.forEach(p => p.then(resolve));
    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)
    ])
  );
});
```

- **ServiceWorker-Side Templating.**
  Esta es la última modificación que voy a presentar aunque va más allá del alcance del proyecto pero considero que es realmente interesante de conocer. Esta solución es para la paginas que no pueden tener la respuesta del servidor cacheada, por ejemplo cuando se usa Server Side Rendering. Este es un método que se utiliza en algunas webs que permite servir el contenido ya renderizado, lo cual es más rápido, pero no siempre es posible. Sin embargo esto implicaría incluir el estado de los datos en el caché, lo cual no tiene demasiado sentido. Por lo tanto, cuando los developers utilicen este método deberán solicitar un JSON de datos junto con una plantilla para renderizarlo en su lugar.

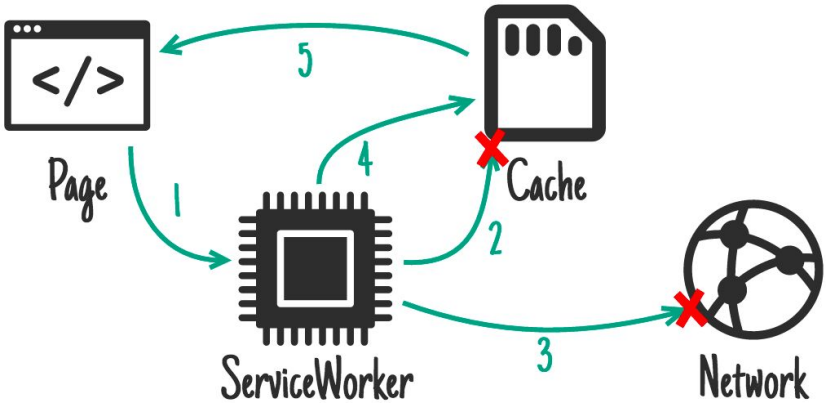



```
importScripts('templating-engine.js');

self.addEventListener('fetch', function(event) {
  var requestURL = new URL(event.request.url);

  event.respondWith(
```

```
    Promise.all([
      caches.match('/article-template.html')
       .then(function(response) {
        return response.text();
      }),
      caches.match(requestURL.path + '.json')
        .then(function(response) {
            return response.json();
        })
     ]).then(function(responses) {
     var template = responses[0];
     var data = responses[1];
      return new Response(
       renderTemplate(template, data), {
        headers: {
          'Content-Type': 'text/html'
        }
      });
    })
  );
});
```

Por último pero no por ello menos importante, me gustaría definir un método para lidiar con los escenarios más pesimistas, ya que se trata de una buena práctica:

- **Generic Fallback.**
  Esta estrategia se asegura de controlar qué contenido se devuelve cuando la petición a la memoria caché y a la red fallan. Por lo tanto, es posible devolver una respuesta sólida de contenido cuando el usuario está offline y sin caché sin deteriorar la experiencia y de esta forma evitamos desenganchar al usuario de la PWA.



Con todas estas explicaciones sobre las diferentes estrategias de caching y las que proporciona Workbox, podrías pensar que usar Workbox limita a utilizar una de las estrategias que define pero eso no es cierto. Workbox provee herramientas para que no tengas escribir el código de cero pero se pueden utilizar como deseemos y añadirle la lógica que creemos necesaria para sacar partido del service worker. Por ejemplo, en el código de abajo veremos cómo podemos filtrar una petición según su origen para evitar problemas de CORS:

```
self.addEventListener('fetch', (event) => {
  const {request} = event;
  const url = new URL(request.url);

  if (url.origin === location.origin && url.pathname === '/') {
    event.respondWith(new StaleWhileRevalidate().handle({event,
request}));
  }
});
```

Anexo 5: React:

Qué es un componente en React?

React se basa en una estructura de componentes. Una aplicación es un componente que está compuesto por componentes y estos, a su vez, pueden estar compuesto por más componentes y así sucesivamente. El nivel de profundidad es infinito y dependerá de cada desarrollador como quiera estructurar los elementos en el proyecto. Principalmente, existen dos criterios para definir si elemento debería conformar un nuevo componente o adherirse al componente actual:

1. El primero y más relevante consiste en determinar si el elemento se va a reutilizar en varias ocasiones. Un ejemplo podría ser los botones a lo largo de una aplicación. Si queremos reutilizar el diseño o la lógica una buena práctica sería crear el componente y reutilizarlo en las distintas secciones de la aplicación. El tener un único código nos facilitará mucho en el futuro a la hora de modificarlo y/o solucionar problemas a la vez que se consigue un código más legible. En conclusión, no se debe escribir el mismo código dos veces, o como se dice en el mundo de la programación: **DRY** "Don't repeat yourself".

2. El segundo criterio se basa en evaluar si un componente es demasiado farragoso o verboso. Por lo general, se intenta que estos sean sencillos y hagan referencia a un elemento y unas funcionalidades muy determinadas. React está optimizado para renderizar solo los componentes afectados por una interacción, por lo que componentes muy grandes nos hacen perder esta habilidad. Si el código que hemos creado se extiendo mucho, a partir de un cierto número de líneas deberíamos dividirlo en varios componentes.

Tipos de componentes y su estructura

Conceptualmente los componentes son como funciones de Javascript ya que reciben inputs arbitrarios llamados "*props*" y devuelven JSX, es decir, elementos de React que describen la interfaz de usuario. Fundamentalmente existen dos tipos de componentes: componentes de clase y componentes funcionales.

- **Componentes funcionales.** Los componentes funcionales son literalmente funciones que reciben como parámetro las *"props"* y devuelven JSX. La forma más simple de definir un componentes sería la siguiente:

```
function Welcome(props) {
    return <h1>Hello, {props.name}</h1>
}

//También podemos usar arrow functions
const Welcome = (props) => <h1>Hello, {props.name}</h1>
```

- **Componentes de clase.** Estos componentes se definen usando la sintaxis de clase del ES6. El siguiente código genera el mismo componente que el componente funcional de arriba:

```
import React from 'react';

