UNIVERSIDAD NACIONAL AUTONOMA
DE MÉXICO

FACULTAD DE ESTUDIOS SUPERIORES ARAGÓN

APLICACIONES WEB EMPLEANDO TECNOLOGÍAS DEL NAVEGADOR PARA NUEVAS EXPERIENCIAS DE USUARIO

T E S I S

QUE PARA OBTENER EL TÍTULO DE

INGENIERO EN COMPUTACIÓN

P R E S E N T A:

MARIO ABRAHAM OCHOA TOVAR

ASESOR:
ING. JOSÉ ANTONIO ÁVILA GARCIA

Ciudad Nezahualcóyotl, Estado de México 2021

Índice General

INTRODUCCIÓN

Hoy en día la vida gira alrededor de las tecnologías para dispositivos móviles y web, ahora bien, el desarrollo web ha causado un gran impacto en la sociedad desde la salida del lenguaje Javascript, causando un gran impacto en los usuarios de cómo ven las páginas web dinámicas actualmente, al igual que los desarrolladores se enfrentan a varios retos para programar esas mismas páginas.

En los primero dos capítulos de esta investigación se explicaran las bases fundamentales del desarrollo web. En el primer capítulo se explican los aspectos generales en el desarrollo web, el desarrollo móvil y herramientas que ayudan a que el desarrollo sea más efectivo.

En el capítulo dos se definen los fundamentos del desarrollo web, esto es parte importante conocer para entender las etiquetas que se muestran en el código final.

A su vez en ese mismo capítulo se define uno de los lenguajes de programación más importantes para el desarrollo de web. Se abarcan los fundamentos más importantes para entender el “como” funciona interactividad en los sitios y en las aplicaciones.

Posteriormente en el capítulo tres se abarcan los fundamentos de las aplicaciones web progresivas, en donde se mencionan algunas de las estrategias más usadas para almacenar datos de manera eficiente.

Finalmente en el capítulo cuatro se pondrá en práctica todos los fundamentos mencionados anteriormente para desarrollar una sencilla aplicación que tendrá como características el ser rápida, instalable y no dependerá de la conectividad a internet.

Con el paso del tiempo, el desarrollar páginas web ha ido evolucionando a medida de las necesidades del usuario, una de esas necesidades el optar por usar únicamente un dispositivo móvil por la internet, haciendo uso de WiFi o datos móviles. Esto lleva a un problema, y es que el uso de esos datos móviles o la red en que esté conectado el usuario ocasione una ralentización de carga para el sitio web y el usuario no pueda acceder a él.

Un reciente reporte de *Think with Google* afirma que el tiempo promedio que le toma cargar completamente a una página web es de 22 segundos, y si esto es poco, Francisco Siutti, CEO and Founder de Nodus Company afirma que el 53% de los usuarios abandonan un sitio web si tarda más de 3 segundos, y una vez cargado el sitio los usuarios esperan que el sitio sea rápido, sin desplazamiento irregular ni interfaces de respuesta lenta.

EL problema que hoy existe es la falta de conocimientos en las nuevas herramientas que ofrecen los navegadores para poder desarrollar aplicaciones web, dando lugar a que los usuarios tengan que depender de aplicaciones nativas y tener buena conexión a internet para que su experiencia sea la mejor.

Aunque hoy en día existen herramientas que prometen ser una solución, sin embargo estas son bastantes lentas y el sobre ocuparlos llega a ser una pésima idea.

Por lo que en el año 2015 surge un concepto que lleva el nombre de “Aplicaciones web progresivas”, esto cubre una serie de estándares que aseguran de convertirla en una página que de la mejor experiencia e interacción a los usuarios.

EL objetivo es conocer las herramientas las herramientas necesarias para poder desarrollar una aplicación web cumpliendo con los siguientes puntos:

- Que se pueda usar sin internet y no dependa de una conexión rápida y estable para su funcionamiento.
- Que sea rápida su carga y que responda casi inmediatamente a las interacciones con el usuario.

A su vez dar el conocimiento práctico y visual del código de una simple aplicación que funcione con los puntos mencionados anterior mente.

De esta manera surgió el interés de compartir los conocimientos necesarios, explicados con detalle sobre cómo llegar a poder desarrollar este tipo de aplicaciones.

Unidad 1 Aspectos generales en el desarrollo web y móvil

1.1 FUNCIONAMIENTO DE LAS PAGINAS WEB

Antes de empezar con el desarrollo de una aplicación progresiva primero se debe entender cómo funciona una página web, es decir los procesos internos se tienen que ejecutar al momento en que nosotros ingresamos alguna página ya sea para a YouTube, Outlook, Google.

Al momento de ingresar en alguna página suceden una serie de eventos, dividiéndolos en dos tipos, el *frontend* que es todo lo que está sucediendo a la persona que navega y el *backend* que es todo lo que sucede en el servidor en donde se encuentra para que la página funcione.

Cuando se ingresa en algún sitio web desde el ordenador se le llama el cliente y el equipo donde está alojada la página con todos los archivos y toda la base de datos donde está toda la información que tiene que enviar para poder visualizar la página se le llama el servidor.
El servidor contiene toda la información desde los usuarios y todos los archivos que hacen la función que se conecte y nos la pueda mostrar.

Es importante tener claro que, el *frontend* es todo lo que corre en el navegador, lo que visualmente se ve y el *backend* es lo que corre en el servidor.

Por ejemplo, las contraseñas nunca deben estar en el frontend, siempre deben estar *backend* de manera encriptada. Al momento de escribir las claves en la parte del *frontend,* se envía esa información a un servidor y el servidor hace la comparación para dar el acceso a la cuenta.

En el funcionamiento de las páginas web se involucran distintas tecnologías, en pocas palabras se trata de dos operadores, un servidor y un cliente. El servidor se ejecuta continuamente en una computadora manteniéndose a la espera de peticiones de ejecución que le hará un cliente en internet.
El servidor web se encarga de contestar estas peticiones de forma adecuada entregando como resultado una página web u otro tipo de información, el cliente puede ser una computadora o un teléfono que están conectados a internet mediante el cual solicitan la información al servidor.

El funcionamiento de las páginas web es el siguiente:

- El cliente solicita una información a través de internet en este caso las páginas web y el servidor intercepta la petición y envía al cliente la información que le fue solicitada para ser visualizada en el navegador.

Todo esto consiste en la intervención de muchas tecnologías como las que se mencionan a continuación para que esto funcione de forma correcta

1. La primera de ellas son las DNS, cuyas iniciales son Domain Name System o sistema de nombres de dominio y así como nos lo mencionan en la web de *xatakamovil* "es una tecnología basada en una base de datos que sirve para resolver nombres en las redes, es decir, para conocer la dirección IP de la máquina donde está alojado el dominio al que queremos acceder."

   Esta nos permite controlar la configuración de correo electrónico y sitios web de nombre de dominio.

2. La segunda son las IP, son una etiqueta numérica que identifica de forma lógica a un dispositivo.

3. Los navegadores que son aplicaciones o programas que se utilizan para localizar o mostrar un sitio web.

4. Finalmente, son los lenguajes de programación utilizada para crear sitios web, que son principalmente HTML, CSS y Javascript.

1.2 TIPOS DE APLICACIONES

"Hoy en día, los dispositivos móviles, principalmente los teléfonos inteligentes y las tabletas, se han convertido en artefactos cotidianos en la vida de las personas. Según cifras mundiales, en el primer trimestre del presente año, se vendieron más de 418,6 millones de celulares, de los cuales el 51,6% fueron teléfonos inteligentes o "Smartphone" (Ruiz, 2013).

Hoy en día decenas persona interactúa con miles de aplicaciones que funcionan de diferente manera. Cada aplicación tiene una forma distinta de ejecutarse, por la razón de que existen diferentes tipos, tres de ellos son los siguientes:

- Aplicaciones nativas
- Aplicaciones web
- Aplicaciones híbridas

Como resultado cada uno de los tipos de aplicaciones debe pasar por un proceso de ejecución para que puedan funcionar en distintos sistemas operativos.

Ártica Navarro (2016) describe una aplicaciones móvil como “Un simplemente un programa informático creado para llevar a cabo o facilitar una tarea en un dispositivo informático. Cabe destacar que aunque todas las aplicaciones son programas, no todos los programas son aplicaciones” (p.2).

Florido Benítez (2017) explica que “Las aplicaciones móviles, tienen un software adaptado a un dispositivo móvil y que se integra como un instrumento más del Mobile marketing.”(p.11)1.2.1 Aplicaciones web

1.2.1 APLICACIONES WEB

Las aplicaciones web se refieren al uso de tecnologías web como *HTML, CSS y JS* para el desarrollo de estas aplicaciones. Por ejemplo si se quiere crear un sitio web no hay otra opción más que usar esas tecnologías. Entonces si desea crear una aplicación móvil a partir de los datos que tiene un sitio web se debe aprender un lenguaje nativo y esto suele ser complicado para algunas empresas.

Con el uso de las tecnologías web mencionadas se puede crear una aplicación móvil basada en el mismo lenguaje de las tecnologías web.
Esto es posible, ya que los dispositivos móviles cuentan con un navegador como Google, Chrome o Mozilla. Si ya se tiene una aplicación web y se quiere pasarla a móvil lo que hay que hacer es adaptarla a las diferentes pantallas de los dispositivos. A esto se le llama diseño web responsivo.

Lo que se logra con esto es que al final es tratar de convertir un sitio web o aplicación web en una aplicación móvil aunque al final siempre será una aplicación web.

Algunas de las ventajas que se tienen al desarrollar este tipo de aplicaciones son:

- Fáciles de crear y mantener.
- Económicas de desarrollar que las aplicaciones híbridas y nativas por la razón de que no hay que contratar a un grupo de desarrolladores para que la están haciendo.
- Son multiplataforma.

Desventajas:

- Se requiere de un navegador para ejecutarla.

- Son mucho más lentas que las app nativas dependiendo de que tanto pese nuestra app web determinara su rendimiento.
- Para el usuario son muy poco interactivas.
- No pueden ser enviadas a las tiendas de aplicaciones como la PlayStore de android o Mac Store de Apple.
- No puede interactuar con las utilidades del dispositivo. No podemos acceder al WiFi, bluetooth, GPS o cualquier funcionalidad nativa del dispositivo.

A Partir de los puntos anteriores se puede pensar que el desarrollar una aplicación web no es una buena opción, sin embargo, como se mencionó al principio del capítulo, el desarrollar una aplicación móvil para una plataforma requiere el aprender un lenguaje para cada uno.

En la web se ha estado extendiendo el uso de HTML, CSS Y JS y en realidad toda la web se basa en esos tres conceptos. Si pudiéramos llevar esos conceptos al navegador web de nuestros dispositivos móviles entonces nos ahorramos el trabajo y es lo que se ha estado buscando en estos últimos años y actualmente parece que se está logrando.

La empresa Mozilla desarrolló en 2013 una propuesta la cual propone un sistema operativo para los dispositivos móviles llamado “Firefox OS” que permite crear aplicaciones web usando los tres conceptos. Hoy en día ese proyecto quedó atrás por la razón de que poseía todas las desventajas que se mencionaron.

1.2.2 APLICACIONES NATIVAS

Arroyo (2011) introduce a las aplicaciones nativas el siguiente texto:

“Existe una gran controversia alrededor del tema de ¿cuál tipo de aplicación móvil desarrollar: aplicaciones móviles nativas, aplicaciones móviles web o aplicaciones móviles híbridas? Las respuestas son múltiples, sin embargo, para los efectos de este artículo, la decisión del por qué implementar aplicaciones móviles nativas y no aplicaciones web o aplicaciones híbridas, se fundamenta principalmente en el aprovechamiento óptimo de alguna de las funcionalidades de los dispositivos móviles, tales como el “GPS” (Global Positioning System), acelerómetro, captura de imágenes, audio y vídeo, entre otros, los cuales se implementan naturalmente mediante el uso de lenguajes de programación nativos para cada sistema operativo, según el tipo de dispositivo móvil. Esto no significa que con un lenguaje de programación web, tal como el HTML 5 o cualquier otro orientado al desarrollo de aplicaciones móviles web, no se pueda hacer uso de dichas funcionalidades, sino que el proceso de desarrollo de aplicaciones móviles web, mediante lenguajes web que aprovechen las

funcionalidades específicas de los dispositivos móviles, es sumamente complejo y en ocasiones implica un esfuerzo mayor que el de desarrollar una aplicación móvil nativa.”

También describe las aplicaciones móviles nativas como “pequeños programas que se instalan para ampliar las funcionalidades del dispositivo” (Arroyo, 2011)

Estas son el tipo de aplicaciones que la mayoría de las personas considera cuando quiere crear una aplicación desde cero o básicamente cuando quiere empezar a crear aplicaciones.

Son el tipo de aplicación más común en los dispositivos móviles, podemos encontrarlas fácilmente de la tienda de nuestro dispositivo ya sea Android o IOS.

Para el desarrollo de estas aplicaciones en necesario aprender un lenguaje para cada plataforma. Usan sus propios lenguajes proporcionados por cada empresa que está detrás de la plataforma como por ejemplo java, Swift, objetive-c

Algunas de las ventajas que se tiene con este tipo de aplicaciones son:

- Son bastantes rápidas debido a que el lenguaje en que son desarrolladas está enfocada en esa plataforma
- Pueden ser distribuidas en las tiendas de aplicaciones móviles porque en realidad está todo de acuerdo a la regla que da la empresa. El uso de su entorno de desarrollo y su lenguaje de programación que da la empresa para construir en su plataforma son aceptadas en la tienda de aplicativos móviles
- Son mucho más interactivas e intuitivas para el usuario.
- Pueden interactuar con las utilidades del dispositivo, por ejemplo los sensores, la cámara o el WiFi y así el aprovechamiento del dispositivo es del 100

Desventajas

- Son creadas para ejecutarse en una plataforma específica.
- Los lenguajes de cada una de las plataformas son demasiados complicados debido a que ya se han venido desarrollando desde hace ya muchos años.
- Tienen herramientas muy pesadas las llamadas IDE o entornos de desarrollo integrado que nos ofrecen las mismas empresas para el desarrollo de sus plataformas

- Son muy costosas para una empresa que desea crear aplicaciones para diferente plataforma debido a que se tiene que contratar a desarrolladores para cada una de ellas.
- Son difíciles de mantener por el hecho de tenerlas en dos plataformas diferentes

1.2.3 Aplicaciones híbridas

Ártica Navarra (2014) define en su tesis las aplicaciones híbridas como "Aplicaciones utilizan tecnologías web (HTML, Javascript y CSS) pero no son ejecutadas por un navegador. En su lugar, se ejecutan en un contenedor web (webview), como parte de una aplicación nativa, la cual está instalada en el dispositivo

Desde una aplicación híbrida es posible acceder a las capacidades del dispositivo, a través de diversas API. Las aplicaciones híbridas ofrecen grandes ventajas permitiendo la reutilización de código en las distintas plataformas, el acceso al hardware del dispositivo, y la distribución a través de las tiendas de aplicaciones. En contrapartida, se observan dos desventajas de las aplicaciones híbridas respecto del caso nativo: y) la experiencia de usuario se ve perjudicada al no utilizar componentes nativos en la interfaz, y ii) la ejecución se ve ralentizada por la carga asociada al contenedor web.

Afortunadamente, existe una diversidad de frameworks que permiten desarrollar aplicaciones híbridas" (p.11)

Este tipo de aplicación es una gran solución para algunos de los problemas a las anteriores tipos de aplicaciones.

Lo único que hacen este tipo de aplicaciones son utilizar *HTML, CSS y JS* pero haciendo uso de librerías o frameworks.
Los frameworks son un conjunto de código que con un objetivo en común, en este caso el desarrollo móvil.

Estas para poder ejecutarse necesitan un webview, el webview es un componente que tienen los dispositivos móviles que usan las aplicaciones híbridas que hace que luzcan como una aplicación nativa y no como una aplicación web aunque eso tiene ventajas y desventajas que se mencionan a continuación:

Ventajas:

- Son fáciles de construir. Así como las aplicaciones web, ya se tiene la aplicación creada y solo se tienen que adaptar a los dispositivos.
- Son baratas debido a que la aplicación ya está construida.
- Solo se necesita una aplicación para todas las plataformas.
- No necesitan un navegador para correr porque están utilizando un componente que simula un navegador.
- Pueden acceder a las utilidades del dispositivo usando APIs. Es decir pueden acceder a la cámara, GPS o el WiFi por medio de una serie de plugins llamados API's.
- Son más rápidas de desarrollar que las aplicaciones nativas.

Desventajas

- Son más lentas que las aplicaciones nativas porque al final se están ejecutando en un componente que simula un navegador.
- Son mucha más caras que las aplicaciones web porque se tiene que conocer o adaptar directamente los plugins o *API's*, es decir se requiere mucho mayor conocimiento.
- Son menos interactivas que las aplicaciones nativas.

Se han mostrado las ventajas que nos proveen las aplicaciones híbridas y web, pero en realidad lo que se requiere en que tengan el mismo rendimiento que las aplicaciones nativas utilizando solamente el lenguaje *Javascript*.

Hoy en día podemos usar el lenguaje Javascript para crear componentes nativos, es decir lo que escribimos en *HTML, CSS y JS* se convierta en componentes nativos facilitando el desarrollo y que al final se convierta en una aplicación nativa para que tenga todas las ventajas ya mencionadas de las aplicaciones nativas.

Toda la interfaz es utilizando básicamente la interfaz nativa pero a través de tecnología web facilitando el trabajo.

Lo que hay que considerar que actualmente no hay tantos *plugins* para interactuar con el dispositivo pero a medida que avance el tiempo básicamente están siendo creados y esto está aumentando muy rápido, así que probablemente vamos a encontrar muchos plugins dentro de poco tiempo porque en realidad tenemos empresas muy importantes que están utilizando este tipo de herramientas y las están haciendo evolucionar, por ejemplo tenemos a Facebook, Uber, Tesla, Airbnb.

1.3 DESARROLLO DE APLICACIONES MÓVILES

Antes de empezar a desarrollar aplicaciones móviles se debe considerar que no es nada sencillo, ya que cada aplicación móvil tiene un propio lenguaje de programación llamado lenguaje nativo.
Cada una de las empresas que están detrás de la plataforma proporciona un lenguaje de programación para desarrollar en esa plataforma.

Por ejemplo en los sistemas operativos Android tenemos por detrás a Google, dándonos herramientas como Java, Kotlin. Y si nosotros queremos crear una aplicaciones para esta plataforma necesitamos saber ese lenguaje de programación.

Para la plataforma IOS pasa el mismo, tenemos a Apple por detrás y debemos saber los lenguajes que nos otorgan como Objetive-C y Swift.

Y por último para crear en Windows tendríamos que saber .Net para su desarrollo

Cada uno de estos lenguajes es distinto uno de otro y es muy complejo el tener que aprender otro ya que empezamos desde cero aprendiendo un nuevo lenguaje solo con el fin de poder desarrollar en diferentes plataformas. Cada uno de esos lenguajes en realidad no es nada pequeños, tienen mucho tiempo desarrollándose y eso lo hace aún más complicado.

Es por esa razón que hay personas que solo se especializan en una sola plataforma

Hoy en día tenemos los tres tipos de aplicaciones que mencionamos en el punto anterior. Las aplicaciones Web, híbridas y Nativas

El crear una aplicación móvil es el sueño de muchos, la idea es mostrar los diferentes caminos donde básicamente puedes arrancar el prototipo de tu primera aplicación sin necesidad de programar y después el cómo la vas a ir programando.

Lo primero que hace uno para crear la App es diagramar el mockup de esa app. Se puede realizar de maneras muy sencillas desde dibujarle en papel o usar herramientas. Una de esas herramientas se llama balsamiq mockups, se puede encontrar entrando a su página https://balsamiq.com. La imagen 1 se muestra la página principal del sitio.

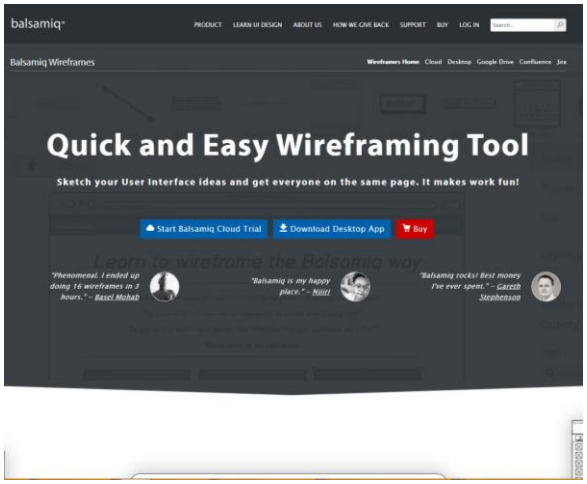

Imagen 1 Página principal *Balsamiq*
Fuente: https://balsamiq.com

Básicamente lo que uno puede hacer es realizar frames de la aplicación a desarrollar como se muestra en la siguiente imagen 1.

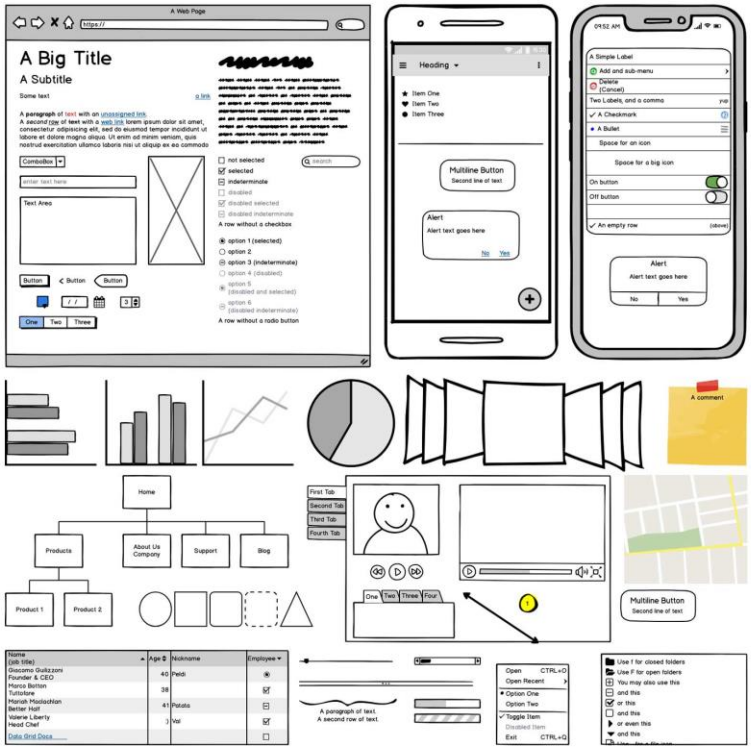

Imagen 1.2 Maquetado de una aplicación
Fuente: https://balsamiq.com

En la imagen superior da referencia a cómo se vería una aplicación en su versión web y en su versión móvil.

Algunas personas lo que hacen es usar *photoshop* o lo diseñan directamente, lo cual es una mala idea porque es muy importante probar la idea física en alguna versión de muy baja resolución antes de pasársela a un diseñador gráfico.

Si tenemos algún dispositivo *IOS*, ya sea un *Ipad* o un *Mac*, la empresa Apple nos ofrece un servicio de nombre Swift Playground. En la imagen 1.3 se muestra la página principal del sitio.

Imagen 1.3 Página principal Swift Playgrounds
Fuente: https://www.apple.com/swift/playgrounds/

Básicamente en *Swift Playgrounds* es un sistema para aprender el código nativo de las aplicaciones *IOS* en tu mismo dispositivo en donde prácticamente es arrastrar y soltar y nos permite crear versiones muy pequeñitas de cómo una aplicación funcionaria.

Otra de las herramientas que se pueden encontrar es *MIT App Inventor*, es básicamente un arrastrar y soltar de cómo crear aplicaciones móviles. Nosotros podemos elegir que queremos ejecutar para después exportar. En la imagen 1.4 se muestra el sitio principal

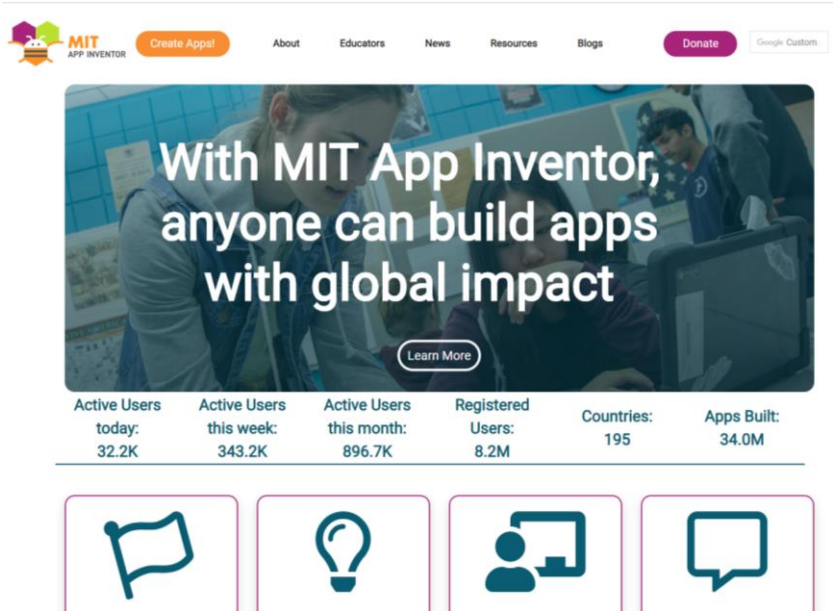

Imagen 1.4 Sitio principal de MIT App Inventor
Fuente: https://appinventor.mit.edu/

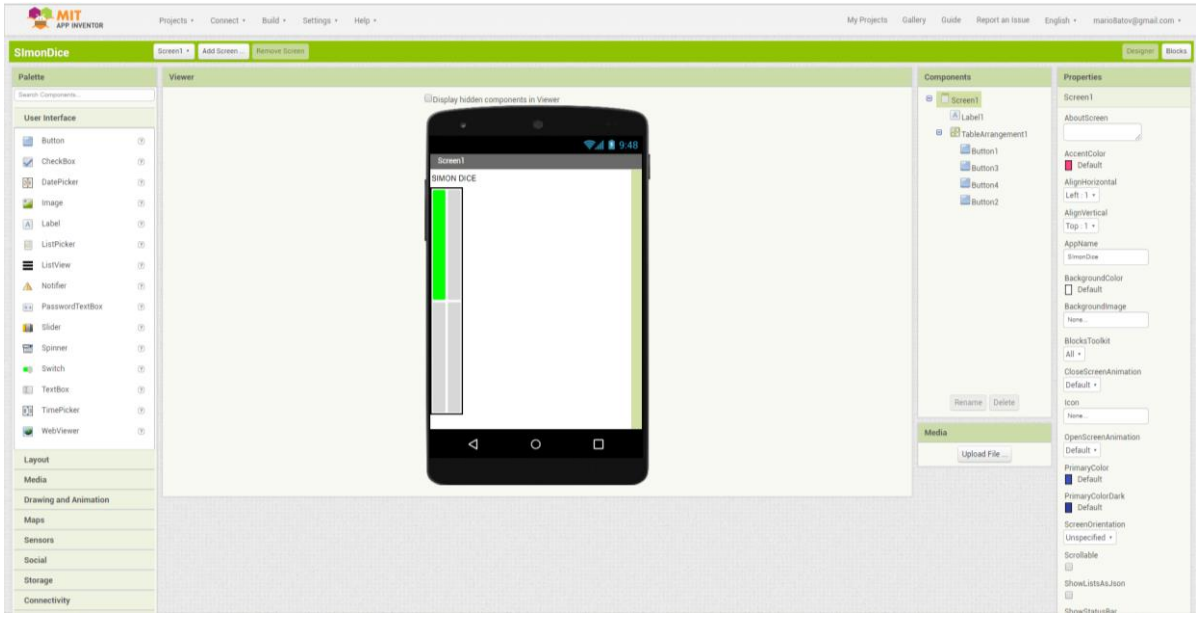

Imagen 1.4.1 Laboratorio de MIT App Inventor para desarrollo de la aplicación

Para llevar a cabo una aplicación profesional implica una serie de cosas, inicialmente incluye un Kit de ***UI*** (User Interface o Interfaz de usuario) o una aproximación de ***UI***. En la imagen 1.5 se muestra un ejemplo de la ***UI*** de diferentes pantallas de una aplicación

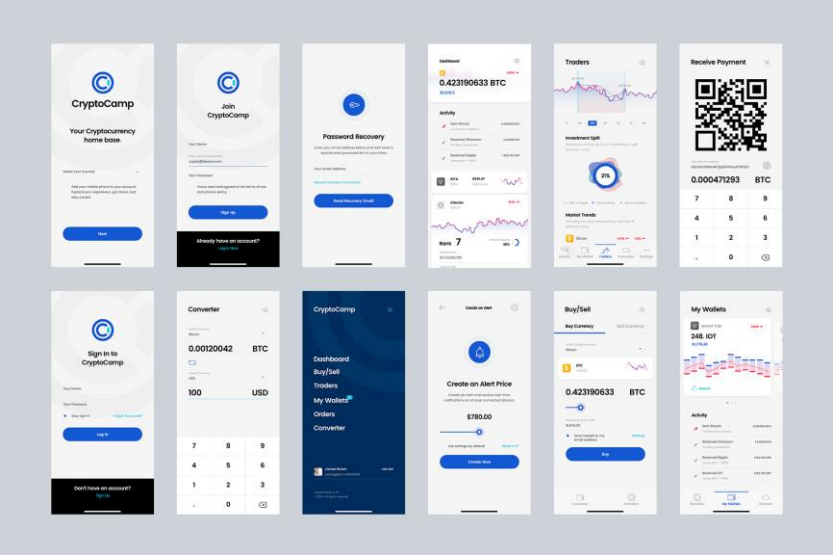

Imagen 1.5 Interfaz de una aplicación
Fuente: https://s3.envato.com/files/256146082/06_preview.jpg

Lo que se hace en el mundo profesional es que dentro de una aplicación de diseño gráfico es colocar cada una de las pantallas de nuestra aplicación y como están conectadas entre sí como se muestra en la imagen de arriba, esto normalmente se llama “Storyboard” de una aplicación.

Una de las herramientas más optimizadas para este tipo de diseño es *Adobe XD*, está enfocada al ***UI*** o diseño de experiencias de usuario. Podemos usar otras herramientas como photoshop o illustrator pero estas no están optimizadas para el prototipo de las aplicaciones móviles.

Otra herramienta optimizada para el diseño de aplicaciones es Sketch (https://www.sketch.com/), la ventaja de *Sketch* sobre *Adobe XD* es que podemos crear librerías de nuestros componentes que estarán en nuestra App como los botones y acciones y con simplemente exportarlas para poder ver el cambio de todas partes. La desventaja es que solo es compatible con equipos Mac, no está disponible para Windows o sistemas Linux.

A la hora de diseñar nuestra aplicación hay que ser sinceros con nosotros mismos, si de verdad no sabemos diseñar lo mejor que podemos hacer es contratar a alguien que sepa.

Una herramienta complementaria que podemos añadir una vez hecho el diseño es *Zeplin* (https://zeplin.io/). Lo que hace esta herramienta es agarrar el diseño gráfico y le agrega assets dividiéndolo por partes mostrándonos exactamente cuánto mide cada una de las pantallas, nos dice que partes es un título o una imagen. En ocasiones nos permite exportar a código nativo *swift* o *Java*.

Una vez que tengamos nuestro diseño o mockup de nuestras aplicaciones podemos empezar hablar sobre la parte técnica

.

La imagen 1.6, proporcionada por Platzi nos da un claro ejemplo de cómo funciona el ecosistema de desarrollo de aplicaciones móviles cuando uno empieza por crear App nativas.

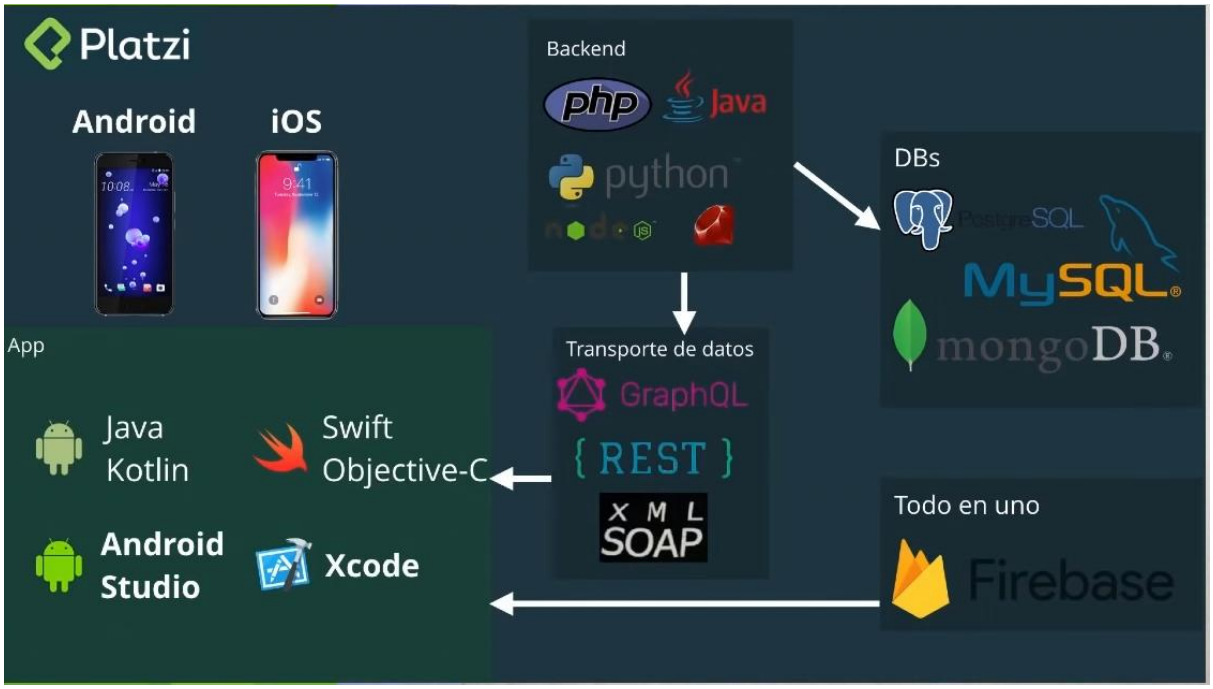

Imagen 1.6
Fuente: https://platzi.com/

Empezamos con la decisión para que plataforma queremos crear nuestras apps, ya sea Android o IOS y usar las herramientas que nos ofrecen como ya vimos anteriormente, para Android nos ofrecen Java o Kotlin con un IDE de desarrollo de nombre Android Studio para poder compilar nuestro código y para IOS elegimos Swift y Objetive-C que a la vez nos ofrecen un IDE de desarrollo para compilar el código de nombre Xcode.