class Welcome extends React.Component {
      render(){
            return <h1>Hello, {this.props.name}</h1>
      }
}
```

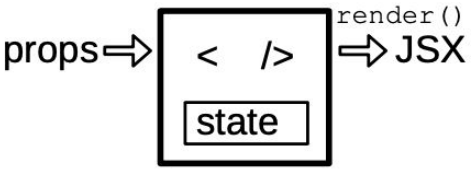
Los componen de clase tenían la capacidad de tener un estado y los "lifecycle hooks", los cuales permitían crear componentes con estado y nos permitía sacar más provecho de la lógica de React para sus componentes, pero a su vez, añaden más complejidad. Como veremos posteriormente, con los React Hooks se provee también de estas habilidades a los componentes funcionales y se presenta un paradigma muy interesante.
En el otro lado encontramos los componentes sin estados, los cuales no contienen ningún estado y simplemente reciben props para devolver JSX, como los que hemos visto hasta ahora.

Estado y Ciclo de vida

En los componentes de clase podemos crear un estado para generar contenido dinámico dentro del componente. Algunas características sobre el estado:

- El estado es un objeto plano de Javascript en el que podemos guardar los datos del componente.
- El estado se parece a las "*props*" pero es privado y solo se puede controlar por el mismo componente.
- Podemos variar el estado del componente en función de eventos, interacción con el UI, la red o lo que sea. Estos cambios tienen que expresarse dentro del componente si queremos modificar el estado.
- Cuando el estado se modifica React volverá a renderizar automáticamente el componente.

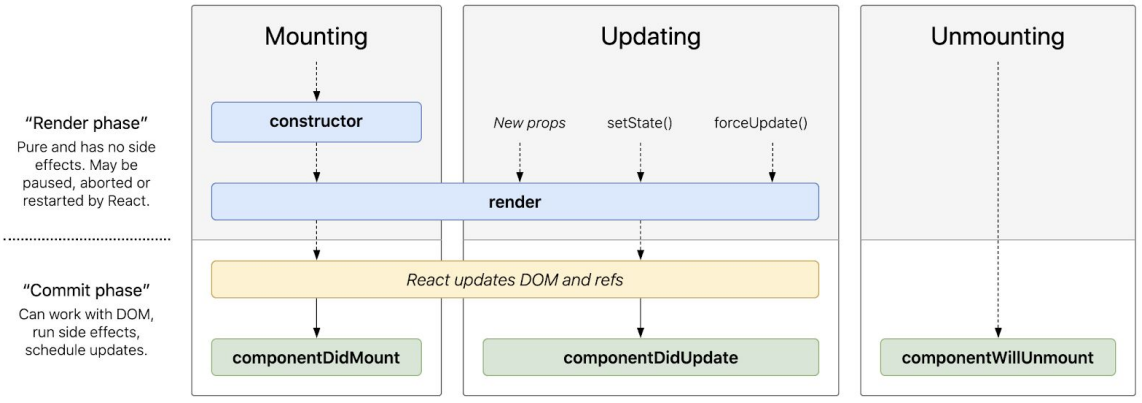
Con el estado, dotamos al componente de la capacidad de variar su forma a partir del mismo, no solo de las "*props*". El componente de clase debe implementar el método *render()*, el cual devolverá el JSX y siempre será llamado al renderizar un componente.



El ciclo de vida de un componente en React puede definirse como una serie de métodos que son invocados en diferente etapas de la existencia del componente. Podemos definir cuatro etapas que transcurren en el siguiente orden:

1. **Inicialización.** En esta fase se construye el componente con las *"props"* que se le proporciona y el estado inicial que se ha definido. Esto se lleva a cabo en el constructor de la clase.
2. **Montaje.** Esta etapa renderiza el JSX que devuelve el método *render()*.
3. **Actualización.** Es la fase en la que el estado se ha modificado y el componente se vuelve a renderizar para repintarse.
4. **Desmontaje.** Es la etapa final en la que el componente se elimina de la página.

A continuación mostraré un esquema que recoge las funciones más comunes que se lanzan en cada etapa, conocidos como "lifecycle hooks":



React Hooks

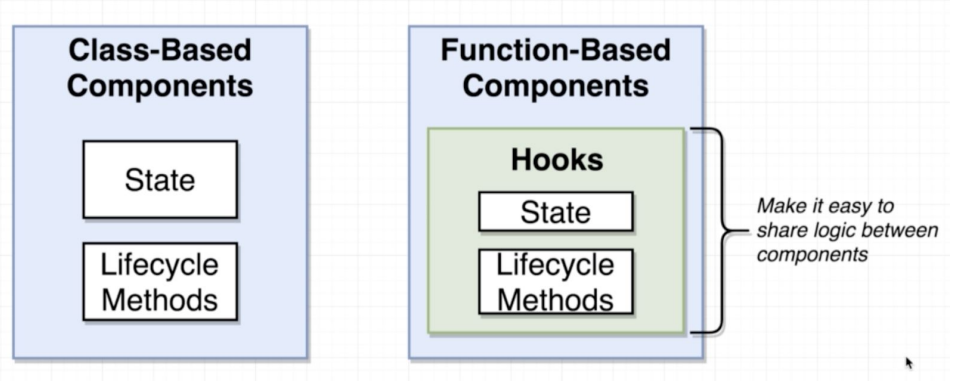
Hasta la versión 16.8, React ya podía reusar lógica con los componentes de clase. Sin embargo, a medida que estos componentes crecían eran menos legibles y complejos de reusar. Se solían utilizar "Higher Order Components (HOC)" que eran como *wrappers*, es decir, componentes que cubrían como una cap a otro proveyendo funcionalidades. De esta forma, se conseguía reusar la lógica de componentes pero podían llegar a quedar capas y capas creando un monolito inentendible.

Con la arquitectura que hemos visto hasta este punto se presentan tres problemas:

- Es complicado reusar la lógica entre componentes.
- Las clases y su estructura son más complicados que las funciones.

- A medida que los componentes crecen y aumenta su lógica, son cada vez más complejos y difíciles de entender.

Finalmente, en la versión 16.8 llegaron los React Hooks. Los React Hooks nos permiten usar estado, ciclo de vida y otras herramientas de React en los componentes funcionales.
Con los hooks podemos construir funciones reutilizables para administrar parcial o totalmente la lógica interna de un componente funcional del mismo modo que lo hacen los componentes de clase. De esta forma, los React Hooks facilitan la reutilización del código y mantienen un proyecto bien estructurado, por lo que permite limpiar nuestros componentes para crear funciones externas separadas que puedan utilizarse las veces que queramos, ya sea funciones asíncronas, solicitudes de API, etc. Además, los hooks nos permiten conectarnos a las características del estado y del ciclo de vida de los componentes funcionales. Es importante mencionar que React no tiene pensado eliminar los componentes de clases pero vamos a tratar esta potente herramienta.



Los React Hooks introducen así una nueva forma de estructurar los componentes y los proyectos. En el siguiente anexo, el **Anexo 5**, presentaré los Hooks ***useState*** y ***useEffect***, los cuales aportan el estados y los hooks para el ciclo de vida respectivamente, y los compararé con las mismas funcionalidades pero en componentes de clases. Como conclusión tras investigar sobre el rendimiento de los Hooks decidí aplicarlos en el proyecto, por lo que todos los componentes serán funcionales, algunos con estado y otros sin.

React Hooks en Bilingualy

Como he mencionado previamente, la aplicación se ha desarrollado con todos sus componentes funcionales excepto el archivo que contiene la vista principal, que debía ser un componente de clase porque Framework7 lo requería. De esta forma, la app está repleta de Hooks, y además, tecnologías como Redux y Firebase también han adoptado los Hooks y se han utilizado en el proyecto.

Lo que realmente era un reto fue crear Hooks propios, puesto que requería de haber entendido bien su función y por otra parte ser capaz de abstraer una lógica compartida entre componente y trasladarla un Hook. Finalmente, fui capaz de crear estos tres hooks:

- **usePrevious:** Este Hook utiliza el hook *useRef* para almacenar el valor actual que se le pasa como parámetro y devolver el valor inmediatamente anterior. En los componentes se ha utilizado para saber cuál era el estado anterior cuando el componente se vuelve a renderizar.

```
import { useRef, useEffect } from "react";

export default function usePrevious(value) {
  const ref = useRef();
  useEffect(() => {
    ref.current = value;
  });
  return ref.current;
}
```

- **useWindowDimensions:** Este hook configura un oyente que escuchará el evento que se lanza cuando el tamaño de la pantalla se configura y también cuando cambia. Esto se utiliza para conocer el tamaño desde el Javascript, no desde el CSS. Cuando el Hook se desmonta el oyente se borra. Este hook no solo se usa en componentes sino que también desde otro hook, lo cual abre nuevas posibilidades.

```
import { useState, useEffect } from 'react';

function getWindowDimensions() {
    const { innerWidth: width, innerHeight: height } = window;
    return { width, height };
}

export default function useWindowDimensions() {
    const [windowDimensions, setWindowDimensions] =
useState(getWindowDimensions());

    useEffect(() => {
        function handleResize() {
            setWindowDimensions(getWindowDimensions());
        }

        window.addEventListener('resize', handleResize);
        return () => window.removeEventListener('resize',
handleResize);
    }, []);
```

```
    return windowDimensions;
}
```

- **useMobileKeyboard:** Este hook utiliza el Hook *useWindowDimensions* para detectar cuando estamos en un móvil y se ha desplegado el teclado virtual, ya que este reducirá considerablemente el tamaño disponible que el usuario puede ver y los componentes lo utilizan para reorganizar la vista.

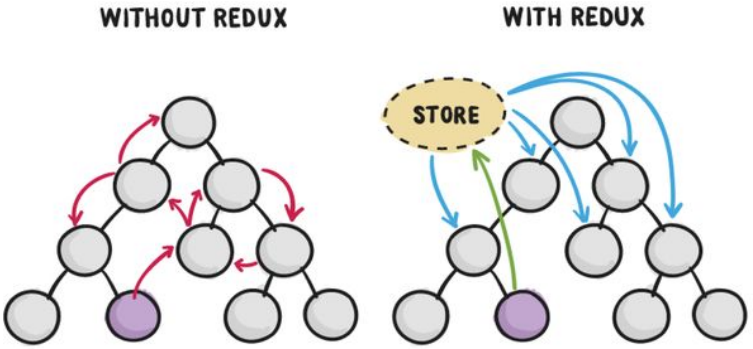
```
import { useState, useEffect, useRef } from "react";

import useWindowDimensions from "./UseWindowDimensions";

export default function useMobileKeyboard() {
  const [keyboardOpen, setKeyboardOpen] = useState(false);
  const initialHeight = useRef();
  const { height, width } = useWindowDimensions();

  //Set initial viewport height to handle mobile keyboard
  useEffect(() => {
    initialHeight.current = height;
  }, []);

  //If mobile and viewport height changes set keyboard to open
  useEffect(() => {
    setKeyboardOpen(width < 450 && height < 0.8 *
initialHeight.current);
  }, [height]);

  return keyboardOpen;
}
```

Redux

En muchas ocasiones surge la necesidad de acceder de manera más sencilla a las variables de estado. La metodología que más sencilla que sugiere React es ir pasando la información desde los componentes padres a los hijos a través de las props, y desde los componentes hijos a los padres a través de callbacks. El problema aparece cuando queremos compartir información entre componentes cuyo parentesco es muy lejano. En ese caso el valor de las variables debe ascender desde el primer componente a través de la jerarquía hasta que llegue al componente padre que tienen en común, para posteriormente enviar esta información dirección descendente hasta conseguir alcanzar el componente destino. Cuando más grande es la aplicación, más casos encontraremos en los que debemos aplicar este procedimiento, por lo que el flujo global de la aplicación

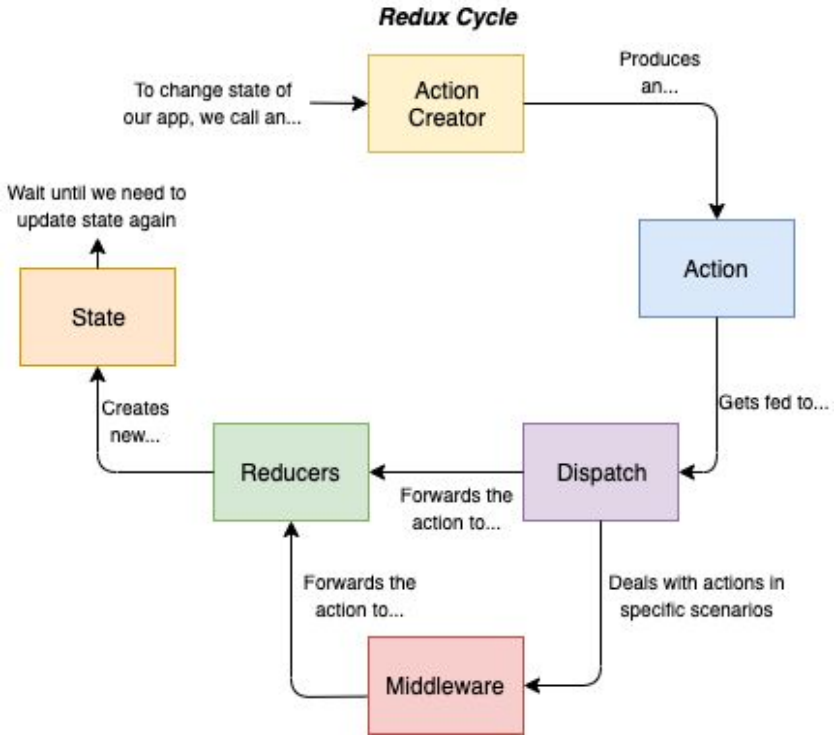
se vuelve un completo caos. Existen varias soluciones posibles para facilitar este flujo de comunicación con el estado y presentaré la que escogí: Redux.

Redux es una pequeña librería de Javascript que proporciona las herramientas para que dar a cada componente de React la parte del estado global que necesita. Redux es también un patrón de diseño que nos permite crear un flujo óptimo y estructurado de los datos en toda la aplicación. El estado global se gestiona fuera de React y se conoce como ***Store***.
Aplicando la filosofía de Redux, el componente origen enviaría el valor de la variable directamente al estado global, y el componente destino recibiría esta propiedad sin tener que pasar por toda la jerarquía, garantizando así un flujo de comunicación mucho más limpio.



Flujo y conceptos básicos de Redux

A continuación, veremos cómo fluye toda la información en la arquitectura de Redux, desde los que un componente intenta iniciar un cambio en el estado global hasta que recibe un cambio:



En la imagen observamos los conceptos básicos para entender el flujo de Redux. A continuación explicaré cada uno de los elementos que conforman este ciclo:

- **Action.** La única forma de cambiar el estado global es a través de acciones. Cuando lanzamos una acción con el método *"dispatch"* esta repercute en un cambio en el *store***.** A nivel práctico es importante saber que el *store* es un objeto inmutable y lo que realmente sucede es que se genera uno nuevo en cada cambio. Las acciones deben contener un "*type*" que identifica la acción para que posteriormente los reducers ejecuten una variación u otra. Adicionalmente, pueden contener un "*payload*" con datos por si son necesarios para el cambio de estado.

```
{
  type: 'ADD_ARTICLE',
  payload: { title: 'React Redux Tutorial', id: 1}
}
```

- **Action Creators.** Es considerado una buena práctica envolver las acciones con una función y dicha función es el *action creator*.

```
export function addArticle(payload){
  return {type: "ADD_ARTICLE", payload}
};
```

- **Reducer.** Cuando se lanza una acción llega a los reducers y estoc calculan el siguiente estado en función del "*type*" de dicha acción.

```
const initialState = {
  articles: []
};

function rootReducer(state = initialState, action) {
  if (action.type === "ADD_ARTICLE"){
    return Object.assign({}, state, {
      articles: state.articles.concat(action.payload)
    });
  }
  return state;
}
```

- **Middleware.** Se trata de un nodo opcional en el esquema básico del flujo de Redux. Dado que este fue diseñador para realizar acciones síncronas, los middleware son funciones capaces de interceptar la acción y ejecutar alguna lógica antes de que llegue al reducer. De esta forma obtenemos la capacidad de

ejecutar funciones asíncronas como por ejemplo realizar peticiones a un servidor. Otras situación interesante sería cuando queremos sacar lógica de los componentes para conseguir que sean más reusables y quizás, en función de esta lógica, queremos lanzar otra acción. Bien, los middlewares nos ofrecen también esta habilidad de poder llamar otras acciones y realizar esta lógica aislada dentro del middleware convirtiéndola en reusable y más fácil de testear, manteniendo los componentes más limpios.
Son muchas las librerías que ofrecen el servicio de middleware pero para este proyecto se eligió **Redux-Thunk**, ya que era una de las más simples y utilizadas. Un ***thunk*** es una función que actúa como un envoltorio de una expresión que retrasa su evaluación. Para que se entienda con un ejemplo simple, la siguiente expresión se calcula inmediatamente:

```
const sum = 1 + 2;
```

En cambio, si escribimos una función:

```
const sum = () => 1 + 2;
```

Estamos retrasando la ejecución de esta expresión para más tarde. Del mismo, se aplica Redux-Thunk en Redux. Creamos *thunks* (funciones) que serán ejecutadas luego y recuperarán la información que necesitemos, normalmente de un backend.

Un ejemplo de *thunk* utilizado en el proyecto sería el siguiente:

```
const singOut = () => async (dispatch,
                             getState,
                             {getFirebase}
                             ) => {
  const firebase = getFirebase();

  try {
    await firebase.auth().signOut();
    await firebase.logout();
    dispatch({ type: SIGNOUT_SUCCESS });
  } catch (e) {
    dispatch({ type: SIGNOUT_FAILED });
  }
};
```

Como podemos ver, para la función de cerrar sesión ***signOut*** se ha aplicado un thunk, es decir, un función asíncrona que esperará a que varias promesas se resuelvan. De ser así haría *dispatch* de la acción “SIGNOUT_SUCCESS”, si al menos una fallase el *dispatch* sería de “SIGNOUT_FAILED”.

Redux en Bilingualy

A medida que se desarrolló el proyecto la store fue creciendo hasta contener tres secciones:

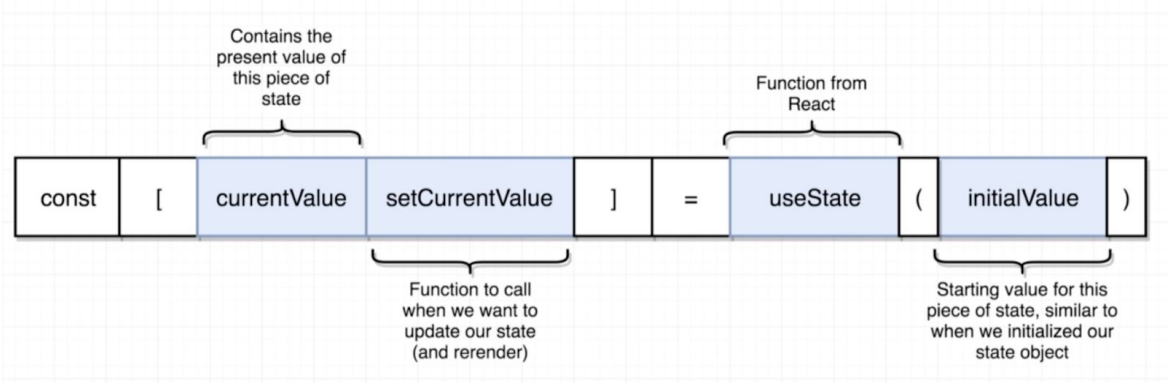
- **Auth.** Esta parte del estado gestiona las llamadas asíncronas a firebase para realizar las acciones de *Sign In, Sign Up* y *Sign Out*, es decir, gestiona el servicio de autenticación.
- **Device.** Esta sección se encarga de controlar si la PWA tiene conexión actualmente a Internet o no. De cara a un futuro también se encargaría de almacenar por ejemplo, los periféricos disponibles del télefono. Su funcionalidad es gestionar la información compartida del dispositivo.
- **SocialStory.** Este parte gestiona todo el proceso de creación de una historia social, de esta forma los componentes se liberan de almacenar esta información durante los distintos pasos y al finalizar podemos subir esta historia a Firestore. La lógica asíncrona de guardar o borrar historias sociales también se guarda aquí para liberar a los componentes y aprovechar el middleware.

Anexo 5: Desarrollo de React Hooks

Existen una lista enorme de Hooks y además, podemos crear nuestros propios Hooks. Probablemente los dos que primero debemos conocer son el *useState* y *useEffect*

El hook de estado: useState

Este hook se declara con la siguiente estructura:



Como podéis ver, esta función retorna dos elementos: el primero es un objeto que contiene el valor actual del estado y el segundo una función para actualizarlo. Durante el renderizado inicial, el estado devuelto (*currentValue*) es el mismo que el valor pasado como primer argumento (*initialValue*). Por otro lado, la función setCurrentValue acepta un nuevo valor de estado y pone en cola una nueva representación del componente. Es similar a this.setState en una clase, pero no combina el estado antiguo y el nuevo.

A continuación, evaluaremos su funcionalidad con un ejemplo real. En este primer bloque encontraremos un componente de clase con estado, el cual un parágrafo que muestra un contador almacenado en el estado, inicializado en 0 y que incrementa al clicar un botón.

```
import React from "react";

export default class Example extends React.Component {
  // Declare a new state variable, which we'll call "count"
  constructor(props) {
    super(props);
    this.state = { count: 0 };
  }

  render() {
    return (
      <div>
        <p>You clicked {this.state.count} times</p>
        <button
          onClick={() => this.setState({count: this.state.count +
1})}>
```

```
          Click me
        </button>
      </div>
    );
  }
}
```

Por otro lado, aquí muestro un ejemplo de cómo construir la misma funcional con un componente funcional y usando el hook *useState*:

```
import React, { useState } from "react";

function Example() {
  // Declare a new state variable, which we'll call "count"
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

Así de simple, como hemos podido ver, ahora el estado no se almacena en un "objeto estado" que contiene todas las variables, sino que utilizaremos un hook *useState* para cada variable de estado que queramos añadir.

Hook de efecto: useEffect

El hook de efecto nos da la habilidad de realizar efectos, como su propio nombre indica, en las etapas del ciclo de vida de un componente funcional. Lo que más, cubre y unifica las tres funciones core del ciclo de vida: *componentDidMount, componentDidUpdate* and *componentDidUnmount*.