Para que nuestra aplicación tenga comunicación con internet tenemos dos caminos.
El primer de ellos es usar un sistema todo en uno, uno de los más populares y de los pocos que hay se llama *firebase* de Google.
El segundo de ellos es usar una base de datos. Bases de datos como *Oracle*, *SQL Server*, *Postgres*, *MySQL* o usar una base de datos no relacional como *MongoDB*.

Una vez que decidamos qué servicio y que tipo de base de datos usar un lenguaje de *backend* para hacer la comunicación con la base de datos. Algunos ejemplos de lenguajes más usados para servicios de *backend* son *PHP, Java, Python, nodeJs, Ruby* entre otros.
Ese lenguaje de backend es el que va hacer la consulta a la base de datos para después enviar esos datos a nuestra aplicación a través de un *API*.

Una *API* es básicamente una estructura de datos que está claramente predecible con el objetivo de conectar el frontend con el backend. Existen muchas capas de transporte de datos, normalmente los servicios modernos usan *REST* que funcionan con objetos *JSON (Javascript Object Notation)* que es código parecido a Javascript pero con datos encapsulados que nuestra aplicación puede capturar como si fuera una variable abierta.
Otra capa de transporte de datos muy popular es GraphML, teniendo una ventaja sobre la anterior la cual podemos hacer consultas directas haciendo que nuestro backend sea menos pesado al procesar los datos.

Se puede ahorrar tiempo si solo se usa el servicio todo en uno de *Firebase*, pero tiene grandes desventajas. El servicio es caro, no se tiene control sobre la base de datos y escalabilidad. Es muy recomendable usar este servicio cuando queremos arrancar una aplicación desde cero y con el tiempo que nuestra App vaya escalando podremos migrar hacia otro servicio.

Aquí es donde entran en juego las herramientas para el desarrollo de aplicaciones móviles multiplataforma que consisten en crear App tanto para Android y IOS utilizando el mismo código.

El primer que ofrece Microsoft con *Xamarin* que permite crear aplicaciones móviles como para Android y IOS y siendo aplicaciones nativas, es decir que tendrá el mismo rendimiento que una aplicación desarrollada con *Java o Swift.*

Las ventajas que da Xamarin es que está integrado perfectamente con las herramientas de Microsoft, como por ejemplo con Visual Studio o la nube de Microsoft. *Xamarin* utiliza el lenguaje de programación *C#*, Al usar esta herramienta se tendrá que saber ese lenguaje. También nos ofrece acceso a *APIS* nativas, que nos ofrecen acceso a la cámara, GPS y otras utilidades de nube. En la imagen 1.7 se muestra el logo oficial de esta herramienta y en la imagen 1.7.1 se muestra el sitio oficial.

Imagen 1.7 Logotipo de Xamarin

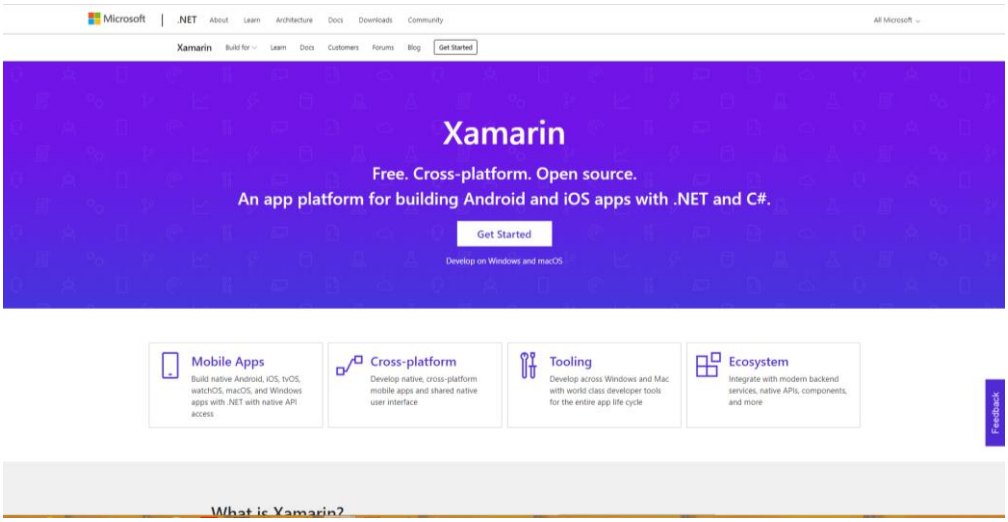

Imagen 1.7.1 Sitio principal de Xamarin
Fuente: https://dotnet.microsoft.com/apps/xamarin

Como segunda herramienta tenemos a *IONIC 4*.

Imagen 1.8 Logotipo de IONIC

Esta herramienta a diferencia de Xamarin no permite crear aplicaciones nativas, si no que nos permite crear aplicaciones híbridas, es decir será una aplicación móvil pero sin el mismo rendimiento de una App nativa.
Es una aplicación web que se comporta como una aplicación nativa, aun así es una buena solución para el desarrollo móvil.

Una de las ventajas de usar Ionic es que es bastante simple desarrollar en el si es que tenemos conocimientos sobre desarrollo web.

Ionic usa el concepto de los componentes para poder llevarlas a dispositivos móviles al igual como Xamarin tenemos acceso a distintas APIs nativas que nos permiten el acceso a la cámara y otras utilidades del dispositivo.

En su última actualización, ionic mete el concepto de PWA (Aplicación web progresiva) que otra solución para crear aplicaciones móviles.

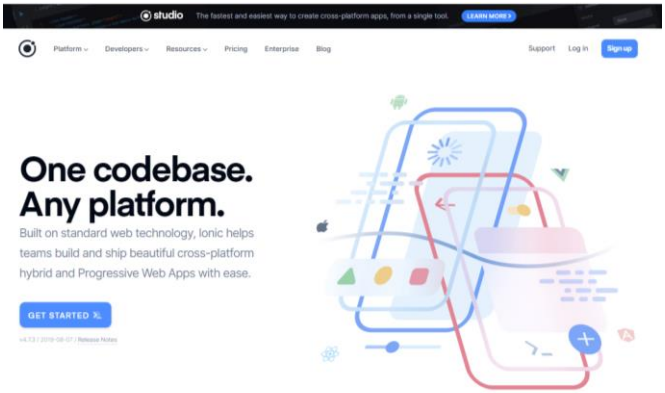

Imagen 1.8.1 Sitio oficial de IONIC
Fuente: https://ionicframework.com/

Como tercera herramientas tenemos React Native.

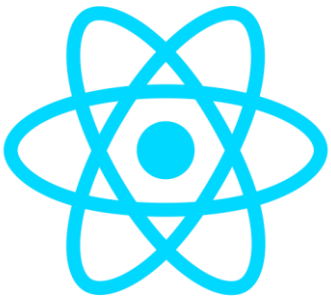

Imagen 1.9 Logotipo de react native

React native ha sido una de las mayores soluciones en los últimos años. Esta librería o framework nace de React, que es una biblioteca que Facebook creo para poder desarrollar interfaces.

Cuando hablamos de react hablamos de aplicaciones que ser comporta exactamente igual como una aplicación desarrollada en java o swift.

Esta solución es la más llevadera para poder crear una aplicación nativa usando Javascript, debido a que ya lleva varios años desarrollándose. Una de sus desventajas es que tenemos que conocer muy bien su librería que es React para poder usar React Native.

Imagen 1.9.1 Sitio oficial de react native
Fuente: https://facebook.github.io/react-native/

Como última solución es Flutter

1.10 Logotipo de Flutter

Flutter es una solución que nos ofrece Google para desarrollar una aplicación móvil nativa para Android y IOS usando una sola base de código, es decir el mismo concepto que las anteriores soluciones con la única diferencia de que flutter utiliza “Dart” como lenguaje de programación. Dart es un lenguaje no tan popular que Google trató de impulsar para el desarrollo móvil y hoy en día lo está logrando gracias a flutter.

Flutter es un SDK, es decir un conjunto de tecnologías que nos permiten crear aplicaciones móviles haciendo uso de un compilador y un conjunto de componentes.

Esta tecnología lo que hace es usar un motor de renderizado 2D para pintar los elementos en pantalla y personalización de la interfaz.

Unidad 2: FUNDAMENTOS DE PROGRAMACION WEB

2.1 LENGUAJE HTML, CSS y JAVASCRIPT

HTML, CSS y Javascript son los tres lenguajes principales para crear aplicaciones web. De estos tres lenguajes, Javascript es el principal de hacer la interactividad en los navegadores, por lo tanto se puede programar con diferentes librerías.

Asimismo, los navegadores contienen otros dos lenguajes, HTML y CSS, sin embargo no son lenguajes de programación.

Diego Gauchat (2012) describe el lenguaje HTML como "Una herramienta que con tres características: estructura, estilo y funcionalidad. Nunca fue declarado oficialmente pero, incluso cuando algunas APIs (Interface de Programación de Aplicaciones) y la especificación de CSS3 por completo no son parte del mismo, HTML5 es considerado el producto de la combinación de HTML, CSS y Javascript. Estas tecnologías son altamente dependientes y actúan como una sola unidad organizada bajo la especificación de HTML5. HTML está a cargo de la estructura, CSS presenta esa estructura y su contenido en la pantalla y Javascript hace el resto que es extremadamente significativo." (p.1)**, el gran libro de HTML. CSS y JavaScript. Juan Diego Gauchat. 2012**

HyperText Markup Language o por sus siglas en inglés *HTML* y por su traducción al español *Lenguaje de Marcas de Hipertexto*, es un lenguaje de etiquetado específicamente para el uso de desarrollo web

- *HTML* no solo es un lenguaje de etiquetado, también es llamado el archivo sobre el que se trabaja. Dentro de *HTML* interactúa con otros dos lenguajes, uno llamado CSS y otro que internamente se llama Javascript.

En él se define la información y el contenido del documento, es importante recordar que no es un lenguaje de programación, si no que dentro de *HTML* tendremos únicamente la información del sitio web.

- En *CSS* por su traducción en inglés *Cascade Style Sheet* u Hojas Estilos en Cascada se obtendrá el diseño de las páginas web.

El lenguaje que hace que todo sea interactivo es *Javascript*, este lenguaje hace la interactividad las páginas web.

Cuando se menciona interactividad es decir el hecho de el usuario haga algo con el computador y curra alguna acción.

2.2 HTML

Como ya se dijo en el punto anterior, el nombre de HTML viene de las siglas en inglés HyperText Markup Language, lo que significa que es un lenguaje que ayuda a interconectar textos.

También es llamado un lenguaje de etiquetas. Esas etiquetas son palabras clave que le dice al texto cuál es el título, el contenido, un párrafo y también indicarán las fotografías, tamaño y colores a utilizar en el sitio web.

Diego Gauchat menciona en su libro que “HTML5 no es una nueva versión del antiguo lenguaje de etiquetas, ni siquiera una mejora de esta ya antigua tecnología, sino un nuevo concepto para la construcción de sitios web y aplicaciones en una era que combina dispositivos móviles, computación en la nube y trabajos en red.
Página (Introducción)

2.2.1 ETIQUETAS ESTRUCTURALES

Una de las etiquetas más usadas para agrupar es *div* y por mucho tiempo era la única opción. Esto causaba problemas en la legibilidad del código, además de problemas en dónde cerrar y abrir nuevas etiquetas div.

Debido a esto, HTML5 introdujo nuevas etiquetas que imitaban el comportamiento de un *div*, ser un contenedor de otras etiquetas, pero adicionalmente tendría nombres que darían un significado semántico con solo leerlo.

Las etiquetas estructurales permiten la maquetación de una página web y se representan de la siguiente manera:

<etiqueta> …. </etiqueta>

Se forman por una etiqueta de apertura y otra de cierre. La etiqueta de apertura se identifica con el signo menor que y mayor que y entre ellas la palabra clave.

Por otro lado la etiqueta de cierre lleva el mismo formato con la diferencia que enseguida del signo menor lleva una diagonal, seguido de la palabra clave y cerrando con mayor qué, como se muestra en el siguiente código.

```
<!DOCTYPE html>
<html>
<head>
<title>Título del documento</title>
<link rel=”stylesheet” href=”misestilos.css”>
</head>
<body>Este es el cuerpo de la página<body>
</html>
```

En el código anterior se muestra que primero hay una etiqueta *DOCTYPE,* esta etiqueta indica que el formato que se maneja en el nuevo concepto de *HTML5.* A comparación con las demás etiquetas, está no requiere una etiqueta de cierre

Enseguida está la etiqueta de apertura de *HTML* seguida por *head* que indica la cabecera del sitio web.

Diego Gauchat (2012) indica recalca en su libro *el gran libro de HTML. CSS y Javascript que* “Los documentos HTML se encuentran estrictamente organizados. Cada parte del documento está diferenciada, declarada y determinada por etiquetas específicas”. (p.2)

Por ende Diego Gauchat (2012) escribe que “Dentro de las etiquetas <head> se define el título de nuestra página web, declararemos el set de caracteres correspondiente, proveeremos información general acerca del documento e incorporaremos los archivos externos con estilos, códigos Javascript o incluso imágenes necesarias para generar la página en la pantalla. El cuerpo representa la parte visible de todo documento y es especificado
entre etiquetas <body>.”(p.4)

De igual importancia está la <*title*> dentro de él se especifica el título del documento.

Por último la etiqueta <*body>,* como su traducción al español indica es el cuerpo de la página, dentro de ella estarán los títulos, subtítulos, párrafos, tablas y demás componentes que se le pueden agregar al sitio.

Finalmente Diego Gauchat (2012) comenta que “Otro importante elemento que va dentro de la cabecera del documento es <link>. Este elemento es usado para

incorporar estilos, códigos Javascript, imágenes o iconos desde archivos externos. Uno de los usos más comunes para <link> es la incorporación de archivos con estilos CSS.”(p.7)

En el código anterior cada etiqueta tiene un su etiqueta de cierre, por ejemplo si olvidáramos colocar su etiqueta de cierre en alguna, en el sitio web no se mostraría ese componente.

Con *HTML5* se tienen nuevas etiquetas que permiten manejar mejor la estructura y la organización del sitio web. La imagen que se muestra a continuación es una clásica representación de la organización de un sitio web

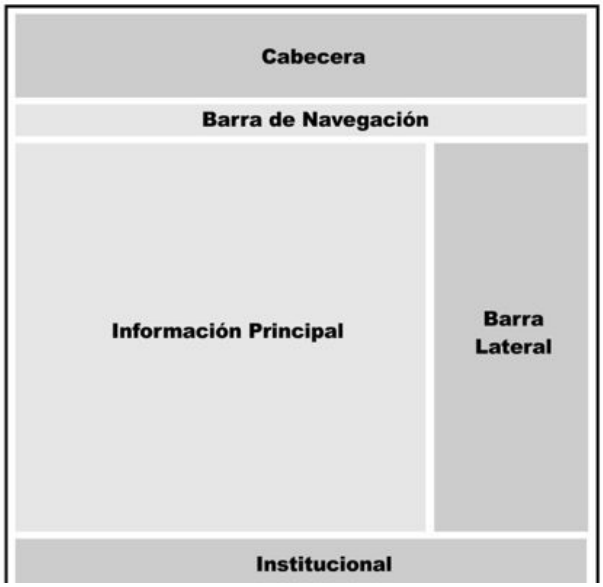

Imagen 2 Estructura de una página web
Fuente: El gran libro de HTML. CSS y Javascript
Diego Gauchat. 2012

Las nuevas etiquetas que nos da la versión de HTML son las siguientes:

1. Header
2. Section
3. Article
4. Aside
5. Footer
6. Nav

Como resultado de las nuevas etiquetas que se proporcionan podemos usar para ilustrar la organización con etiquetas *html5* como se muestra en la imagen 2.1

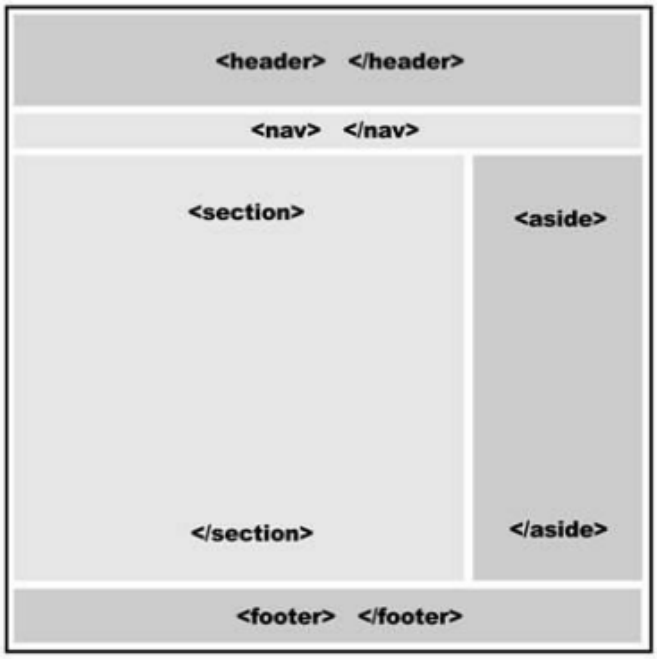

Imagen 2.1 Estructura de una página web con etiquetas
Fuente: El gran libro de HTML. CSS y Javascript
Diego Gauchat. 2012

2.2.1.1 HEADER

La etiqueta “Header” o encabezado son los elementos de introducción a la página o a una sección. Puede haber varios encabezados en una misma página o por lo general, puede tener elementos de navegación. Esta etiqueta generalmente está envolviendo a los títulos de nuestra página.

Diego Gauchat (2012) escribe que el elemento <head>“Tiene el propósito de proveer información acerca de todo el documento, <header> es usado solo para el cuerpo o secciones específicas dentro del cuerpo”(p.11), como se muestra a continuación.

```
<!DOCTYPE html>
<html>
<head>
<title>Título del documento</title>
</head>
<body>
        <header>
        <h1>Título de la página<h1>
```

```
        </header>
</body>
</html>
```

2.2.1.2 NAV

Nav es otra etiqueta estructural, dentro de ella se hace navegación interna de la página. Con navegación interna se refiere a que podremos pasar entre páginas del mismo sitio y no a páginas externas.

Generalmente dentro de las etiquetas *nav* vamos a poner una lista desordenada y con ayuda de los estilos le daremos el diseño de una barra de navegación.

Pese a lo anterior se define de la siguiente manera sin usar estilos

```
<!DOCTYPE html>
<html>
<head>
<title>Título del documento</title>
</head>
<body>
        <header>
        <h1>Título de la página<h1>
        </header>
        <nav>
                <ul>
                <li>Página 1</li>
                <li>Página 2</li>
                <li>Página 3</li>
                </ul>
        </nav>
</body>
</html>
```

2.2.1.3 SECTION

Esta etiqueta se usa para agrupar contenido que está relacionado. Dentro de ella se pueden meter etiquetas de títulos, párrafos, artículos y las etiquetas que se deseen.

Se define cómo <section>..<section> y se introduce en el código como se muestra a continuación

```
<!DOCTYPE html>
<html>
<head>
<title>Título del documento</title>
</head>
<body>
        <header>
        <h1>Título de la página<h1>
        </header>
        <nav>
                <ul>
                <li>Página 1</li>
                <li>Página 2</li>
                <li>Página 3</li>
                </ul>
        </nav>
        <section></section>
</body>
</html>
```

2.2.1.4 ARTICLE

Article o por su traducción al español artículo es una etiqueta que tiene un significado propio, es decir podemos quitar y ponerlo dentro de otra página sin perder el significado. Se le considera que debe utilizarse para definir contenido autónomo e independiente ya sea que provenga de otra fuente de información. Dentro de article podemos poner secciones, encabezados, pie de página, etc.

Siguiendo el código del ejemplo 4 podemos colocar la etiqueta article de la siguiente manera:

```
<!DOCTYPE html>
<html>
<head>
<title>Título del documento</title>
</head>
<body>
        <header>
```

```
        <h1>Título de la página<h1>
        </header>
        <nav>
                <ul>
                <li>Página 1</li>
                <li>Página 2</li>
                <li>Página 3</li>
                </ul>
        </nav>
        <section>
                <article></article>
        </section>
</body>
</html>
```

2.2.1.5 ASIDE

Aside como lo indica en su traducción al español “A un lado”.
Generalmente se usa la etiqueta aside para cosas que no estén relacionadas a la página.

Por ejemplo si quitamos un elemento y si la página no pierde significado, es un elemento adecuado para la etiqueta *aside*.

Es frecuente poner en la etiqueta aside alguna nota o publicidad que va alrededor del artículo.

Se define de la siguiente manera

```
<!DOCTYPE html>
<html>
<head>
<title>Título del documento</title>
</head>
<body>
        <header>
        <h1>Título de la página<h1>
        <header>
        <nav>
                <ul>
                <li>Página 1</li>
```

```
            <li>Página 2</li>
            <li>Página 3</li>
            </ul>
        </nav>
        <section>
            <article></article>
            <aside></aside>
        </section>
</body>
</html>
```

Diego Gauchat (2012) escribe que el elemento <aside> "podría estar ubicado del lado derecho o izquierdo de nuestra página de ejemplo, la etiqueta no tiene una posición predefinida. El elemento <aside>solo describe la información que contiene, no el lugar dentro de la estructura. Este elemento puede estar ubicado en cualquier parte del diseño y ser usado siempre y cuando su contenido no sea considerado como el contenido principal del documento. Por ejemplo, podemos usar <aside> dentro del elemento <section> o incluso insertado entre la información relevante" (p.4)

2.2.1.6 FOOTER

Footer es una de las etiquetas más sencillas a utilizar, por su traducción al español es el pie de página aunque también es llamada barra institucional, es utilizada normalmente para compartir información general sobre el autor o la compañía que ha creado el proyecto; como copyright, términos y condiciones, etc.

Normalmente, el elemento <footer> representa el fin del cuerpo de nuestro documento y tiene el propósito antes descrito. Sin embargo, la etiqueta <footer> puede ser usada varias veces dentro del cuerpo para representar el fin de las diferentes secciones. (La etiqueta <header> también puede ser utilizada varias veces dentro del cuerpo).

A continuación se muestra dónde colocar la etiqueta *footer*

```
<!DOCTYPE html>
<html>
<head>
```

```
<title>Título del documento</title>
</head>
<body>
        <header>
        <h1>Título de la página<h1>
        </header>
        <nav>
                <ul>
                <li>Página 1</li>
                <li>Página 2</li>
                <li>Página 3</li>
                </ul>
        </nav>
        <section>
                <article>
                        <h2>Titulo en article</h2>
                        <p>Párrafo en article</p>
                </article>
                <aside><p>Párrafo en aside</p></aside>
        </section>
        <footer>Pie de página</footer>
</body>
</html>
```

En la imagen 2.3 se muestra el resultado sin código *CSS*

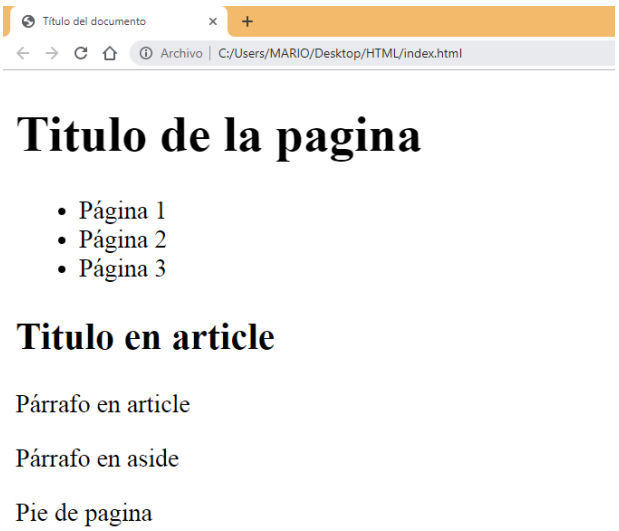

Imagen 2.3 Footer

2.2.3 ENCABEZADOS Y PÁRRAFOS

En todos los sitios web encontramos encabezados, estos son construidos dentro de los documentos *HTML.*

Los encabezados son los titulares o frases principales que ayudan a indicar cuál es el título de algún tema dentro de algún sitio. Suelen colocarse cómo cómo referencia al título en general y tener subtítulos si es que dentro del sitio hay información.

Existen seis etiquetas para dar referencia a los encabezados, sin embargo solo las primeras tres se usan para dar uso a los encabezados.

Se les nombra etiquetas *h* y cada una de ellas se diferencia por un número que indica el tamaño de fuente. Es otras palabras cuando se escribe la letra *h* debe ir seguida del número que indica el tamaño de fuente, así como se muestra a continuación.

<h1>Texto muy grande<h1>
<h2>Texto grande<h2>
<h3>Texto algo más grande de lo normal<h3>
<h4>Texto normal<h4>
<h5>Texto pequeño<h5>
<h6>Texto muy pequeño<h6>

Cada etiqueta *h* tiene una importancia diferente empezando por *<h1>.* Esta etiqueta indica el título principal de la página web y se especifica dentro de la etiqueta *<header>,* por ende el navegador lo entenderá que es título más importante

Con *<h2>* es de igual importancia en conjunto con la etiqueta anterior, con ella se logra identificar los títulos secundarios, a diferencia de la anterior el texto es grande pero poco más pequeño a la anterior etiqueta.

<h3> de la misma manera nos da como resultado un título pero más pequeño al anterior, es decir le va restando importancia conforme se indique que número de etiqueta *h* se requiera, como se muestra a continuación

<!DOCTYPE HTML>
<html>

```
<head>
<title>Título del documento</title>
</head>
<body>
        <header>
        <h1>Título de la página</h1>
        </header>
        <h2>Título de la página</h2>
        <h3>Título de la página</h3>
        <h4>Título de la página</h4>
        <h5>Título de la página</h5>
        <h6>Título de la página</h6>
</body>
</html>
```

La imagen 2.4 nos da como resultado lo siguiente en el navegador

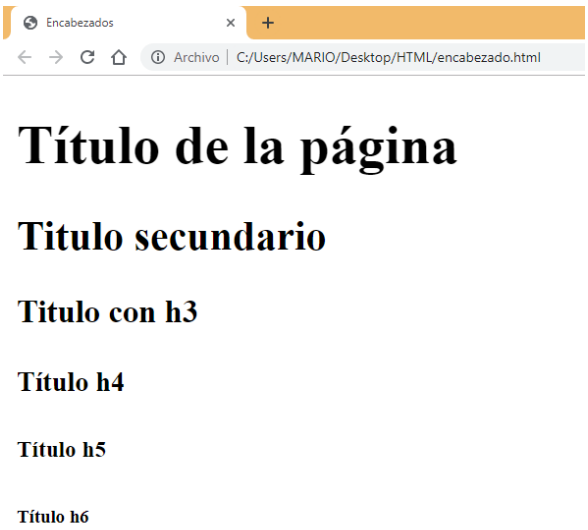

Imagen 2.4 Títulos en HTML

En particular el resultado que se muestra es el mismo, desde luego que el tamaño cambia consecuentemente por las etiquetas que se están especificando.

Es importante recalcar que *<h1>* es una de las etiquetas de encabezados más importantes pues es la que tiene mayor tamaño de fuente, por lo tanto se recomienda ponerla en el *<header>.*

Posteriormente de un subtítulo o título se encuentran los párrafos, en ellos se encuentran el resto de contenido textual.

Jorge Ferrer, Víctor García y Rodrigo García describen un párrafo en *HTML* como "Un conjunto de frases sobre un mismo asunto. En HTML para demarcar un párrafo se usa la etiqueta <p>, situándose la instrucción de inicio al comienzo del párrafo y la instrucción de fin tras la última frase. Entre ellas pueden insertarse tantos saltos de línea como se deseen así como muchos otros elementos HTML." (p. 21)

La etiqueta *<p>* se usa para escribir párrafos dentro del cuerpo de la página o la etiqueta <body>, de la siguiente forma

```
<!DOCTYPE html>
<html>
<head>
<title>Título del documento</title>
</head>
<body>
        <p>Esto es un párrafo, los párrafos son frases o contenidos formados por textos largos
        </p>
        <p> Este es otro párrafo </p>
</body>
</html>
```

Cada párrafo se encuentra diferenciado por un salto de línea por un salto de línea que lo hace en automático al terminar la etiqueta.

En *HTML* si se requiere de dar un salto de línea se puede usar otra etiqueta, la etiqueta
, en contraste si no se quiere colocar esa etiqueta se debe abrir una nueva etiqueta de párrafo como se muestra en el ejemplo 9. Se muestra que inmediatamente después de cerrar el primer etiqueta *<p>* se abrió una nueva, por lo tanto se nos mostrará un nuevo párrafo al abrir el navegador.

Finalmente se muestra en la siguiente imagen el resultado

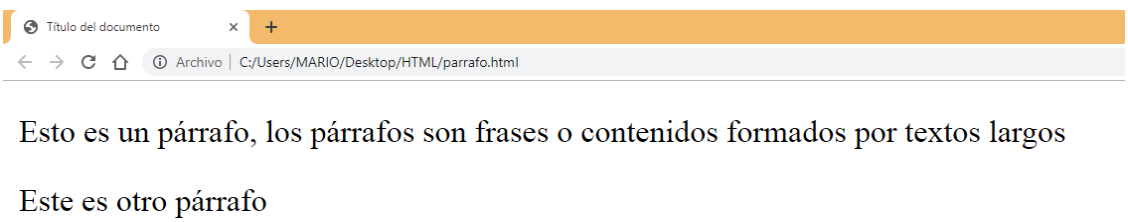

Imagen 2.5 Párrafos

2.2.4 FORMULARIOS

Los formularios son fundamentales al momento de recolectar información, de igual importancia al hacer registros para los visitantes del sitio. Esta información es formalmente enviada a un servidor para poder procesarla y gestionarla en una base de datos.

Un formulario en *HTML* es un contenedor para recolectar información dada por el usuario de distintas formas, en particular por líneas de texto o subida de archivos, pasando por opciones, fechas, contraseñas y mucho más.

Cómo ya se dijo, esa información recolectada por el formulario puede ser enviada a un servidor para procesarla y ser administrada.

Para abrir un formulario en *HTML* requiere de la etiqueta <*form*>*,* dentro de ella encerrara a un conjunto de controles.

Está etiqueta cuenta con varios atributos como *action*, que indica la ubicación del agente procesador; *method* que determina el método utilizado para empaquetar el formulario antes de ser enviado al agente procesador.

El conjunto de controles que están dentro de la etiqueta <*form*> se definen por otras etiquetas, una de ellas es <*input*>*.*

Con <*input*> se tendrá una caja de texto en donde se escribe la información, sin embargo esa misma etiqueta también nos da como resultado un botón.

Lo que identifica el input como un botón o una caja de texto es el atributo *type.* Cómo se aprecia a continuación, la etiqueta input tiene varios atributos

```
<!DOCTYPE html>
<html>
<head>
<title>Título del documento</title>
</head>
<body>
<form id=”form name=”form” method=”post” action=”form.php”>
        <input id=’email’ name=’email’ type=’text ’
        placeholder=”Introduce tu email”>
        <input  id=’button’ name=’submit’ type=’submit’
        value=’Suscribete’>
```

<form>
</body>
</html>

De los atributos empleados el primero es *id*, este atributo sirve como identificador de la etiqueta dentro del HTML y con el poder darle diseño.

El segundo atributo que se muestra es *name,* funciona como el valor que se identificara del lado del servidor.

El *placeholder* no se considera un atributo, es considerado una propiedad de la etiqueta <*input*> cuando son de tipo texto*,* en él se define el texto que aparecerá dentro del campo del formulario a modo de ayuda para las visitas.

El texto definido dentro de la propiedad *placeholder* aparecerá dentro del campo correspondiente con un tono gris clarito, indicando que no es realmente el texto que se va a enviar, sino lo comentado, una ayuda para informar de lo que debe contener ese campo.

Cómo ya se dijo, la etiqueta <*input*> también nos puede dar como resultado un botón, esto cambiando específicamente el atributo *type.*

Al tener el *type* en *submit* se tiene que especificar un valor o *value* como se muestra en el ejemplo 12 en la segunda etiqueta <*input*> que tiene el tipo *submit.* Ese valor será mostrado como texto en el botón de esta etiqueta.

Tanto una como la otra son etiquetas <*input*> diseñadas para usar en formulario dentro de las etiquetas <*form*>*.* Estas mismas pueden estar dentro de etiquetas de párrafos con el fin de poder escribir que campo es el que se debe llenar.

La imagen 2.6 es el resultado que nos da esta etiqueta usando el *type text* y *submit.*

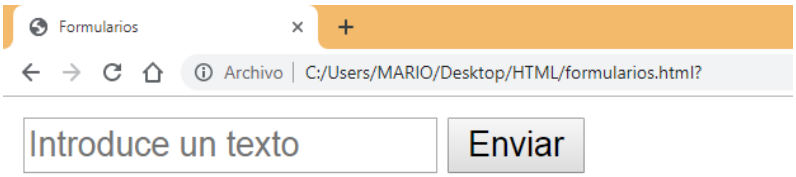

Imagen 2.6 Formulario

2.2.3.1 ATRIBUTO EMAIL

El atributo email permite que dentro de los formularios, específicamente dentro de las cajas de texto poder identificar una sintaxis de email. Está sintaxis está formada por un *arroba "@".*

De manera que se pueda identificar lo que es un email al llenar el campo de texto, por lo tanto si no se escribe la sintaxis adecuada para un email, este mismo nos enviara un error.

Para hacer uso del atributo se cambia el tipo dentro de la etiqueta input en sus atributos de la siguiente forma.

```
<!DOCTYPE html>
<html>
<head>
<title>Título del documento</title>
</head>
<body>
        <form id="form name="form" method="post" action="form.php">
        <input id='email' name='email' type=email'
        placeholder="Introduce tu email">
        <input  id='button' name='submit' type='submit'
        value='Suscribete'>
        <form>
</body>
</html>
```

La imagen 2.7 muestra el resultado del código anterior.

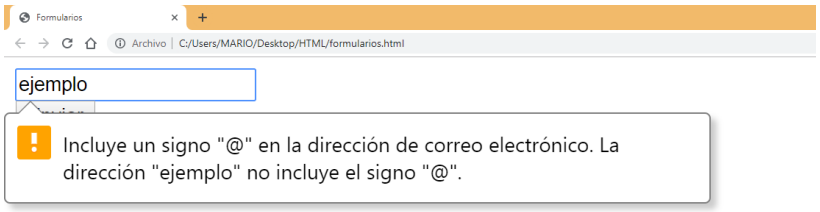

Imagen 2.7 Atributo email

2.2.3.2 ATRIBUTO URL

El atributo URL tiene como objetivo validar una dirección de internet. También representa un campo para un solo URL absoluto. El control asociado a este campo es una caja de texto que permite a los usuarios editar una sola línea de texto regular.

```
<!DOCTYPE html>
<html>
<head>
<title>Título del documento</title>
</head>
<body>
        <form id="form name="form" method="post" action="form.php">
        <input id='email' name='email' type=url placeholder="Introduce tu
        url">
        <input  id='button' name='submit' type='submit' value='Enviar'>
<form>
</body>
</html>
```

En la imagen 2.8 se muestra el resultado del código anterior.