A continuación, plantearé diferentes escenarios con el objetivo de mostrar su utilidad. Existen muchos escenarios posibles pero antes considero necesario mostrar la función básica y principal sin elementos extra. Primero mostraremos el componente de clase equivalente:

```
class Example extends React.Component {
  constructor(props) {
    super(props);
```

```
    this.state = {
      count: 0
    };
  }

  componentDidMount() {
    document.title = `You clicked ${this.state.count} times`;
  }

  componentDidUpdate() {
    document.title = `You clicked ${this.state.count} times`;
  }

  render() {
    return (
      <div>
        <p>You clicked {this.state.count} times</p>
        <button onClick={() =>
          this.setState({ count: this.state.count + 1 })}
        >
          Click me
        </button>
      </div>
    );
  }
}
```

Ahora veamos con un componente funcional. Tener en cuenta que el hook que veremos a continuación se ejecutará solamente cuando el componente se cree y cuando se actualice:

```
import React, { useState, useEffect } from 'react';

function Example() {
  const [count, setCount] = useState(0);

  // Similar to componentDidMount and componentDidUpdate:
  useEffect(() => {
    // Update the document title using the browser API
    document.title = `You clicked ${count} times`;
  });

  return (
    <div>
```

```
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

Este ejemplo extiende el que he presentado con el hook *useState*. Aquí estamos informando a React que queremos realizar algunos efectos cuando el componente se renderice. React recordará la función que le hemos pasado, conocida como efecto, para llamarla posteriormente después de realizar la actualización del DOM. En este ejemplo, el efecto se encargará de configurar el título del documento después de cada renderización, incluso la primera vez, cuando el componente se ha montado. El *useEffect* nos permite realizar cualquier función, incluso la búsqueda de datos o llamar a cualquier API imperativa. De ahora en adelante, vamos a evaluar varias implementaciones que fortalecen este hook clasificándolas si necesitan un *cleanUp* o no. El *cleanUp* es el efecto que se guardará para ejecutarse cuando se vaya a desmontar el componente.
Para darle sentido a los efectos que necesitan un cleanUp, imaginemos que queremos configurar una suscripción a alguna fuente de datos externa. Al finalizar y desmontar el componente, deberíamos limpiar esta suscripción para no introducir una pérdida de memoria. Básicamente, esto significa que tenemos que ejecutar algo de código cuando el componente se está desmontado. Veamos un ejemplo con un componente de clase:

```
class FriendStatus extends React.Component {
  constructor(props) {
    super(props);
    this.state = { isOnline: null };
    this.handleStatusChange = this.handleStatusChange.bind(this);
  }

  componentDidMount() {
    ChatAPI.subscribeToFriendStatus(
      this.props.friend.id,
      this.handleStatusChange
    );
  }

  componentWillUnmount() {
    ChatAPI.unsubscribeFromFriendStatus(
      this.props.friend.id,
      this.handleStatusChange
    );
  }
```

```
  handleStatusChange(status) {
    this.setState({
      isOnline: status.isOnline
    });
  }

  render() {
    if (this.state.isOnline === null) {
      return 'Loading...';
    }
    return this.state.isOnline ? 'Online' : 'Offline';
  }
}
```

Para convertirlo en un componente funcional tenemos que saber que cuando el *useEffect* retorna una función, React la interpretará como el cleanUp. Por lo tanto, nuestro hook se suscribirá y se dará de baja con un solo *useEffect* Hook, lo cual tiene todo el sentido, ya que la finalidad de ambos efectos se vincula al mismo sujeto. Sería algo tal que así:

```
import React, { useState, useEffect } from 'react';

function FriendStatus(props) {
  const [isOnline, setIsOnline] = useState(null);

  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }

    ChatAPI.subscribeToFriendStatus(props.friend.id,
handleStatusChange);
    // Specify how to clean up after this effect:
    return function cleanup() {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id,
handleStatusChange);
    };
  });

  if (isOnline === null) {
    return 'Loading...';
  }
  return isOnline ? 'Online' : 'Offline';
}
```

Existe otro importante escenario para relevar más funcionalidades sobre este hook y es la comparación con el *ComponentDidUpdate* ya que este recibe como parámetros las props previas y el estado previo antes de actualizarse, lo cual nos permite decidir qué acción realizar en función de estas variables. Es una situación que se afronta comúnmente:

```
componentDidUpdate(prevProps, prevState) {
  if (prevState.count !== this.state.count) {
    document.title = `You clicked ${this.state.count} times`;
  }
}
```

Por el lado de los Hooks, podemos decirle al useEffect que se ejecute l montarse y se actualice solo si ciertas variables han cambiado entre actualizaciones. Para hacer esto, podemos pasarle una lista con las variables que ha de tener en cuenta como segundo argumento al hook:

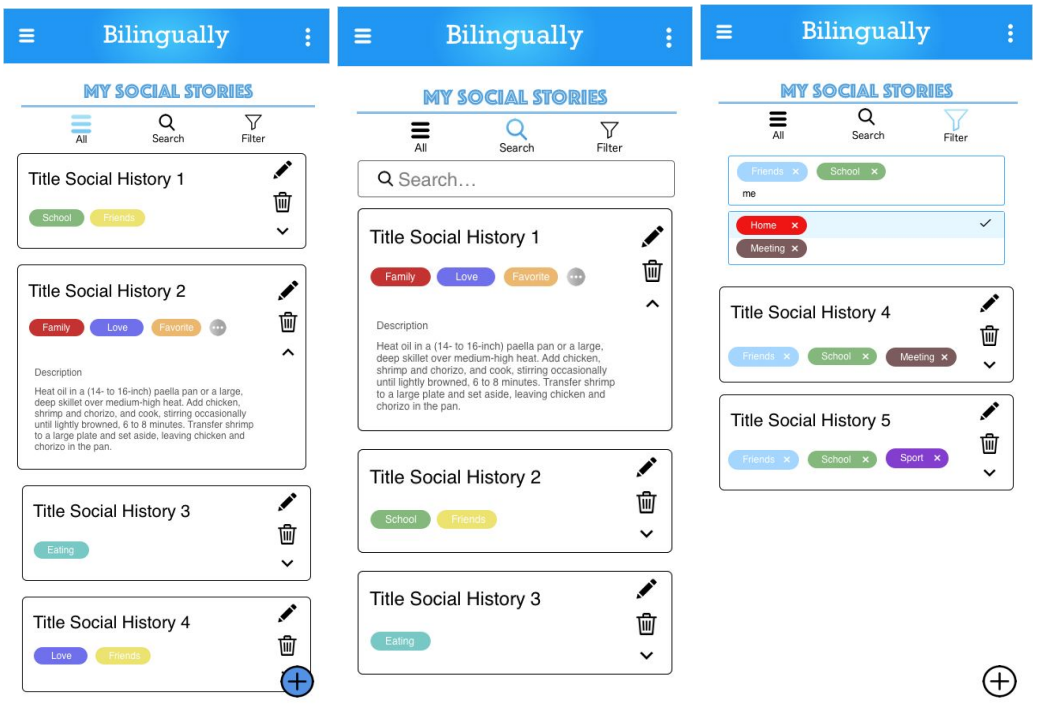
```
useEffect(() => {
  document.title = `You clicked ${count} times`;
}, [count]); // Only re-run the effect if count changes
```

Y por último pero no menos importante, hay un útimo escenario para cubrir. Se puede dar el caso en que queramos que un efecto y un *cleanUp* se ejecuten solo al montar y desmontar el componente. Para conseguirlo, pasaremos una lista vacía como segundo argumento y React lo traducirá como que el efecto no depende de las props ni del estado. De este modo, se ejecutará el efecto al montarse y el *cleanUp* al desmontarse y en ninguna otra renderización lo hará.

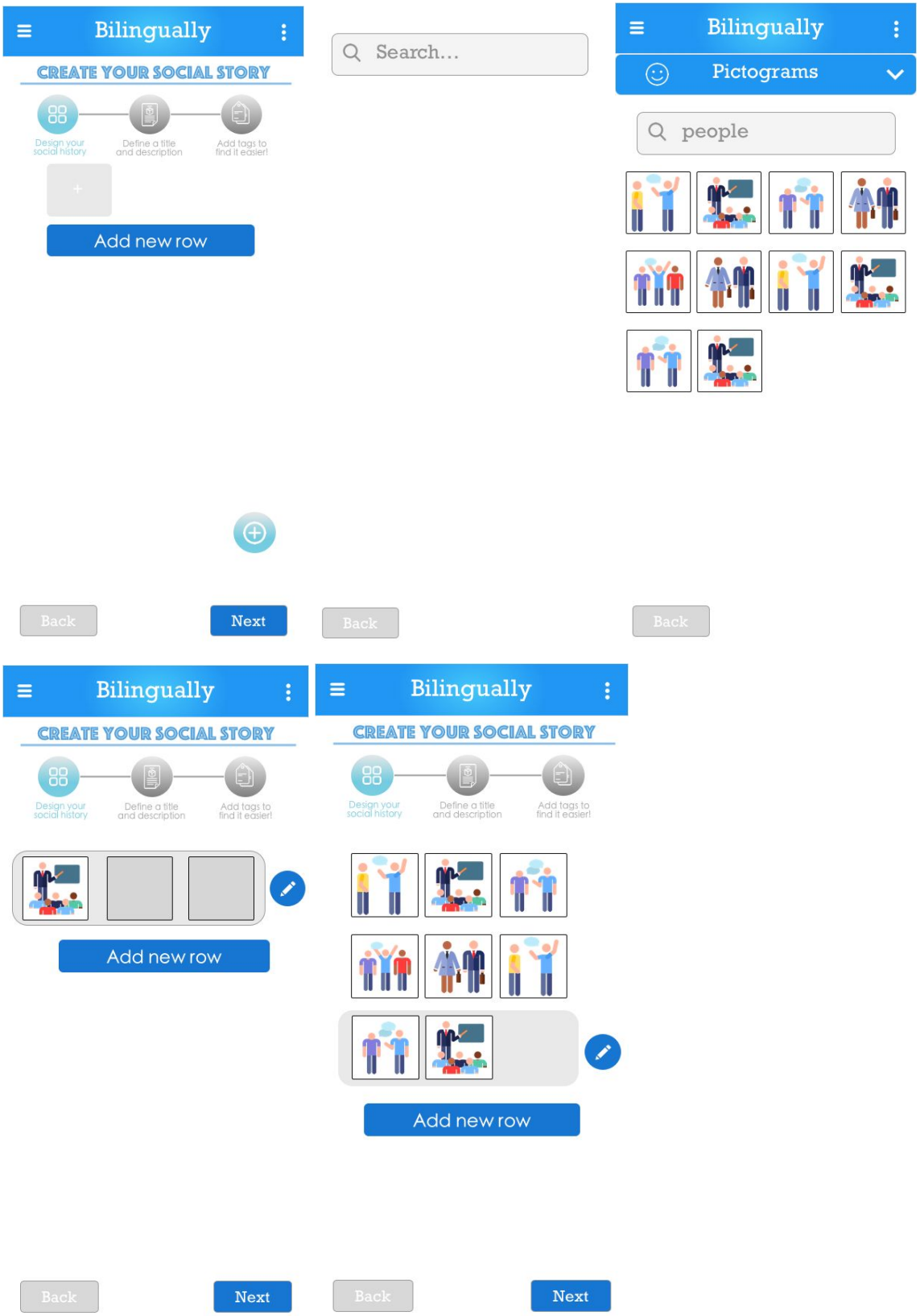
```
useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }

    ChatAPI.subscribeToFriendStatus(props.friend.id,
           handleStatusChange);
    return function cleanup() {
      ChatAPI.unsubscribeFromFriendStatus(
         props.friend.id,
         handleStatusChange
      );
    }, []}
);
```

Anexo 6: Sketch inicial de la aplicación

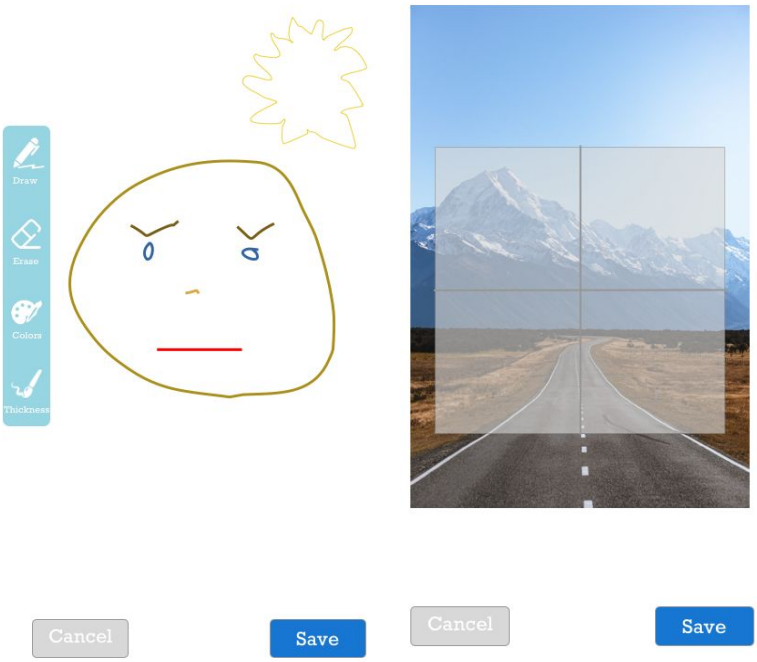
Vista de la pantalla inicial de la aplicación. En ella encontramos todas las historias sociales y los distintos filtros para realizar búsquedas:



Creador de historias sociales, Paso 1. En el podemos ver cómo podemos buscar pictogramas a partir de palabras y ir añadiéndolas para montar la historia:

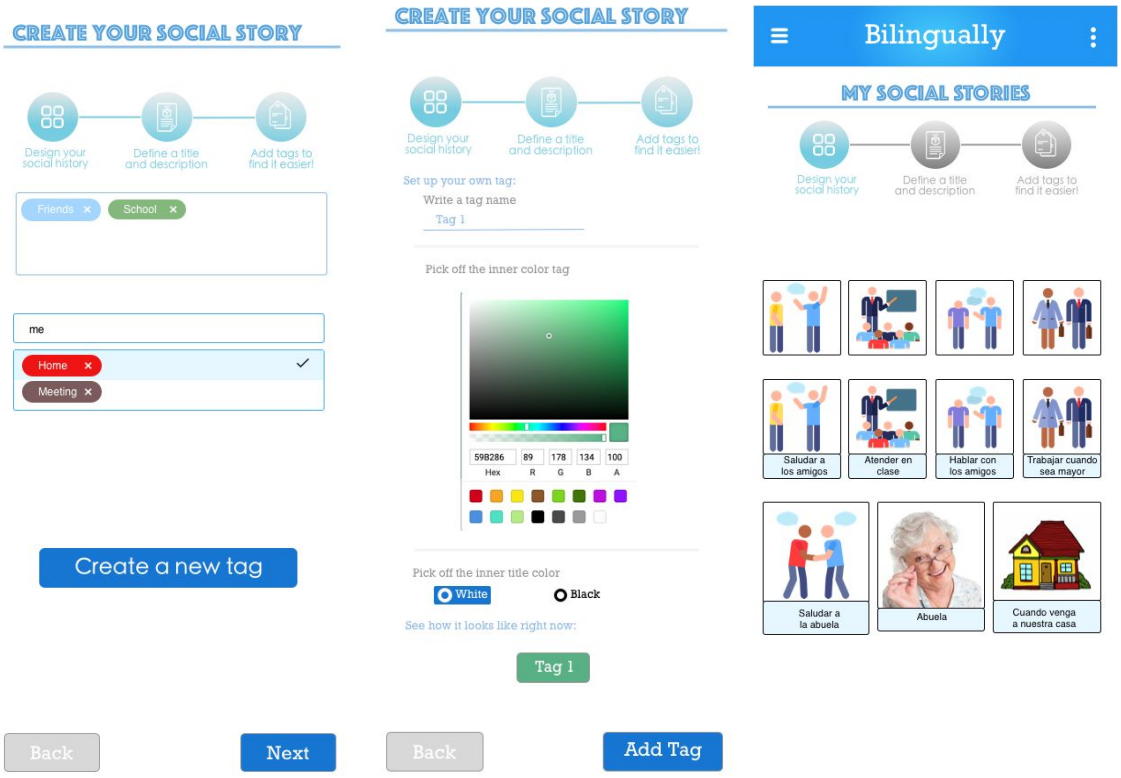


Además de los pictogramas que nos ofrecerá la API, la app también iba a contar inicialmente con una herramienta para añadir dibujos y otra para acceder a la cámara del dispositivo y añadir imágenes.



La app iba a poder definir cuántos pictogramas colocar en cada fila y que texto queremos añadir debajo de cada pictograma:

En los siguientes pasos de la creación, podemos añadir título, descripción, la privacidad y las etiquetas de la historia para finalmente poder visualizar el resultado final:



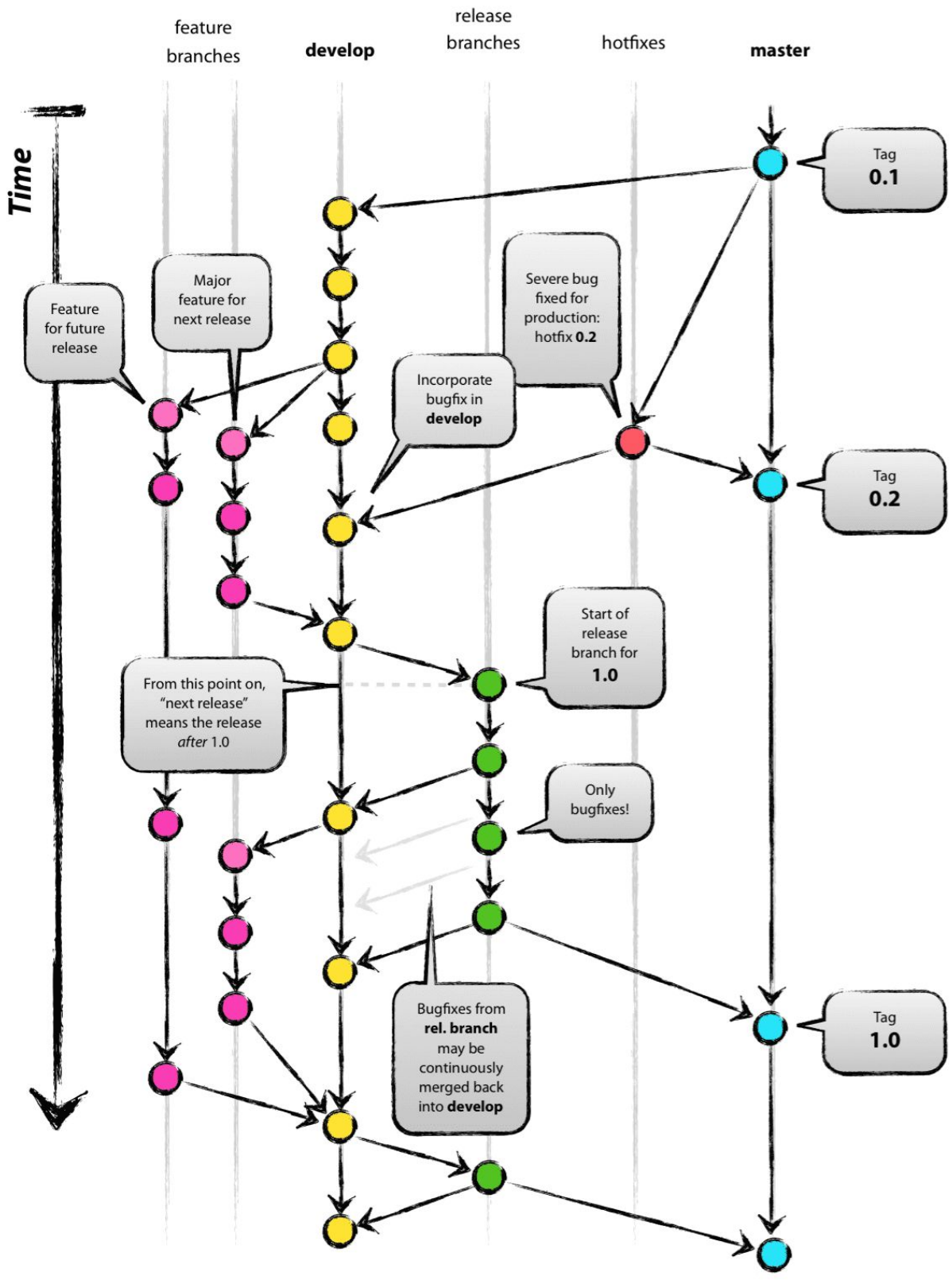
Anexo 7: Branching model de Git

Este modelo se basa en dos ramas principales: **master** y **develop**. Primero, siempre se trabaja sobre la rama de develop y cuando tengamos una versión estable del proyecto realizaremos un merge a la rama master. Un merge es la acción en la que se combina el código de una rama a otra para que esta última adopte los cambios de la primera. Una vez hecho esto se compila el código de esta rama y se sube al servidor web. De este modo, la rama master representará la versión actual de producción, es decir, la que se está sirviendo públicamente para los clientes/usuarios. Por lo tanto, solo se publicarán versiones estables y maduras. En un código sólido esto se traduce en que se trata de una versión testeada y tiene el "check" del *Quality Analyst*. En develop, en cambio, irá añadiendo las funcionalidades, que se traduce en más código y archivos, hasta que volvamos a tener una versión disponible, madura y lo suficientemente estable para volver a repetir el proceso de merge con master y actualizar así la versión de producción.