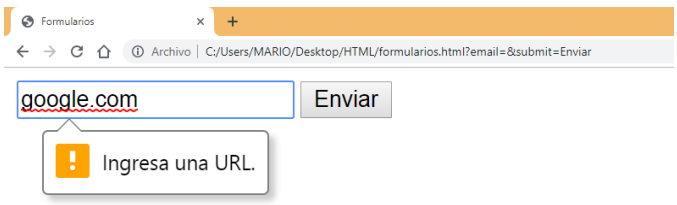

Imagen 2.8 Atributo URL

2.2.3.3 ATRIBUTO TEL

Este atributo ayuda a la validación en el campo de un teléfono, representa un campo para un número telefónico. El control asociado a este campo es una caja de texto que permite a los usuarios editar una sola línea de texto regular, sin requerimientos particulares sobre el formato. Esta conducta ha sido

adoptada debido a la gran variedad de número telefónicos válidos, que hace que las restricciones de propósitos generales sean inapropiadas. Se muestra a continuación la forma de hacerlo.

```
<form id=”form name=”form” method=”post” action=”form.php”>
<input id=’tel’ name=’tel’ type=tel placeholder=”Introduce numero celular”>
<input  id=’button’ name=’submit’ type=’submit’ value=’Enviar’>
<form>
```

El resultado del código se muestra en la imagen 2.9

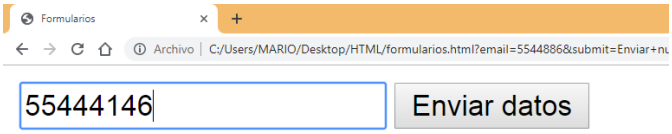

Imagen 2.9 Atributo TEL

Este atributo ayuda al usuario que se encuentra en un dispositivo móvil el mostrar solamente el teclado numérico.

2.2.3.4 ATRIBUTO NUMBER

Con esta etiqueta *number* ayuda a la recolección única de datos numéricos dentro del campo. Dentro de ella se pueden escribir números enteros y decimales. Dentro de la etiqueta podemos tener un valor máximo, un mínimo y un incremento.

De la misma manera tenemos la opción de poner un valor, por lo tanto si se tiene será el valor por defecto en la caja numérica. De la siguiente forma

```
<!DOCTYPE html>
<html>
<head>
<title>Título del documento</title>
</head>
<body>
        <form id=”form name=”form” method=”post” action=”form.php”>
```

```
        <input id='num' name='number' type=number max="15"
        min="40" value="18">
        <input  id='button' name='submit' type='submit' value='Enviar'>
        <form>
<form>
</body>
</html>
```

Se muestra en la imagen 2.10 el resultado del atributo number.

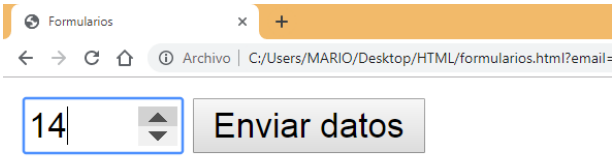

Imagen 2.10 Atributo number

2.2.3.5 ATRIBUTO RANGE

Una de las novedades que tenemos dentro de los formularios HTML5 son los elementos input de tipo rango. Estos elementos son unos sliders que nos permiten seleccionar un valor desplazando un puntero sobre el slider.

Lleva atributos de valor máximo y mínimo, junto con un valor que es opcional para dar por defecto. Las siguientes líneas de código se muestra el atributo range.

```
<!DOCTYPE html>
<html>
<head>
<title>Título del documento</title>
</head>
<body>
        <form id="form name="form" method="post" action="form.php">
        <input id='range' name='range' type=range max="15" min="40"
        value="18">
        <input  id='button' name='submit' type='submit' value='Enviar'>
        <form>
<form>
</body>
```

</html>

Dando como resultado la imagen 2.11 en el navegador

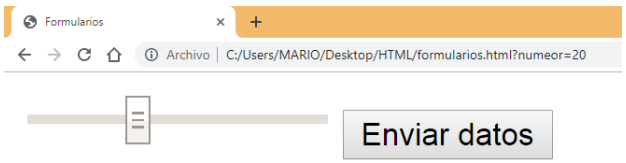

Imagen 2.11 Atributo Range

2.2.3.6 ATRIBUTO DATE

Este atributo permite crear campos de entrada que permite visualizar una fecha, bien a través de un cuadro de texto que valida automáticamente el contenido, o bien mediante una interfaz especial de selección de fechas. El valor resultante incluye el año, el mes y el día, pero no la hora. Los tipos de entrada time y datetime-local admiten entradas de hora y de hora y fecha.

En el siguiente código se muestra el atributo *Date.*

```
<!DOCTYPE html>
<html>
<head>
<title>Título del documento</title>
</head>
<body>
        <form id=”form name=”form” method=”post” action=”form.php”>
        <input id=’date’ name=date type=date max=”2018-12-31”
        min=”2018-01-01” value=”2018-07-22”>
        <input  id=’button’ name=’submit’ type=’submit’ value=’Enviar’>
        <form>
<form>
</body>
</html>
```

Esto da como resultado la imagen 2.12.

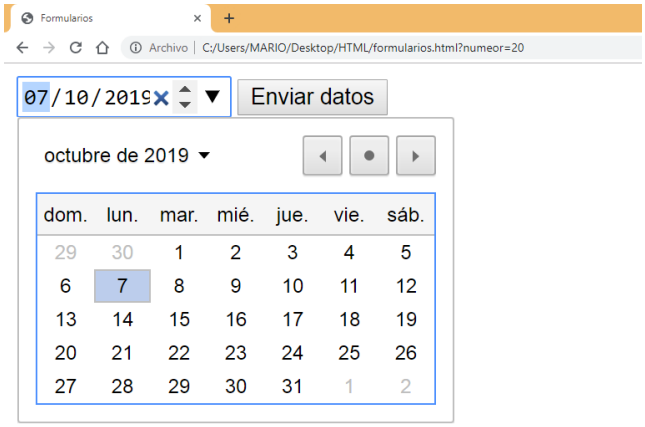

Imagen 2.12 Atributo Date

2.2.3.6 ATRIBUTO REQUIRED

Con el atributo *required* hace que el campo en donde se especifica sea obligatorio, es decir que no se puede dejar el campo vacío.

Permite comprobar que el campo ha sido rellenado antes incluso de pulsar ese botón de envío, sin necesidad de más complicaciones o código extra.

Si el usuario deja este campo en blanco, algunos navegadores arrojarán un mensaje de error, o simplemente colocan el cursor de escritura al primer campo vacío.

El código para el atributo *required* es el siguiente

```
<!DOCTYPE html>
<html>
<head>
<title>Título del documento</title>
</head>
<body>
        <form id="form name="form" method="post" action="form.php">
        <input id='text' name='text' type=text required" >
        <input  id='button' name='submit' type='submit' value='Enviar'>
        <form>
<form>
</body>
</html>
```

Cómo se aprecia en el código, *required* simplemente se indica en el campo que se desee usar como obligatorio. Si dejamos el campo el mismo navegador enviará una alerta de llenar el campo así como se muestra en la imagen 2.13

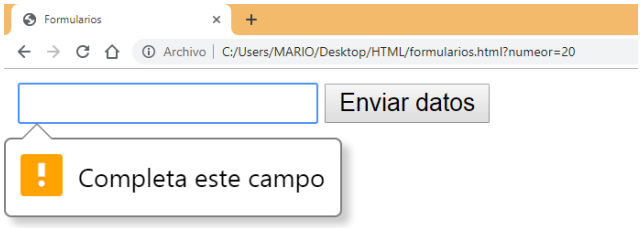

Imagen 2.13 Atributo Required

2.2.4 LISTAS

Este elemento es común encontrarlo en sitios web, su objetivo es colocar una serie de elementos organizados en listas o listados.

Se puede encontrar en menú ya sea que estén de manera horizontal como en la barra de navegación o de forma vertical.

Existen dos tipos de etiquetas para listados. La primera son listas ordenadas, y se caracterizan por tener un orden por número que aparece de forma automática, es decir que no es necesario escribirlo al hacer la lista. La etiqueta de este tipo de lista es *<ol>*

El segundo tipo a diferencia del primero, este no tiene un orden numérico, si no que se caracteriza por una rayita o un círculo redondo al inicio y su etiqueta es *<ul>.*

Para construir una lista se utilizan dos etiquetas, una indica que va comenzar una lista y otra que indicara que se va empezar a definir los elementos dentro de ella.

La etiqueta de apertura de una lista desordenada es <ul> y la de cierre es , y las etiquetas para especificar los elementos son *
*

Estas etiquetas se les llaman etiquetas de bloque, de manera que no es necesario hacer algún salto de línea.

Cómo ya se dijo, las listas mencionadas anteriormente se llaman “desordenadas” porque tienen un círculo negro a la izquierda de cada elemento que indica que no llevan ningún orden.

Las listas ordenadas se construyen de la misma manera, a diferencia de las desordenadas, estas llevan la etiqueta *
* con su cierre *
*

El código *HTML* se muestra abajo.

```
<!DOCTYPE html>
<html>
<head>
<title>Título del documento</title>
</head>
<body>
        <ul>
                <li>Primero<li>
                <li>Segundo<li>
                <li>Tercero<li>
        </ul>

        <ol>
                <li>Primero<li>
                <li>Segundo<li>
                <li>Tercero<li>
        </o>
<form>
</body>
</html>
```

En la imagen 2.14 se muestra el resultado del código listas.

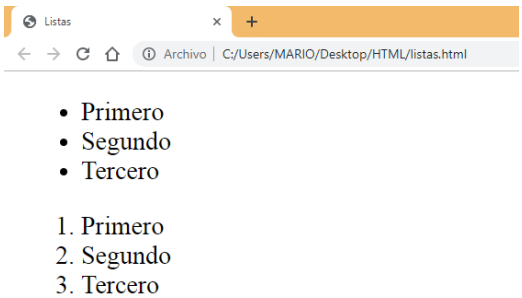

Imagen 2.14 Atributo listas

2.2.5 TABLAS

En documentos HTML una tabla puede ser considerada, resumidamente, como un grupo de filas donde cada una contiene a un grupo de celdas. Esto es conceptualmente distinto a un grupo de columnas que contiene a un grupo de filas, y esta diferencia tendrá un impacto en la composición y comportamiento de la tabla.

Como muchas otras estructuras de HTML, las tablas son construidas utilizando elementos. En particular, una tabla básica puede ser declarada usando tres elementos, a saber, *table* (el contenedor principal), *tr* (representando a las filas contenedoras de las celdas) y *td* (representando a las celdas)

Las etiquetas *<table></table>* son las encargadas de definir el principio y el final de una tabla respectivamente.

Cómo ya se dijo, utilizaremos las etiquetas *<td></td>* para inicio y final de cada una de las celdas y *<tr></tr>* para el inicio y final de cada fila. También podemos definir una cabecera en las tablas haciendo uso de la etiqueta *<th></th>*, en contenido de está aparecerá en negrita y centrado, como se muestra en el siguiente código.

```
<!DOCTYPE html>
<html>
<head>
<title>Título del documento</title>
</head>
<body>
    <table>
        <tr>
            <td>Celda 1</td>
            <td>Celda 2</td>
            <td>Celda 3</td>
        </tr>

        <tr>
            <td>Celda 4</td>
            <td>Celda 5</td>
            <td>Celda 6</td>
        </tr>
```

</table>
<form>
</body>
</html>

En la imagen 2.15 se ve el resultado código Tablas.

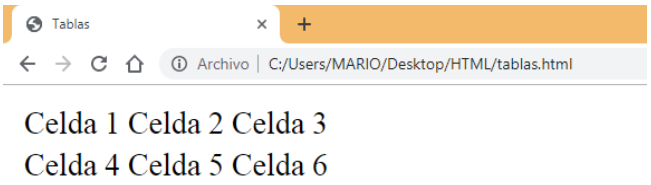

Imagen 2.15 Atributo tablas

2.2.6 IMÁGENES

Las imágenes son un elemento más en las páginas web y aunque adornan hay que añadirlos en el Html de las páginas, salvo algunas excepciones, como es el caso de las imágenes de fondo que sí que irán en el CSS.

Para insertar una imagen en un documento HTML se usa la etiqueta <img>, dentro de ella se especifican algunos atributos como la ruta. El ancho y alto de la imagen.

La ruta es la dirección donde se encuentra guardada la imagen, esta ruta puede cambiar dependiendo en donde se encuentre la imagen, ya sea que esté dentro de alguna carpeta o en la misma del documento. Está ruta se especifica en el atributo *src* .

Se pueden controlar las dimensiones de la imagen mediante dos atributos de la etiqueta *<img>,* estas son width que en español es *"altura"* y *"height",* en ellas se pueden dar valores en píxeles o porcentajes. A continuación se muestra el código para insertar imágenes en *HTML*

<!DOCTYPE html>
<html>
<head>

```
<title>Título del documento</title>
</head>
<body>
        <img src=”ruta/img.jpg” width=”100px” height=”200px”>
<form>
</body>
</html>
```

Cómo resultado obtendremos la imagen en nuestro navegador, tal y como se muestra en la imagen 2.16

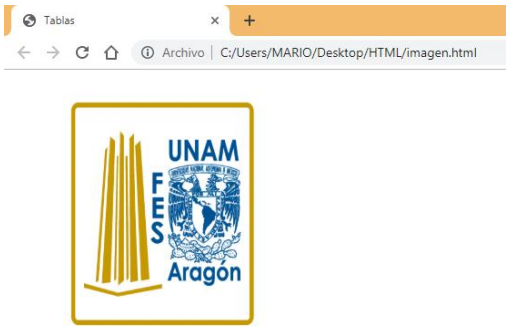

Imagen 2.16 Atributo imagen

2.2.7 ENLACES

Los enlaces, también llamados vínculos o links son textos que al dar click nos re direccionan a otra página.

La etiqueta usada para hacer enlaces en *HTML* es *<a>.* Está etiqueta por sí sola no hace nada, por lo tanto tiene propiedades.

En primera debe tener una ruta y esta se indica con la propiedad *href* seguida de la ruta entre comillas. Se pueden enlazar a otras páginas web como archivos de música, imágenes, etc. El código siguiente muestra la forma de hacerlo.

```
<!DOCTYPE html>
<html>
<head>
<title>Título del documento</title>
</head>
```

```
<body>
        <a href="https://www.googe.com">Google</a>
        <a href="noticias.html">Página Noticias</a>
<form>
</body>
</html>
```

La primera etiqueta de enlace se está dando como referencia al sitio web de Google, como resultado nos dirigirá al sitio en cuanto demos click en el.
En la segunda etiqueta de enlace no re direccionará a una página HTML que no está en la internet pero está en el los archivos del ordenador

El resultado del código se muestra en la siguiente imagen 2.17.

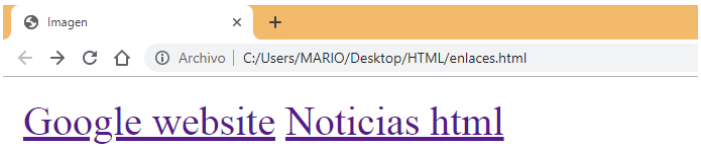

Imagen 2.17 Enlaces

2.3 JAVASCRIPT

Javascript es un lenguaje de programación dinámico, para Javier Eguiluz Pérez (2009) en su libro Introducción a Javascript lo define como “Un lenguaje de programación interpretado, por lo que no es necesario compilar los programas para ejecutarlos. En otras palabras, los programas escritos con JavaScript se pueden probar directamente en cualquier navegador sin necesidad de procesos intermedios” (p.11)

Este lenguaje es usado para agregar interactividad en las páginas web, esto quiere decir que se pueden agregar animaciones en cualquier sitio.

Javascript tiene sus orígenes en 1995, para ese entonces su nombre era *LiveScript* y poco tiempo después cambió a cómo se le conoce hoy en día, Javascript...

Javascript fue creado específicamente para el navegador *Netscape,* hoy en día se le conoce como *Mozilla.*

Este es lenguaje de tipado dinámico y a la vez interpretado. Esto quiere decir que a medida que se le dan instrucciones al ordenador, este mismo las va leyendo y seguido las va ejecutando.

Hoy en día JavaScript es utilizado en muchos lugares, ya no solo se utiliza para el *Frontend*, también puede ser utilizado para el *Backend*. Existe un estándar llamado *ECMAScript* el cual ha ido avanzando desde que inició y en el 2015 fue lanzado *ECMAScript 6* en donde vinieron cambios muy importantes en la sintaxis del lenguaje, cada año sale una nueva versión de *ECMAScript*, aunque después de ser sexta versión los cambios no han sido significativos.