No obstante, los cambios en develop no se suelen hacer directamente en la rama. Para ello, se establecen ramas temporales que se clasifican principalmente en tres tipos:

- **Feature.** Este tipo de rama tiene como objetivo añadir una nueva funcionalidad al software. Un ejemplo en mi proyecto sería la feature que añadía la vista de creación de historias sociales. Cuando esta nueva funcionalidad está acabada, se puede realizar un merge con develop y esta ya se encargará de hacerlo con master cuando creamos convenientes, ya que quizás precisa de añadir otras funcionalidades para cobrar sentido.
- **Release.** La intención de esta rama es de preparar una versión de producción para una nueva entrega a master. Esta rama recibe cambios de corrección de pequeños errores o modificaciones de último momento, además de la corrección del número de versión. Una vez realizadas estas modificaciones, se realiza un merge tanto a la rama master como a la rama develop, para que puedan seguir avanzando a partir de la misma versión. En el proyecto no se ha trabajado con esta rama debido a que no se ha precisado de un versionado ni de releases y directamente se ha apuntado desde develop a la rama de master.
- **Hotfix.** Cuando detectamos un fallo en la versión de producción, surge la necesidad de corregirlo inmediatamente. Para ello, creamos la rama hotfix, en la cual aplicaremos los cambios necesarios para solventar el problema. Posteriormente realizaremos un merge a master y develop. En este proyecto, afortunadamente, no se ha precisado de esta rama por el momento.

Cuando cada una de estas ramas temporales han implementado la función que se les requería se procede a cerrarla y si se quisiera a eliminarlas, puesto que sus cambios ya se encuentran en las ramas principales. La siguiente imagen ilustra a la perfección el protocolo descrito:


feature branches
develop
release branches
hotfixes
master
Time
Tag 0.1
Feature for future release
Major feature for next release
Severe bug fixed for production: hotfix 0.2
Incorporate bugfix in develop
Tag 0.2
Start of release branch for 1.0
From this point on, "next release" means the release after 1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Tag 1.0

Anexo 8: Resultados del UI de Bilingualy

A continuación, se mostrarán las distintas capturas de pantalla de la aplicación.

Gracias al App manifest la aplicación se ve así instalada en un dispositivo móvil:

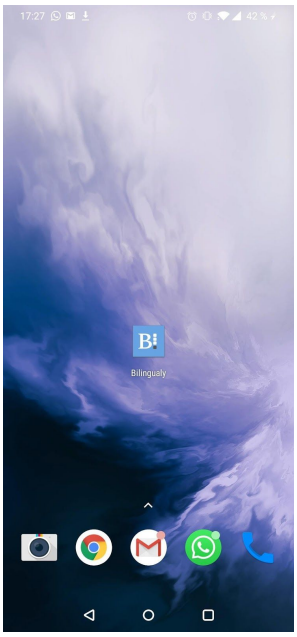


Figura 20: Captura móvil con el icono de Bilingualy

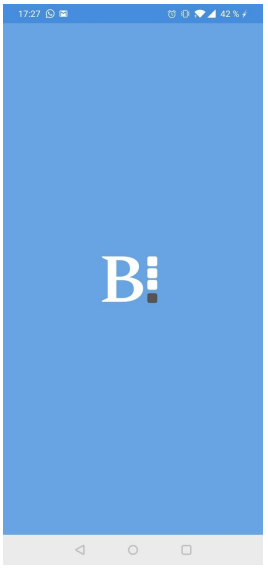


Figura 22: Splash Screen

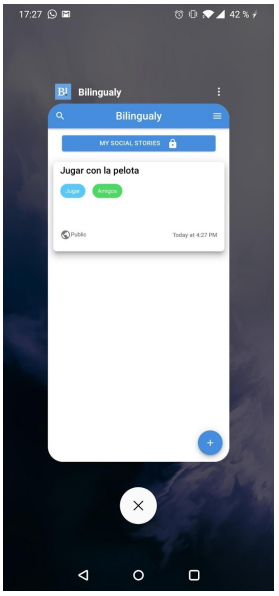


Figura 23: Captura del tasks viewer de un móvil

Vistas del sistema de autenticación y de registro:

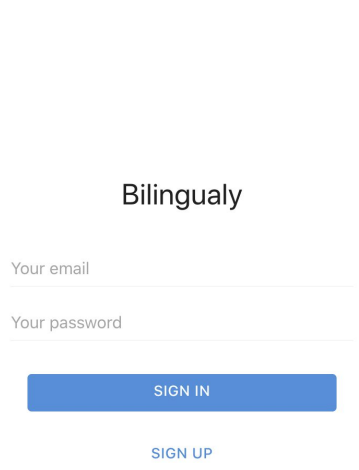


Figura --: Vista “SignIn”

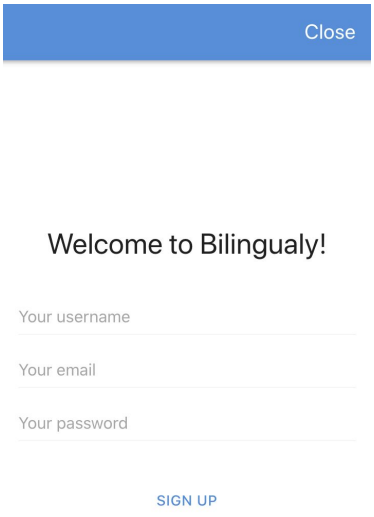


Figura --: Vista “SignUp”

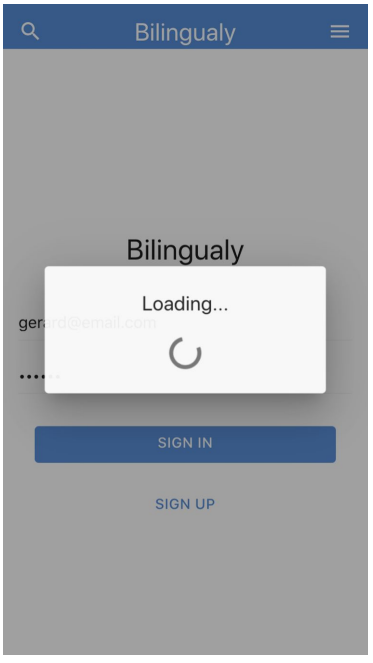


Figura --: Vista cargando autentiación

Vista adaptativa del diseño basado en mobile first:

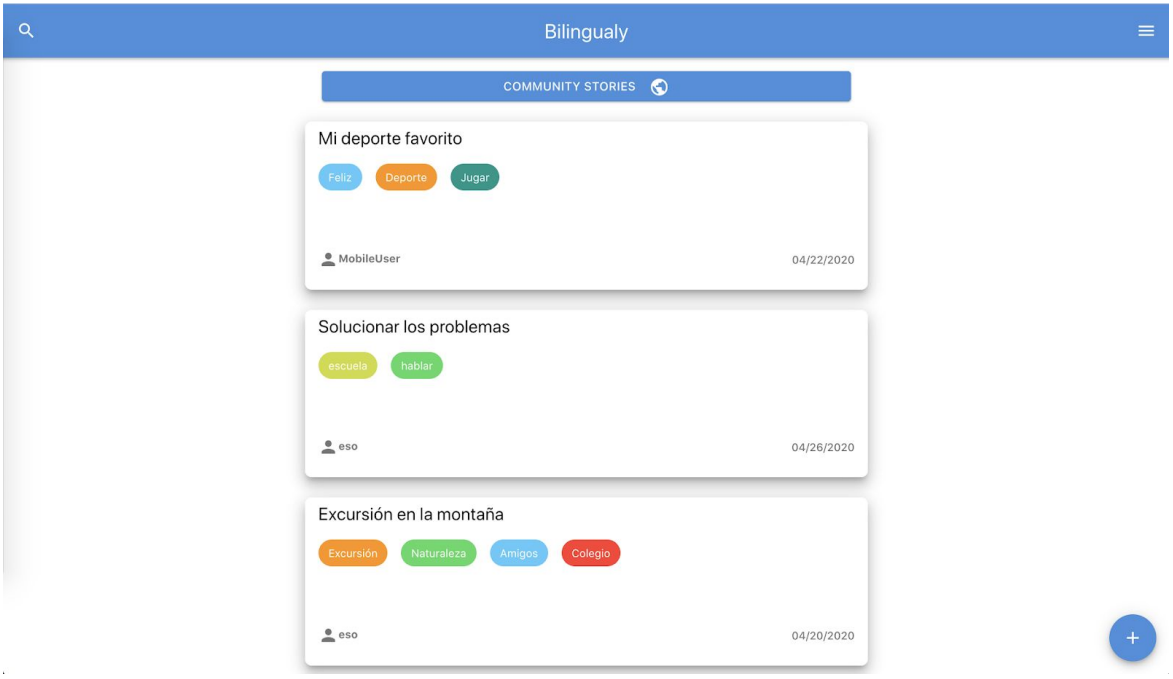


Figura --: Vista de la sección “Community Stories” desde ordenador

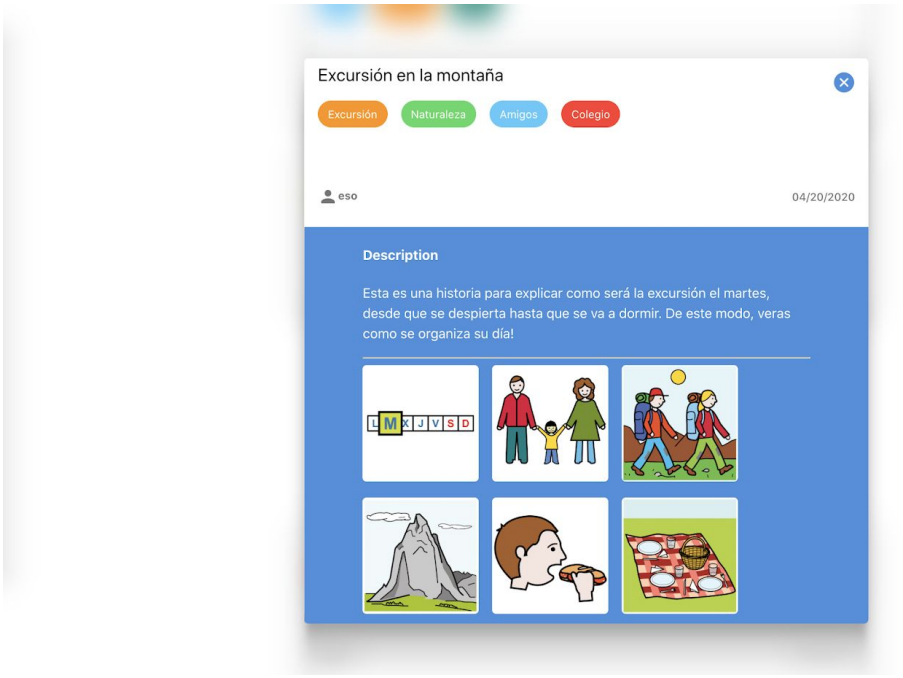


Figura --: Vista de una historia social desde ordenador

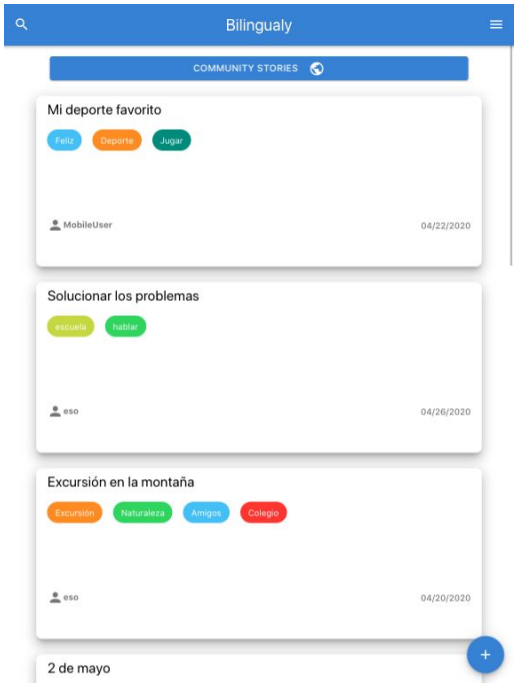


Figura --: Vista pantalla principal desde tablet

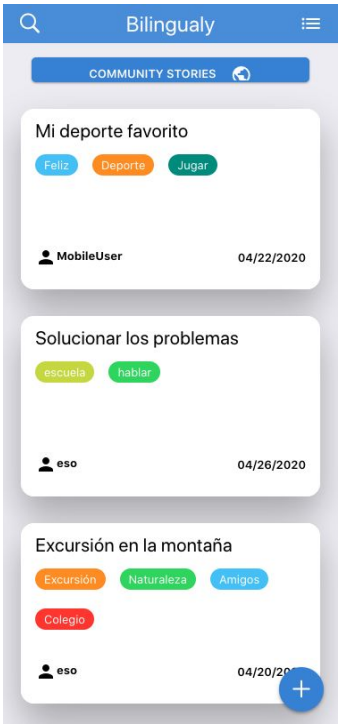


Figura --: Vista pantalla “Community” desde iPhone X

Figura --: Vista pantalla “Community” desde Android

A continuación, presentaré las vistas de la pantalla de “My Stories”

Figura --: Vista “My Stories” con el esqueleto

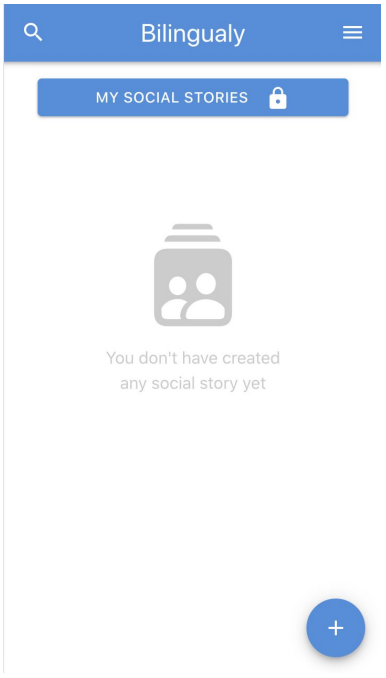


Figura --: Vista “My Stories” con lista vacía

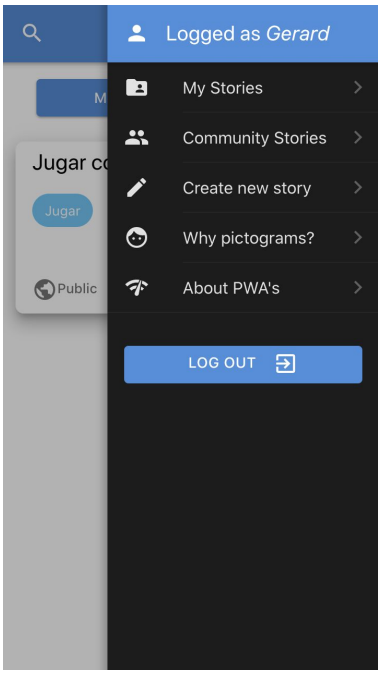


Figura --: Vista Sidebar