2.3.1 SINTAXIS DEL LENGUAJE

El código de *Javascript* se puede encontrar en archivos separados al documento *HTML* sin embargo también se puede encontrar dentro del mismo dentro de las etiquetas ***<script>...</script>***. Dentro de esta etiqueta se escriben las sentencias del lenguaje, para el siguiente ejemplo se incluye una de ellas; ***console.log ()***

A continuación, se muestra un ejemplo de un script dentro de un documento *HTML*

```
<!DOCTYPE html>
<html>
<head>
<title>Javascript</title>
        <script type=”text/javascript”>
                console.log(“Escribiendo en consola”)
        <script>
</head>
<body>

</body>
</html>
```

El resultado del ejemplo anterior se muestra en la siguiente imagen 2.18:

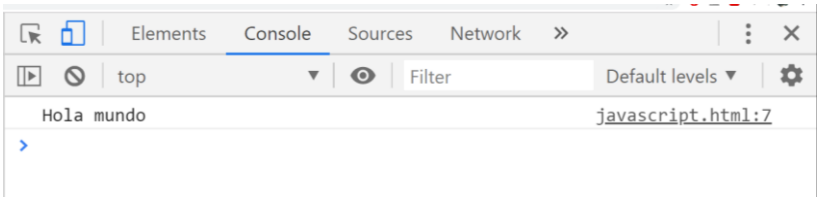

Imagen 2.18 Ejecución en consola

La instrucción ***console.log ()*** es una de las utilidades de Javascript la cual permite mostrar en consola algún mensaje que se especifique

El código *Javascript* debe ser escrito dentro de las etiquetas <*script*> para que la página sea válida, a diferencia si se escribe en un documento diferente, este mismo debe llevar la extensión *.es* indicando que es un archivo de Javascript.

La etiqueta <*script*> se puede incluir en cualquier parte del documento *HTML* aunque se recomienda poner justo en la cabecera <*head*>. Gonzalo Sánchez (2015) informa que "En primera todos los navegadores modernos (incluyendo internet Explorer 8) a partir del 2008 incorporaron un sistema llamado preload scanning que les permite cargar todos los scripts paralelamente sin causar bloqueos.

En segunda el agregar Javascript al inicio consiste en que a veces alguna funcionalidad crítica de la página depende de esos Javascript, por ejemplo si pones Google analytics al final, en el cierre del body y el usuario sale del sitio antes de terminar cargar la página entonces pierdes la data del usuario, otra de las funcionalidades críticas de Rails viene dada por jquery-ujs, si esto se cargará al final de la página tendríamos un problema con los usuarios que son del tipo Quicky Clicker y presionan el los links antes de que la página termine de cargar."

2.3.2 VARIABLES Y TIPOS DE DATO

En el lenguaje de Javascript existen diferentes maneras de definir variables y tipos de datos, Javier Eguiluz define una variable como "Un elemento que se emplea para almacenar y hacer referencia a otro valor."(p.16, 2009)

Las variables en *Javascript* deben tender dos características, un nombre de variable y un tipo de dato. El nombre de variable es importante por la razón de que el compilador del navegador reconoce de donde debe sacar la información, esto lleva el nombre de ***GET*** y donde pone esa misma información, está acción se le llama ***SET***.

El tipo de dato es el correspondiente al que lleva el nombre de la variable, es decir si el nombre de la variable es un número, su tipo de dato debe ser ***number.*** Si intentamos guardar una cadena de texto dentro de esa misma variable, el compilador por consecuencia enviará un error respecto a la variable.

En *Javascript* las variables se crean mediante la palabra reservada ***var,*** sin embargo en las últimas versiones de *ECMAScript* se pueden crear por las palabras reservadas ***let*** y ***const****.* Con *let* declaramos variables cuyo valor puede cambiar a lo largo del tiempo, sin embargo con la palabra ***const*** el cual su significado en español es constante, su valor no se puede cambiar. En los siguientes ejemplos se usará la palabra reservada ***var*** referenciando a uso de *Javascript* puro.

En el primer ejemplo se dos variables numéricas

Ejemplo 1

```
var numero_1 = 5;
var numero_2 = 3:
resultado = numero_1 + numero_2
```

Cómo se aprecia en el ejemplo, la variable se declara del lado izquierdo del signo igual y su valor del lado derecho del signo. En él se están declarando dos variables, la primera de nombre ***numero_1*** con valor *5* y la segunda variable con el nombre de ***numero_2*** con valor de 3. Sin embargo la variable ***resultado*** no está declarada, por lo que el lenguaje de *Javascript* crea una variable global y le asigna el valor correspondiente a las dos anteriores.

Javascript admite cualquier nombre para definir una variable, pero sigue una serie de reglas a considerar.

Los nombres de las variables no deben coincidir con las palabras reservadas y se deben diferenciar las mayúsculas y minúsculas. Se recomienda escribir en minúsculas o bien la primera mayúscula y las demás minúsculas o usando nomenclatura *CamelCase,* es decir si son dos palabras, la primera se escribirá en minúscula, posteriormente la segunda palabra iniciando en mayúscula.

Javascript maneja tres tipos de datos básicos que son: variables de cadena, numéricas y booleanas. También maneja dos tipos de variables compuestas las cuales son arreglos y objetos.

En conjunto manejan otros tipos de variables especiales que son ***null*** y ***undefined***. Los tipos de datos nulos (***null***) se utilizan para comprobar si una variable se le ha asignado un valor o no, representa un valor nulo para cualquier tipo de variable; por el contrario, si esa variable no ha sido inicializada tendrá un valor indefinido (***undefined***).

2.3.2.1 TIPOS DE DATOS NUMETICOS

Con *Javascript* se manejan números enteros y decimales y también números octales y hexadecimales.

A continuación se muestra cómo declarar este tipo de datos.

```
var edad = 23;
var calificacion = 8.8;
var negativo = -20;
var precio = 15.50;
```

Entre ellos se encuentran enteros positivos y negativos con los variables edad y negativo. Con variables flotantes se tiene precio y calificación.

2.3.2.2 TIPOS DE DATOS CADENA

A este tipo de dato también se le llama ***string*** o cadenas de texto y se utilizan para almacenar caracteres, palabras o frases de texto. Estos tipos de datos se encierran el valor dentro de comillas dobles o simples para delimitar su comienzo y su final como se muestra a continuación:

```
var saludar = “Hola mundo”
var texto = “Está es una cadena de texto en javascript”
```

Si se requiere encerrar en comillas algún carácter dentro de la misma cadena de caracteres lo que se hace es encerrar ese carácter con comillas simples. Si el propio texto contiene comillas simples o dobles, la estrategia que se sigue es la de encerrar el texto con las comillas (simples o dobles) que no utilice el texto:

```
var texto = “Está es una cadena ‘que’ contiene comillas dobles”
```

En este ejemplo la cadena de texto está encerrada en comillas dobles, y si se requiere encerrar algún carácter en comillas se deben usar las comillas simples, para el ejemplo, la palabra *contiene* es encerrado en comillas simples.
Por otro lado ocurre lo mismo si la cadena de texto está encerrada en comillas simples. El carácter que se requiera encerrar en comillas se deberá encerrar en comillas dobles, así como se muestra a continuación:

var texto2 = 'Texto "entre" comillas simples'

Existen otros caracteres que son difíciles de incluir en una variable de texto o cadena. Para esto, *Javascript* tiene un mecanismo para incluir caracteres especiales dentro de una cadena de texto. Este mecanismo consiste en sustituir el carácter por los siguientes caracteres:

Una nueva línea	\n
Tabulador	\t
Comilla simple	\'
Comilla doble	\"
Barra inclinada	\\

De esta manera, con el ejemplo anterior se puede rehacer de la siguiente manera

var texto2 = 'Texto \"entre"\ comillas simples'

2.3.2.3 TIPO DE DATO BOOLEANO

Este tipo de dato únicamente tiene dos valores, 1 (**uno**) y 0 (**cero**) o **true** (verdadero) y **false** (falso). No pueden almacenar números y tampoco cadenas de texto. Su funcionamiento es muy sencillo cómo se muestra a continuación:

```
var hayCambio = true
var sinCambio = false
```

La única regla que sigue este tipo de dato es que al declarar el tipo de dato se debe escribir en minúsculas, si el tipo de dato se declara en mayúsculas como **true** o **false** este mismo nos dará un error.

2.3.2.4 OPERADORES MATEMÁTICOS

Los operadores matemáticos son signos que expresan relaciones entre variables y constantes de las cuales se obtiene un resultado, se les conoce comúnmente como expresiones regulares. Los operadores más comunes se pueden encontrar los siguientes:

Suma: +
Resta: -
Multiplicación: *
División: /

En el siguiente ejemplo se puede ver su uso:

```
var suma = 5 + 2;
var resta = 45 - 5;
var multi = 5 * 5;
var dividir = 15 / 3;

var a = 5
var b = 7
var operación = a + b;
```

2.3.2.5 OPERADORES DE COMPARACIÓN

Este operador compara dos valores y determina la relación que existe entre ambos. En la siguiente tabla se muestran los operadores de comparación que existen en *Javascript.*

Tabla de operadores de comparación

Operador	**Descripción**
==	“igual a” devuelve true si los operadores son iguales
===	Estrictamente “igual a”
!=	“No igual a” devuelve true si los operados no son iguales
!==	Estrictamente “No igual a”
>	“Mayor que” devuelve true si el operador de la izquierda es ,Mayor que el de la derecha

>=	“Mayor o igual que” devuelve true si el operador de la izquierda es mayor o igual que el de la derecha.
<	“Menor que” devuelve true si el operador de la izquierda es menor que el de la derecha
<=	“Menor o igual que” devuelve true si el operador de la izquierda es menor o igual al de la derecha

Tabla 1 Operadores de comparación

Empleando la tabla anterior se comparan dos números de la siguiente forma:

```
5 == 5;
5 == 10;
10 >10;
5>=5;
5!=5;
5!=8;
5 !== 6;
```

El resultado de lo anterior se ve en la imagen 2.19:

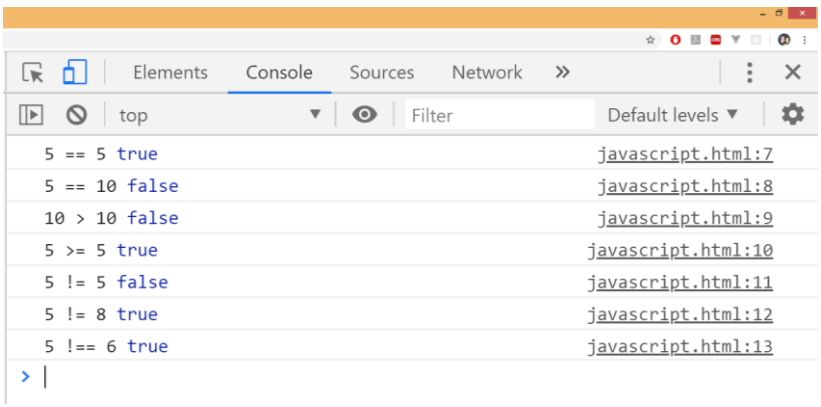

Imagen 2.19 Comparación de dos números

2.3.2.6 OPERADORES LOGICOS

Los operadores lógicos son usados con los operadores condicionales para estructurar expresiones complejas para la toma de decisiones. En *Javascript* existen dos operadores lógicos, AND representado mediante los símbolos *&&* el cual realiza una operación booleana y OR representada por los siguientes símbolos ||.

Las operaciones AND se representan en la siguiente tabla:

Tabla de operaciones AND (&&)

A	B	Resultado
true	true	true
true	false	false
false	true	false
false	false	false

Las operaciones OR se representan en la siguiente tabla:

Tabla de operaciones OR (||)

A	B	Resultado
true	true	true
true	false	true
false	true	true
false	false	false

Su representación en código de Javascript es:

```
var a = true;
var b = false;
resultado_1 =        a && b;
resultado_2 = a || b;
```

El resultado en consola se muestra en la imagen 2.20:

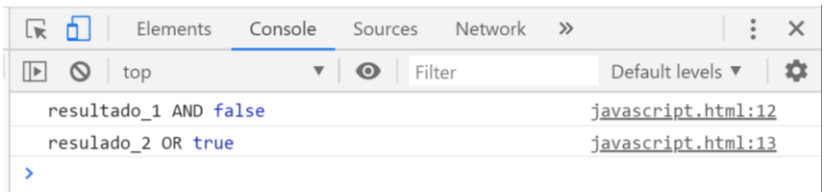

Imagen 2.20 Resultado de operaciones lógicas

2.3.3 SENTENCIAS CONDICIONALES

Las sentencias condicionales realizan instrucciones para evaluar resultados booleanos, una de ellas es la condicional ***IF***. Su definición es la siguiente:

```
if(condición) {
        //instrucciones
}
```

Si la condición se cumple, se ejecutan las instrucciones que se encuentran dentro de los corchetes de la condicional, sin embargo si la condición no se cumple no se ejecuta ninguna instrucción.

Existe otra segunda variante de nombre ***IF...ELSE***. Como ya se dijo la condición ***IF*** evalúa si la condición se cumple sin embargo al estar la condicional ***ELSE*** esta misma evalúa si la condición no se cumple y las ejecuta. Su definición es la siguiente:

```
if(condición) {
        //instrucciones
} else {
        //instrucciones
}
```

Con *Javascript* al igual que otros lenguajes de programación encontramos la sentencia condicional ***Switch***. Está sentencia únicamente evalúa una variable para dar posibles valores a esa misma y dentro de su estructura lleva una sentencia ***break*** que rompe con la estructura para continuar con el programa.
Su definición es la siguiente:

```
switch(condicion) {
        case "..":
                //instrucciones
        break;
        case "..":
                //instrucciones
        break;
}
```

2.3.4 CICLOS

Los ciclos o también llamados estructuras de control de flujo son instrucciones eficientes cuando se desea ejecutar una instrucción repetitiva.

Javascript maneja diferentes ciclos, entre ellos se encuentra el ciclo ***while, do while, for, for in***.

El primer ciclo ***while*** empieza con una condición inicial, posteriormente esa misma condición se evalúa y si la condición es nula o falsa el ciclo termina, de lo contrario la condición seguirá en un ciclo repetitivo hasta que no se cumpla.
Su representación se muestra a continuación:

```
while(condicion){
        //instrucciones
}
```

Para el ciclo ***do..While*** lleva de manera similar la lógica del ciclo mencionado anteriormente, este ciclo primero ejecuta la instrucción pasando por un bloque de sentencias para finalmente llegar a preguntar si se cumple la condición. En caso de que la condición sea verdadera el ciclo regresa al bloque de sentencias, en caso contrario el ciclo termina. Se declara de la siguiente manera:

```
do {
        //bloque de instrucciones
} while {
        ///bloque condicional
}
```

El ciclo ***for*** es una condición repetitiva mientras la condición sea válida. Permite realizar repeticiones de una forma sencilla, pese a lo dicho, su definición no es tan simple como los anteriores ciclos.

Es recomendable ser usado cuando se sabe hasta qué valor se quiere llegar y está formado por un valor iniciar, una condición que controla el ciclo y un incremento.

Para entender mejor los atributos que requiere este ciclo, se siguen los siguientes puntos:

- La "inicialización" establece los valores iniciales de las variables que controlan la repetición.

- La "condición" es el único elemento que decide si continúa o se detiene la repetición.
- El "incremento" es el nuevo valor que se asigna después de cada repetición a las variables que controlan la repetición.

A continuación se aprecia la sintaxis del ciclo for:

```
for(valor inicial; condición; incremento){
        //instrucciones
}
```

Para ilustrar el ejemplo anterior se tiene el siguiente:

```
for(var i = 0; i < 5; i++){
        console.log(var)
}
```

La parte de inicialización del ejemplo anterior se crea la variable *i* y se le asigna el valor de cero (0). La inicialización solamente se tiene en consideración justo antes de comenzar a ejecutar el bucle. Las siguientes repeticiones no tienen en cuenta esta parte de inicialización.

Lo que nos indica el bucle es que mientras la variable *i* valga menos de 5 el bucle se ejecuta indefinidamente.
La variable ***i*** se ha inicializado a un valor de cero (0) y la condición para salir del bucle es que ***i*** sea menor que 5, si no se modifica el valor de ***i*** de alguna forma, el bucle se repetirá indefinidamente.

Por ese motivo, es imprescindible indicar la zona de incremento, en este caso con ***i++***. En este caso la variable se incrementa en una unidad después de cada repetición.

El resultado se muestra en la imagen 2.21:

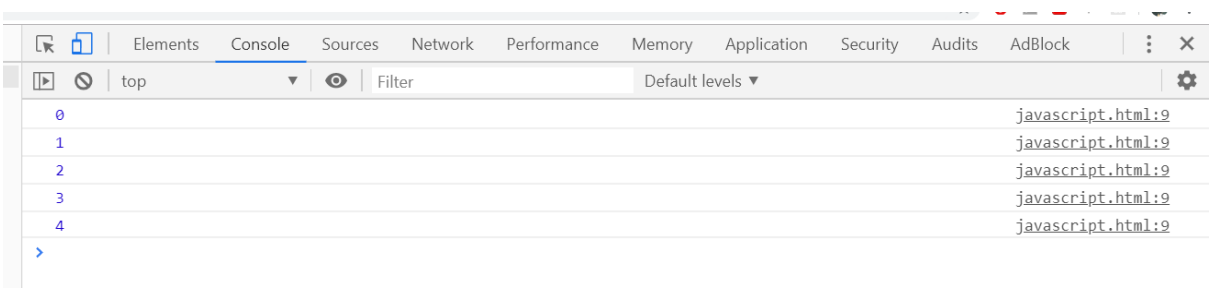

Imagen 2.21 Ciclo for

2.3.5 FUNCIONES

Las funciones son parte fundamental en el lenguaje de *Javascript* en particular porque permiten escribir código que se reutiliza múltiples veces. Dentro de una función se declaran parámetros que son variables solo funcionaran dentro de ella.