Finalmente, mostraré el proceso de creación de una historia social completa y qué ocurre si al momento de guardarlo en el servidor no tenemos conexión. Además, mostrará como cuando vuelve a estar online se notificada que se ha enviado satisfactoriamente.

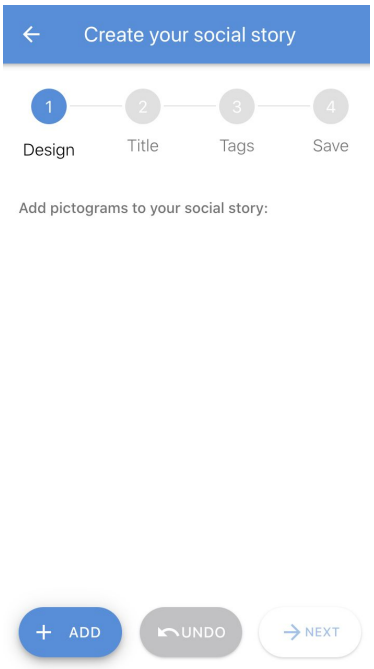


Figura --: Vista creación paso 1

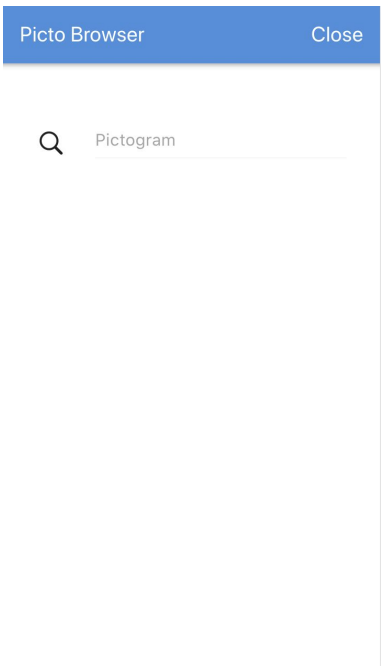


Figura --: Vista buscador de pictogramas

Figura --: Vista buscador con resultados

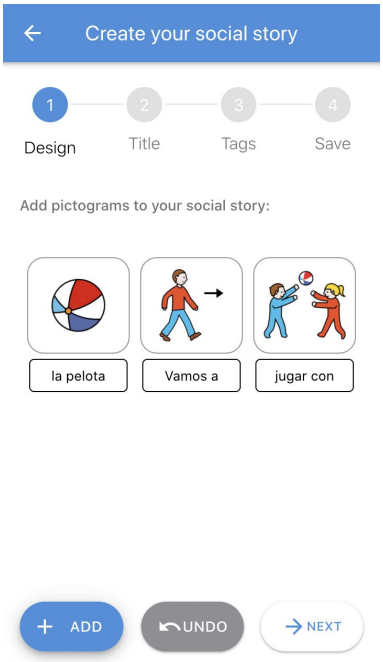


Figura --: Vista creación paso 1 desordenada

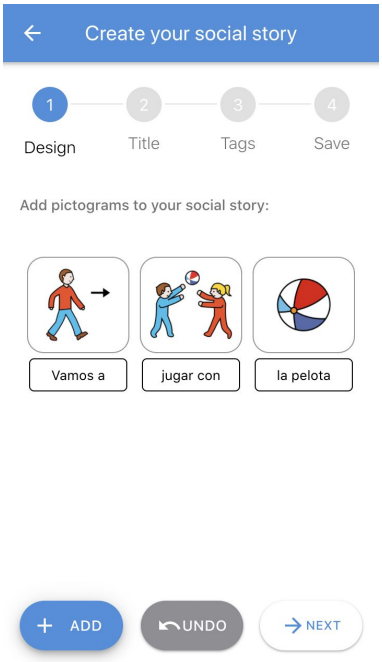


Figura --: Vista creación paso 1 ordenada

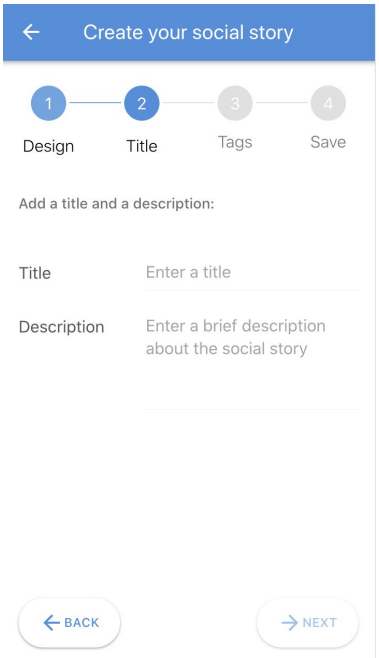


Figura --: Vista creación paso 2

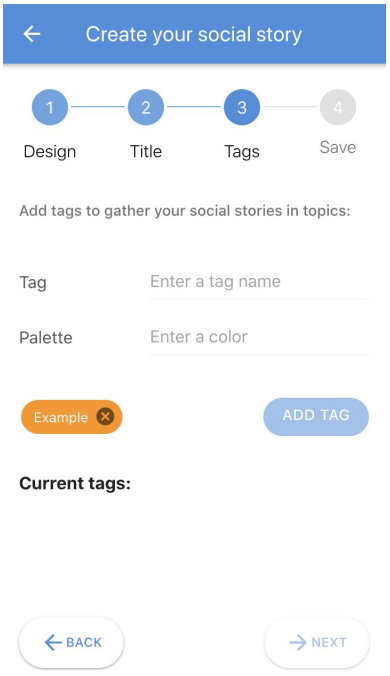


Figura --: Vista creación paso 3

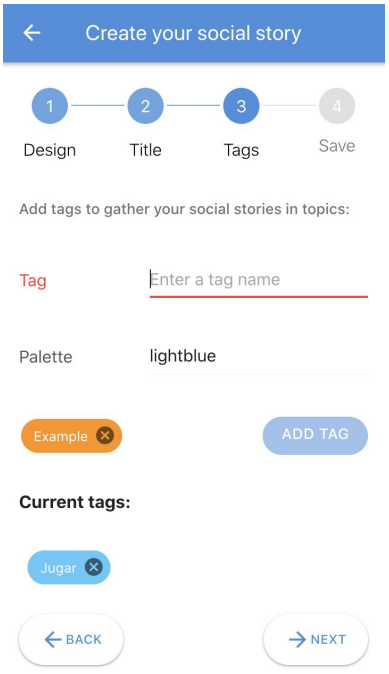


Figura --: Vista creación paso 3 con una etiqueta

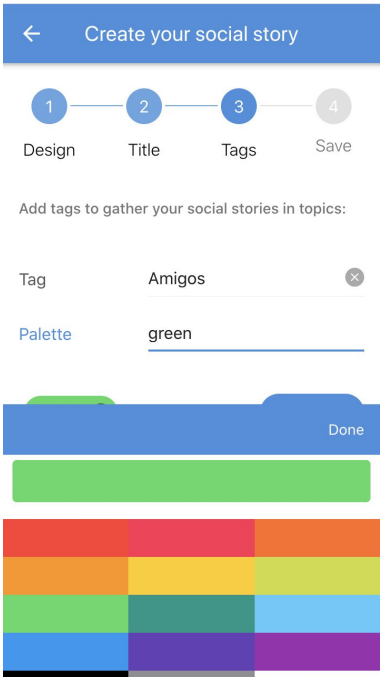


Figura --: Vista creación paso 3 escogiendo color

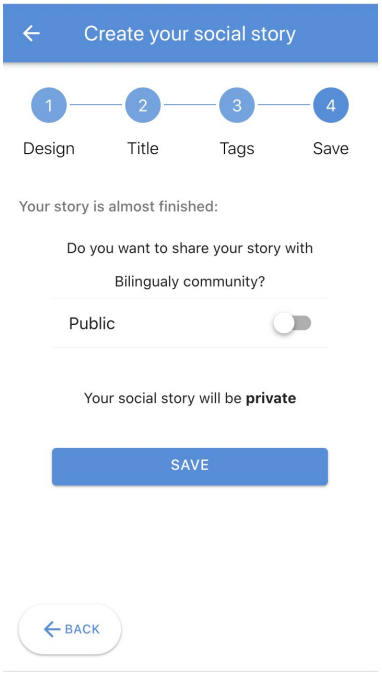


Figura --: Vista creación paso 4

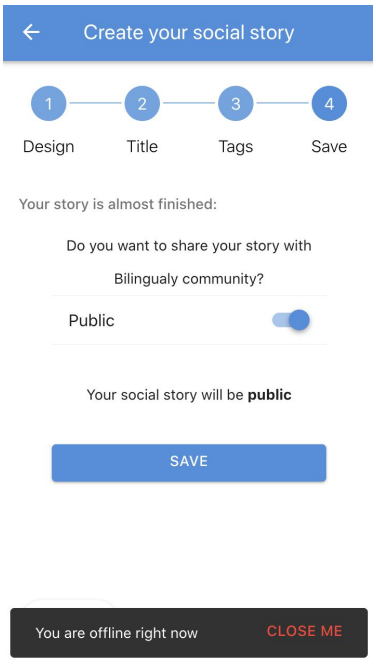


Figura --: Vista creación paso 4 sin conexión

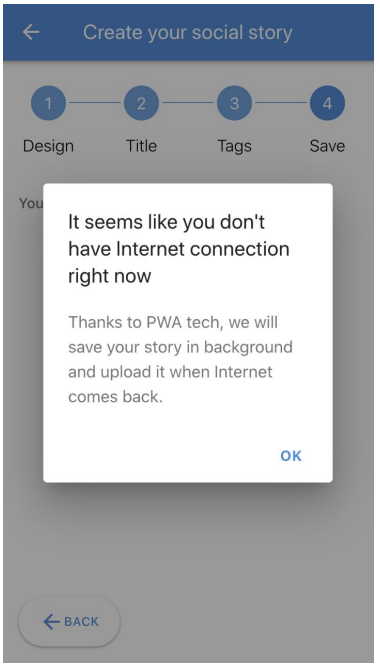


Figura --: Vista creación notificación sincronización segundo plano

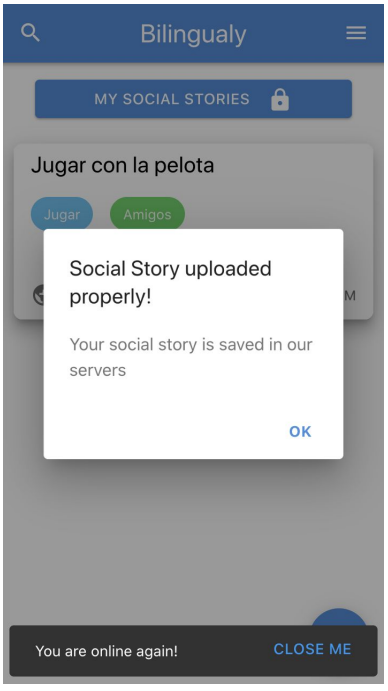


Figura --: Vista “My Stories” vuelve la conexión

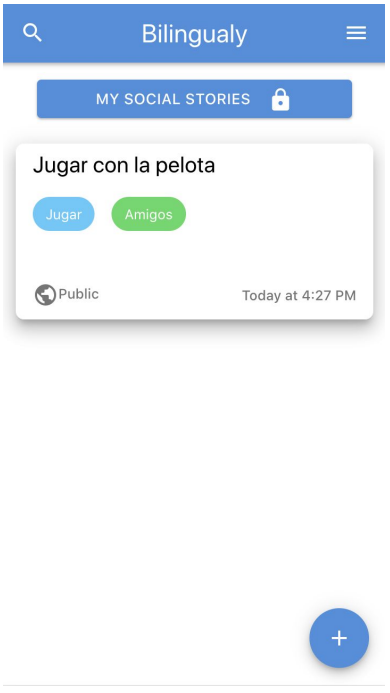


Figura --: Vista “My Stories” historia social disponible

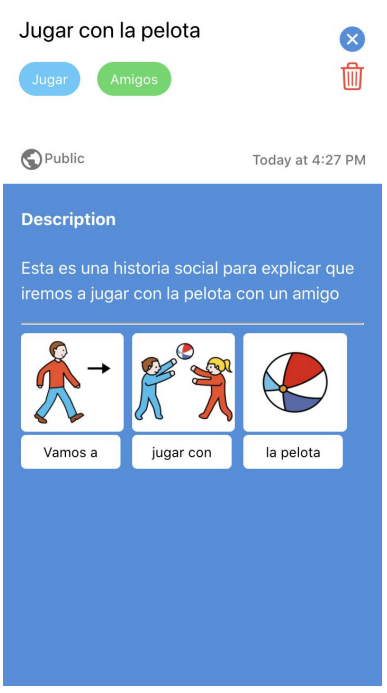


Figura --: Vista historia social completa

Glosario

PWA: Progressive Web App

CERN: European Organization for Nuclear Research

HTML: Hypertext Markup Language

JS: Javascript

CSS: Cascade Style Sheets

IDE: Entorno de Desarrollo Integrado

UI: User Interface

BaaS: Backend as a Service

DB: Base de datos

API: Application Protocol Interface

URL: Uniform Resource Locator

HTTPS: Hypertext Transfer Protocol Secure

ES6: ECMAScrip 6

JSON: Javascript Object Notation

CDN: Content Delivery Network