Son identificadas por la palabra reservada ***function***, seguido de la palabra reservada se indican paréntesis en donde se declaran los parámetros o atributos que obtendrá dicha función; seguido de las instrucciones o sentencias que estarán entre llaves.

Teniendo las instrucciones generada, opcionalmente regresará un valor que se le indicará con la palabra reservada return.

Para indicar el nombre a una función lleva las mismas reglas que el nombre de las variables en *Javascript*.

Su estructura es la siguiente:

```
function nombre() {
        //instrucciones o sentencias
}

nombre();
```

En la mayoría de las funciones se requiere pasar parámetros que serán recibidas por la misma. Si se requiere un parámetro en la función, se debe escribir dentro de los paréntesis y al llamar la función también se debe llamar el parámetro, así como se muestra a continuación:

```
function saludar(nombre) {
        console.log(nombre);
}
var nombre = “Mario Ochoa”;
saludar(nombre);
```

En el ejemplo anterior la función saludar está obteniendo un parámetro llamado “nombre”, está recibirá ese parámetro de cualquier tipo y lo mandará a imprimir en la consola.

Después de la función se declara una variable llamada nombre que será enviada a la función como un parámetro. Finalmente se llama la función saludar con el parámetro correspondiente. El resultado de la función se muestra en la imagen 2.22:

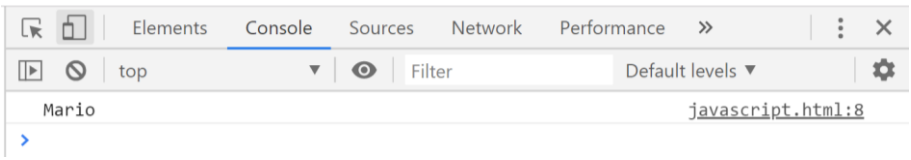

Imagen 2.22 Resultado de la función saludar

Enseguida se pueden añadir más parámetros a la función, por ejemplo:

```
function saludar(nombre, apellido) {
        console.log(nombre, apellido);
}
var nombre = “Mario”;
var apellido = “Ochoa”
saludar(nombre, apellido);
```

Finalmente, las funciones pueden regresar un valor con la palabra reservada **return** pero solo pasa o regresa un valor, es decir no pueden regresar dos o más valores. Aunque pueden regresar más valores como objetos o arreglos como se verá más adelante.

Usando la función del ejemplo anterior se obtendrá el valor de una variable dentro de la misma función, así como se muestra en el ejemplo inferior.

```
function saludar(nombre, apellido) {
        var nombreCompleto = “Hola” + nombre +
apellido;
        return nombreCompleto;
}
var nombre = “Mario”;
var apellido = “Ochoa”
saludar(nombre, apellido);
```

2.3.6 ARREGLOS Y OBJETOS

Julio Giampiere informa en su artículo que “Los arreglos son un conjunto de datos ordenados por posiciones y todos asociados en una sola variable. Los datos pueden ser de cualquier tipo de dato, es decir, es posible crear una array que tenga una cadena en la primera posición, un número en el segundo, un objeto en el tercero y así sucesivamente. Podremos acceder a estos distintos datos de manera independiente o agrupándolos. Cabe resaltar que un array es un objeto” (2017)

Los arreglos tienen la siguiente estructura:

var persona= [“Mario”, 23, true]

Se declara una variable y los datos que tendrá se declaran dentro de corchetes, dentro de ellos se tendrán los datos que pueden ser de cualquier tipo (String, number. boolean..etc).

Existen diferentes maneras de crear un arreglo.

La primera de ellas se indica el número de posiciones, este es opcional ,para ello se escribe la palabra reservada ***new*** seguido de otra palabra reservada ***Array***.

var frutas = new Array(10);

Si se conoce los elementos del arreglo se le indica dentro de los paréntesis.

var frutas = new Array(“Manzana”, “Mango”,”Platano”);

La segunda forma es a través de corchetes:

var frutas = [“Manzana”, “Mango”,”Platano”]

En caso de que el arreglo está vacío sería de la siguiente manera:

var frutas = [ ];

Para acceder a un arreglo se establece el nombre del arreglo junto a dos corchetes indicando la posición que se desea acceder. Las posiciones de los arreglos comienzan a partir del cero, siendo la primera posición.

Accediendo al arreglo fruta:

```
var frutas = [“Manzana”, “Mango”,”Platano”];
frutas[0]
```

Este arreglo mostrará como resultado *“Manzana”.* Sus valores son los siguientes:

0 = “Manzana”, 1 = “Mango”, 2 = “Platano”

Si se desea saber el número de elementos que contiene el arreglo existe el método ***length****.* Se declara de la siguiente manera:

```
frutas.length
```

El resultado obtenido del anterior ejemplo será *3,* el número de elementos que contiene el arreglo fruta.

Los arreglos actúan como una pila, permiten crear un grupo de datos para agregar o quitar. Este tipo de pilas se les conoce **LIFO** por sus siglas en inglés ***Last In, First Out*** y por su traducción *último en entrar - primero en salir,* lo que significa que el elemento más reciente añadido es el primero en ser eliminado. Para ello existen dos métodos, ***push*** y ***pop****.*

El método ***push*** añadirá elementos que se le proporcionen y regresará una nueva longitud al arreglo. Este método acepta cualquier número de argumentos y los agrega al final del arreglo

```
var frutas = [“Manzana”, “Mango”,”Platano”];
frutas.push(“Durazno”, “Melon”);
```

El resultado del arreglo incorporando durazno y melón es el siguiente:

*[“Manzana”, “Mango”,”Platano”,“**Durazno**”, “**Melon**”];*

Ahora el arreglo frutas contiene 5 elementos en conjunto con los valores establecidos.

Si se requiere eliminar los elementos de un arreglo se usa el método ***pop***. Este permite eliminar el último elemento establecido, disminuye la longitud del arreglo hasta dejarlo sin elementos.

```
var frutas = [“Manzana”, “Mango”,”Platano”,“Durazno”, “Melon”];
frutas.pop()
```

Si se muestra el arreglo después de ejecutar el método *pop* se mostrará que el último elemento, en este caso *"Melon"* ha dejado de pertenecer al arreglo

["Manzana", "Mango","Platano","Durazno"];

Yeison Daza describe en su artículo que "En JavaScript, un objeto es un entidad independiente con propiedades y tipos".(2016)

Hay tres maneras de crear objetos:

La primera manera se crea un objeto literal, nombrando una variable y entre llaves {} declarando los objetos con un valor.

var persona = {Nombre: "Mario", apellido: "Ochoa", edad: 23, altura:"177 cm"}

La segunda manera se crea un objeto usando la palabra ***new object()*** enseguida se indica el nombre del objeto seguido de un punto y su elemento, posteriormente su valor.

```
var persona = new object()

persona.Nombre ="Mario"
persona.apellido = "Ochoa"
persona.edad =  23
persona.altura = "177 cm"
```

Finalmente la tercera manera donde se crea un objeto desde una función. Dentro de la función se indican los parámetros que tendrá el objeto y dentro de la misma función indicamos con la palabra ***this*** los nombres de los objetos para finalmente dar su valor a través de una variable.

```
function persona(nombre,apellido,edad,altura) {
        this.Nombre = nombre;
        this.Apellido = apellido;
this.Edad = edad;
this.Altura = altura;
}

var mario = new persona("Mario","Ochoa", 23, "177 cm");
```

2.3.7 DOCUMENT OBJECT MODEL

Document Object Model o por sus siglas en inglés **DOM** es la forma en que internamente el navegador organiza todo el contenido del documento HTML dentro de una estructura de árbol. Dentro de él se pueden manipular los objetos que contiene el navegador.

Carlos Roberto define el **DOM** como “Un modelo que traduce la estructura de un documento HTML a un árbol de objetos cuando este es cargado en un navegador web. JavaScript puede acceder y cambiar todos los elementos de un documento HTML a través de este modelo.” (p.13, 2009).

En el siguiente código se puede observar un documento HTML que tiene un encabezado <*h1*>, un párrafo <*p*>, una imagen <*img*> y un enlace <*a*>.

```
<!DOCTYPE html>
<html>

    <head>
       <title>DOM</title>
    </head>

    <body>
       <h1>Document Object Model</h1>
       <p>DOM es un modelo que traduce la estructura de un
    documento HTML a un árbol de objetos cuando este es cargado
    en un navegador web. JavaScript puede acceder y cambiar todos
    los elementos de un documento HTML a través de este
    modelo</p>
       <img src="../ejemplo.jpg">
       <a href="https://encrypted-
    tbn0.gstatic.com/images?q=tbn%3AANd9GcR6UpsuHKgB4KfjDC
    vV3EX5fje4SbiHvBXOvHvTorA4yYjOSnCh"></a>
    </body>

</html>
```

En la imagen 2.23 se muestra el árbol de objetos generado con **DOM** una vez que el documento HTML ha sido cargado en el navegador.

Se observa que la jerarquía de los objetos generados coincide con la jerarquía de las etiquetas contenidas en el HTML.

El nodo raíz se llama document y representa al documento HTML.

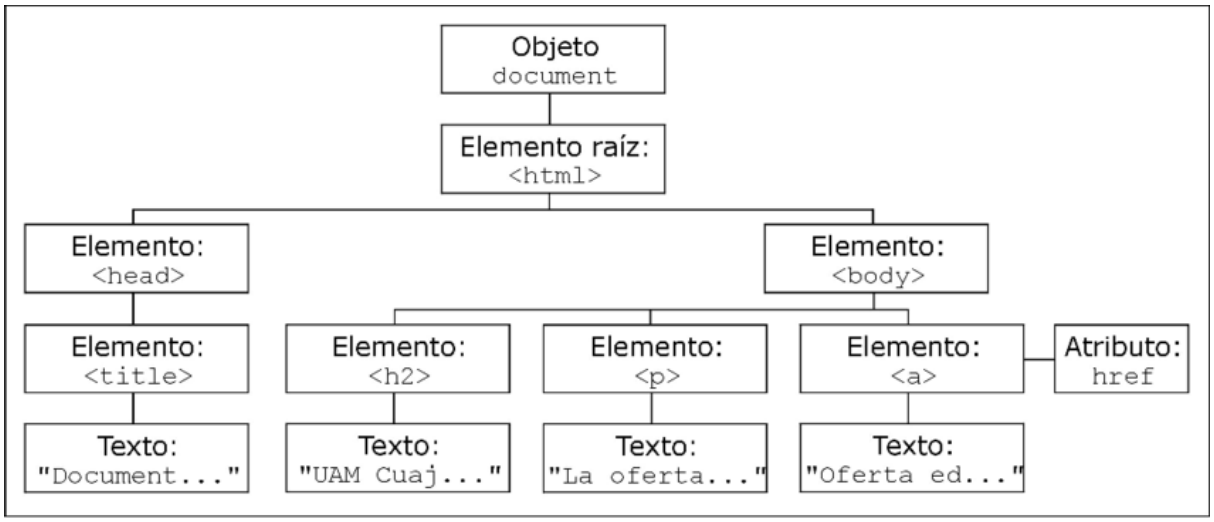

Imagen 2.23 Árbol de objetos
https://www.researchgate.net/figure/Figura-14-Arbol-de-objetos-DOM-para-el-documento-HTML-del-listado-12_fig2_303805672

Unidad 3 APLICACIONES WEB PROGRESIVAS

3.1 FUNDAMENTOS DE APLICACIONES WEB PROGRESIVAS

Para entender el concepto de lo que es una aplicación web progresiva se debe tener en claro que no tiene relación alguna con el diseño en diferentes tamaños de pantallas o también llamado diseño responsivo para el desarrollo de páginas web en dispositivos móviles. Las aplicaciones web progresivas son desarrolladas pensando en que los usuarios finales van ingresar a ella usando primero un dispositivo móvil, llevando el nombre de este concepto como **Mobile first**, siguiendo por tabletas y pantallas más grandes como de escritorio.

Las aplicaciones web progresivas es una aplicación web que también puede ser una página web que progresivamente va implementando características como notificaciones, una ubicación en la pantalla principal del dispositivo y su principal característica es su funcionalidad sin internet. Otra de sus características, a parte de las ya mencionadas anteriormente, es que usan funcionalidades nativas del dispositivo, se actualizan constantemente y su tiempo de carga es menor.

Este tipo de aplicaciones pueden ser confundidas con las aplicaciones nativas, con la diferencia de que el tiempo de carga de las aplicaciones progresivas es menor a las nativas y se actualizan constantemente.

Un punto importante a mencionar es que estas aplicaciones deben funcionar bajo el protocolo *https* y deben tener un certificado pese a que usan un servicio denominado *Service Workers*, este servicio tienen control total sobre la aplicación y esto puede ser consecuente por lo que alguien puede interceptar ese servicio y deshabilitar la aplicación. Es por esa razón que la aplicación debe estar bajo el protocolo *HTTPS.*

Al momento de ingresar a un sitio web tradicional, se realiza una petición a internet el cual responde con los archivos ***index.html***, ***CSS*** *y* ***Javascript*** dando la interfaz del sitio, sin embargo si no se tiene conexión a internet este sitio deja de funcionar. Por algunos años las páginas web o las aplicaciones web tradicionales habían sido así hasta que apareció lo que es el ***service worker*** que no es más que un ***proxy*** que se coloca entre la página web y el internet.

El ***service worker*** es un archivo de *Javascript* plano. A diferencia de las aplicaciones web tradicionales, lo que hace una aplicación progresiva es que las peticiones son interceptadas por el mismo servicio y regresa toda la página web sin comunicarse a internet o bien solo manda llamar ciertos recursos que deben ser cargados desde internet. De esta manera la red responde al *service worker* y este le responde al navegador renderizando toda la información. Es por esta razón la importancia que este bajo un protocolo seguro ya que el service worker puede cambiar absolutamente cualquier cosa de la aplicación.

Para ilustrar cómo funciona una petición que pasa por el servicio se observa la imagen 3.1

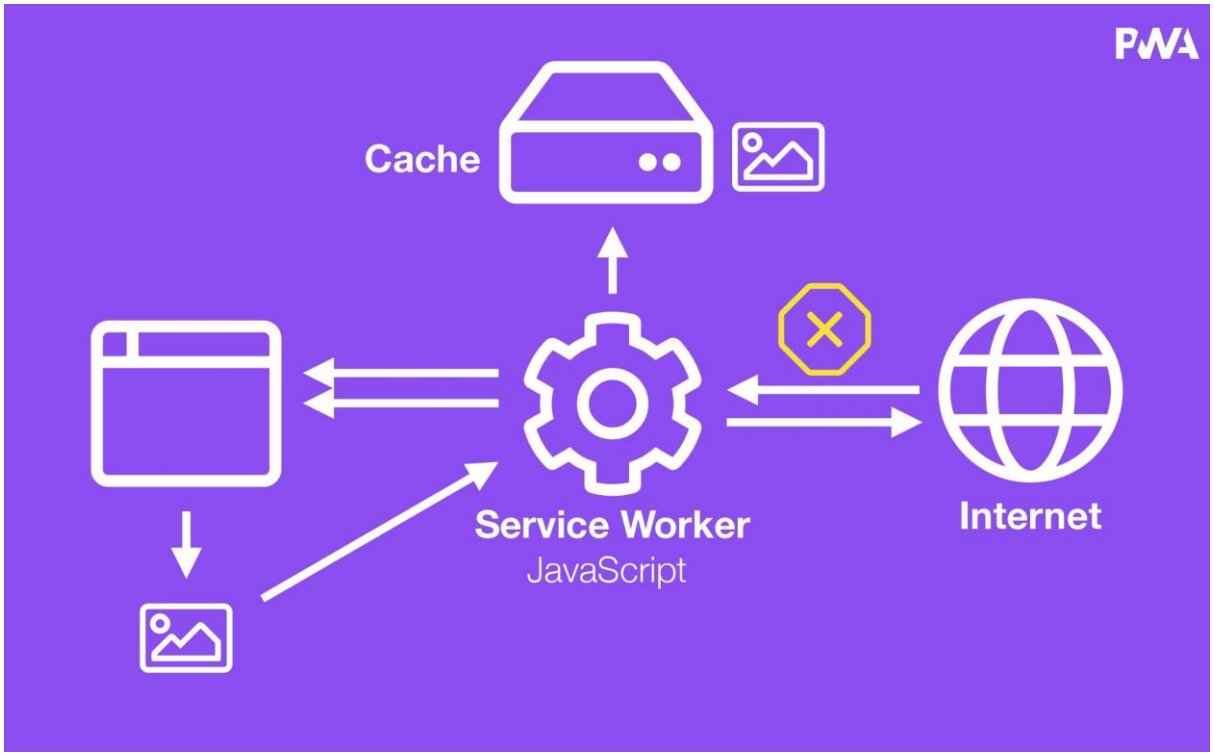

Imagen 3.1 Petición a un service worker
Referencia: Aplicaciones Web Progresivas por Fernando Herrera
https://www.udemy.com/course/aplicaciones-web-progresivas/

En la imagen 3.1 se observa como ejemplo a una página web que va hacer una petición a una imagen, se puede interpretar que esa imagen va directamente a la web, aunque está siendo interceptada por el servicio. En este caso el recibe la solicitud de la imagen y al mismo tiempo revisa en un espacio llamado *cache* si esa imagen existe. Si la imagen existe entonces la regresa al navegador, pese a lo dicho en ningún momento esa petición pasó por la web y demoró centésimas de segundo responderle al usuario.

En contraste si la imagen no existe en el *cache* entonces el servicio hace la petición a internet para obtener esa imagen, una vez que obtiene esa imagen el service worker graba la imagen en el *cache* por si un usuario vuelve a solicitar la misma imagen y no será necesario que se vuelva a conectarse a internet.

En particular el ***service worker*** se ejecuta bajo un hilo independiente a la aplicación web, en otras palabras no se ejecuta en las páginas principales y es totalmente independiente, esto significa que el usuario puede cerrar la aplicación y el servicio va seguir trabajando.

La instrucción para llamar a un ***service worker*** se puede encontrar de la siguiente manera:

navigator.serviceWorker.register(sw.js')

El ciclo de vida de un service worker empieza desde que se llama al archivo de *javascript*, la primera vez cuando se llama el archivo el *service worker* se instala y posteriormente viene el paso de la activación y por último entra en una fase de "*Esperando*". Cuando se llama por segunda vez el archivo de ***Javascript*** el ***service worker*** pasa a una fase de actividad.

En la fase de instalación se descarga el archivo de *Javascript,* enseguida el archivo es revisado y finalmente entra en la fase de instalación. Pese a lo dicho si el archivo falla por razones de sintaxis o que no encuentra algún archivo se va ejecutar un error. Hasta que se recargue el navegador. Si se hace correctamente por ende se entra a la siguiente fase cuando ya está instalado.

Cuando ya está instalado se le llama la fase de "*Espera*", automáticamente si no existe un servicio en ejecución, este seguirá a la siguiente fase. Sin embargo si ya existe otro entonces a que todas las ventanas sean cerradas para poder entrar al siguiente paso o fase.

La siguiente fase es la activación y sucede justo antes de que tome el control de la aplicación. En este momento se pueden hacer limpiezas en el código para evitar errores en la ejecución.

Finalmente la última fase es cuando el *service worker* está activo, en este momento este mismo obtiene control de la aplicación web para realizar las peticiones que se requieran.

3.2 FETCH EVENT Y SERVICE WORKER

Fetch surge con el nuevo estándar de Javascript el cual permite hacer peticiones de forma asíncrona, es decir se espera que las peticiones se realizan primero para poder finalizar esa petición, esto se realiza de forma sencilla a través de promesas. Proporciona una interfaz de Javascript para manipular peticiones y respuestas *HTTP.* Es importante mencionar que la respuesta que se recibe de la petición fetch se debe convertir a un formato **json,** por sus siglas en inglés significan **JavaScript Object Notation.**

La forma de implementar una petición básica de **fetch** se observa en el siguiente código:

```
fetch('http://ejemplo.com/usuarios.json')
        .then( function(respuesta) {
        return respuesta.json()
 })
.       .then(function(objeto){
        console.log(objeto);
})
        .catch(function(err) {
        console.log(err)
});
```

El código anterior se manda una petición ***fetch*** a una dirección la cual se quiere acceder a un parámetro, está llamada regresa una promesa. El método ***then ()*** de esa promesa regresa un objeto respuesta, de ese objeto se llama el método ***json ()*** para extraer el contenido en ese formato, inmediatamente regresa otra promesa que se resolverá cuando se haya obtenido el contenido. Después el método ***then ()*** de esa promesa recibe el cuerpo devuelto por el servidor en formato ***json***, mientras tanto si ocurre algún error se incluye el ***catch ()***

Esto trae consigo una respuesta HTTP, no el archivo ***json***. Para extraer el contenido del **json** desde la respuesta se usó el método ***json()***.

El método **fetch** acepta un segundo parámetro, el cual es opcional y un objeto que permite controlar algunos ajustes. Entre estos ajustes trae consigo el método con el que se desea hacer la petición, las cabeceras que se definen como ***headers***, el modo y el cache.

Para ilustrar el lo anterior, se muestra el siguiente código:

```
    fetch("http://ejemplo.com/usuarios.json",{
    method: 'POST',
    headers: {
      'Content-Type': 'application/json'
    },
    body: JSON.stringify({"a": 1, "b": 2}),
    cache: 'no-cache'
})
.then(function(respuesta) {
    return response.json();
})
.then(function(data) {
    console.log('data = ', data);
```

```
})
.catch(function(err) {
   console.error(err);
});
```

Existen varias opciones a configurar, de las cuales son las siguientes:

method: El método a usar, ya sea POST para enviar datos o GET para recibir
headers: La cabecera que se enviará
body: El cuerpo que se envía al servidor, puede ser una cadena o un objeto.
mode: El modo en que se enviará la solicitud, como por ejemplo: ***cors, no cors, navigate***
cache: La forma en que se utiliza el cache en la petición. Está puede ser por default o de recarga cuando se envía una petición nueva.

Cuando se usa el ***service worker***, se encuentra como un simple archivo de *Javascript* que a medida que va teniendo funcionalidades el archivo va creciendo y se compone varios eventos llamados ***eventListener***.

El evento ***eventListener*** se representa de la siguiente forma:

```
self.addEventListener("Acción...", event => {

});
```

En el ejemplo anterior se muestra la palabra "Acción.." que está dentro del evento, este evento puede tener diferentes funcionalidades, como por ejemplo cuando se está haciendo la instalación del ***service worker*** se manda llamar esa misma función para ejecutar diversas tareas o bien se puede ejecutar cuando se activa el servicio o se recibe alguna notificación.

El proceso de instalación de un ***service worker*** en un dispositivo móvil inicia desde que se navega en alguna dirección en internet el cual regresa una serie de archivos para mostrar la información de la aplicación web o página web. Las peticiones enviadas del dispositivo móvil a la web fueron traer archivos HTML, trajeron el Javascript y el CSS. Posteriormente dentro del código de Javascript se ejecuta una instrucción que se da de la siguiente manera:

```
navigator.serviceWorker.register("/sw.js")
```

Posteriormente de ejecutar esa instrucción, el ***service worker*** se instala en el dispositivo.

En contraste, si algún service worker ya está instalado o se lanza una segunda petición después de haber instalado el servicio, esta petición la hará directamente con ese mismo e inmediatamente regresa una respuesta al dispositivo. Sin embargo habrá ocasiones en que el service worker permite la comunicación en la web, traer la información y procesarla para mostrarla en el navegador web.

Cabe resaltar que el ***service worker*** es independiente a la aplicación web, aunque también puede tener una comunicación con servicios externos.

Es importante saber que aunque se cierre la aplicación web o navegador y no haya alguna comunicación, el ***service worker*** sigue escuchando eventos y seguirá hasta que se instale un nuevo service worker o se elimine. De esta manera permite recibir notificaciones, saber cuándo se recupera o se pierde la conexión a internet y hacer actualizaciones en segundo plano.

Finalmente, se debe recordar que es importante que el ***service worker*** funcione bajo el protocolo HTTPS, de está manera su funcionamiento será correcto.

Para el siguiente ejemplo se importará el módulo de ***node http-server***. Para instalarlo se debe ejecutar como administrador la consola de comandos y ejecutar la siguiente línea:

npm install http-server -g

Este módulo ejecuta un servidor http el cual permite visualizar páginas *HTML*.

Para ejecutar un documento *HTML* con el servidor http previamente instalado se ejecuta el siguiente comando desde la consola o terminal. Se debe tener un archivo *HTML* para que el módulo reconozca que archivo ejecutar.

http-server

Una vez ejecutado lo anterior se mostrará el siguiente mensaje en la consola o terminal. En la imagen 3.2.1

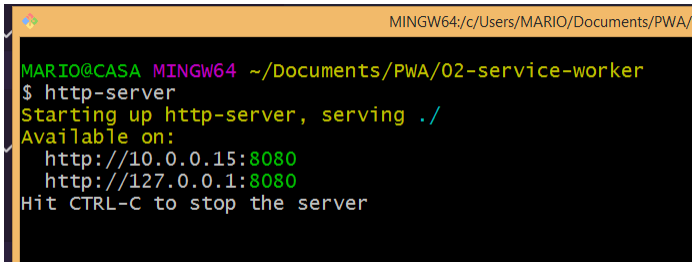

Imagen 3.2.1 Terminal o Consola de comandos

El mensaje que se muestra en la imagen 3.1 indica que se ha iniciado un servidor con una dirección IP y el puerto 8080.

Dentro del navegador se puede observar la siguiente dirección:

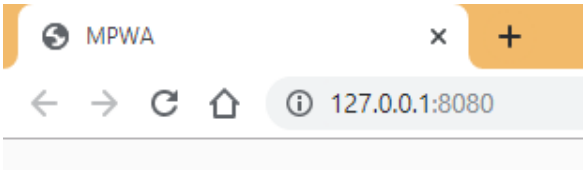

Imagen 3.2.2 Dirección dentro del navegador

En la imagen 3.2.2 se muestra que el documento *HTML* se está ejecutando en un servidor local, a pesar de que nos indica una dirección IP, esta misma puede ser reemplazada por *localhost: 8080*.

Para este primer proyecto, se tendrán dos archivos como un inicio. Un documento ***index.html*** y un archivo *Javascript* que estaré dentro de una carpeta nombrada *js*, de tal manera que se muestra a continuación:

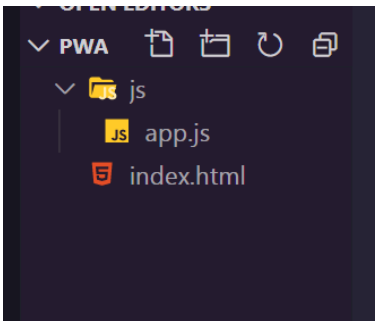

Imagen 3.2.3
Estructura del proyecto

Para asegurar que el navegador pueda utilizar los ***service workers***, se debe escribir las siguientes líneas en el archivo de Javascript:

if(navigator.serviceWorker) {

```
        console.log("Se puede usar el service worker")
}
```

Si el navegador tiene soporte para usarlos. Dará el mensaje Se puede usar el ***service worker*** en la consola del navegador, de lo contrario no aparecerá ningún mensaje.

Para iniciar el proceso de instalación se escribe el archivo de Javascript la siguiente línea de código

```
navigator.serviceWorker.register('/sw.js')
```

El archivo del ***service worker*** se debe colocar en la raíz de la aplicación web o al mismo nivel en donde se encuentra el documento ***index.html***.

El archivo ***sw.js*** no existe en un inicio, por lo tanto se debe crear en la raíz de la aplicación, así como se mencionó anteriormente.

Cuando el archivo ***sw.js*** esté en el proyecto, dentro de él se ejecutarán las instrucciones que le mande la aplicación web.Como prueba de instalación se escribirá en el archivo sw.js un mensaje para que aparezca en consola.

Al recargar el navegador se puede observar en las herramientas de desarrollo el mensaje que se escribió desde el archivo ***sw.js*** cómo se observa en la imagen 3.2.4

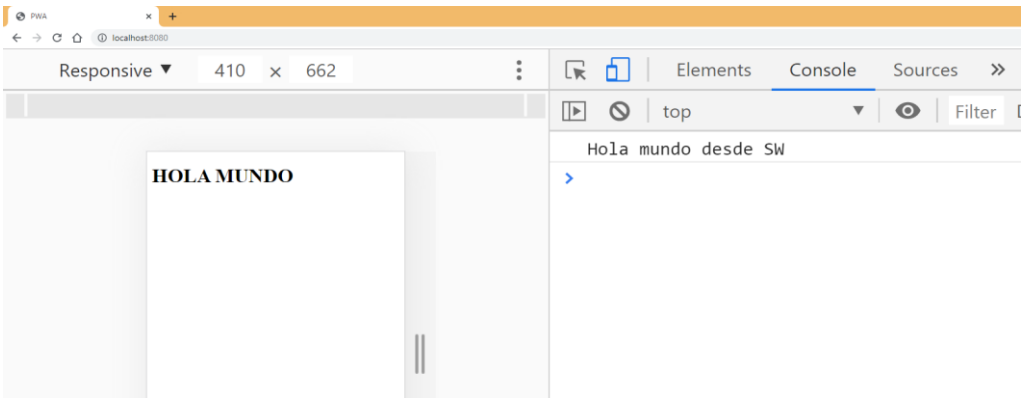

Imagen 3.2.4 Mensaje de consola desde archivo sw.js

Si se vuelve a recargar el navegador se podrá observar el mensaje no vuelve aparecer, esto es porque el ***service worker*** ya se instaló, es decir cuando el navegador interpreta la instrucción de registrar el servicio por segunda vez, no la ejecutó debido a que es el mismo.

Dentro de las herramientas de desarrollo, se encuentra la opción de aplicación, dentro de ella se pueden observar varias opciones, una de ellas son los ***service workers*** la cual podemos observar en la imagen 3.2.5 que se muestra a continuación el status de ese servicio

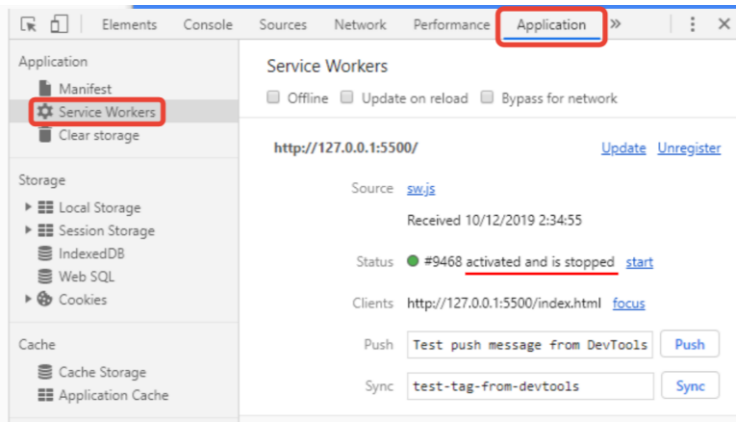

Imagen 3.2.5 Estado del service worker

De esta manera se muestra que el ***service worker*** está activado y ejecutándose en el navegador.

Dentro del archivo ***sw.js*** va contener acciones a sucesos que pasen en la aplicación, por lo que escribir código de Javascript cómo declarar variables no es lo adecuado.

En el siguiente código, se muestra un ejemplo dentro del archivo ***sw.js*** el evento ***fetch***. Este evento funciona para cargar información desde un servicio.

```
self.addEventListener('fetch', event => {
        console.log(event)
});
```

Posteriormente cuando se cargue el navegador se podrá observar que ha entrado un nuevo ***service worker*** que está esperando para ser activado, así como se muestra en la imagen 3.2.6.

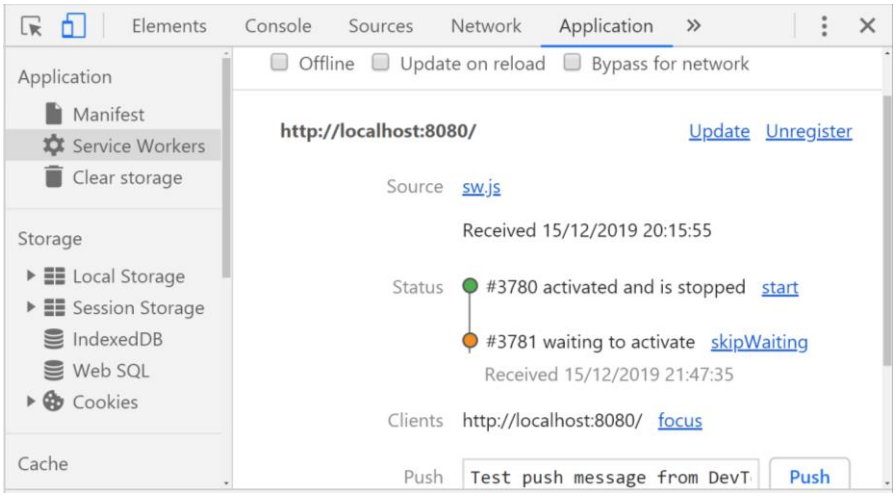

Imagen 3.2.6 Entrada de un nuevo service worker

Mientras el nuevo ***service worker*** no se active, consecuentemente el código que se escriba en el archivo ***sw.js*** no se ejecutará. Para activar el nuevo ***service worker*** se selecciona la opción ***skipWaiting*** que da en las herramientas del navegador.

Cuando el nuevo servicio sea activado se podrá observar en la consola del navegador la petición fetch que se mandó así como se muestra en la imagen 3.2.7

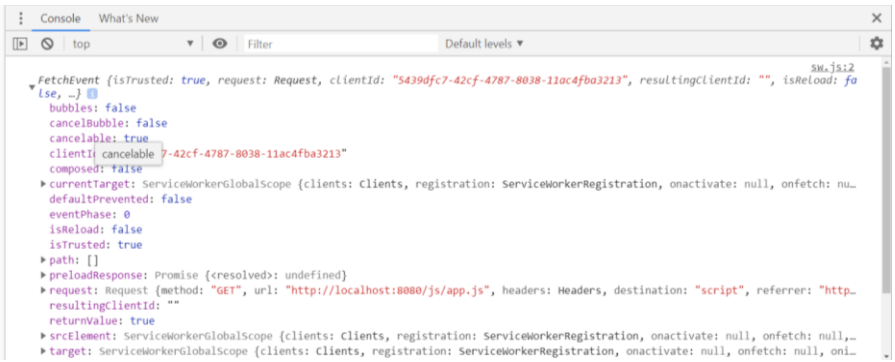

Imagen 3.2.7 Mensaje en consola del evento fetch

Para la siguiente línea de código se manda una respuesta a la petición y retorne la información desde el ***service worker***

```
self.addEventListener('fetch', event => {
        event.respondWith( fetch (event.request ) );
});
```

Dentro de las herramientas de desarrollo del navegador se puede observar en la imagen 3.2.8 que la información que regresa del archivo ***app.js*** lo regresa desde el ***service worker***.

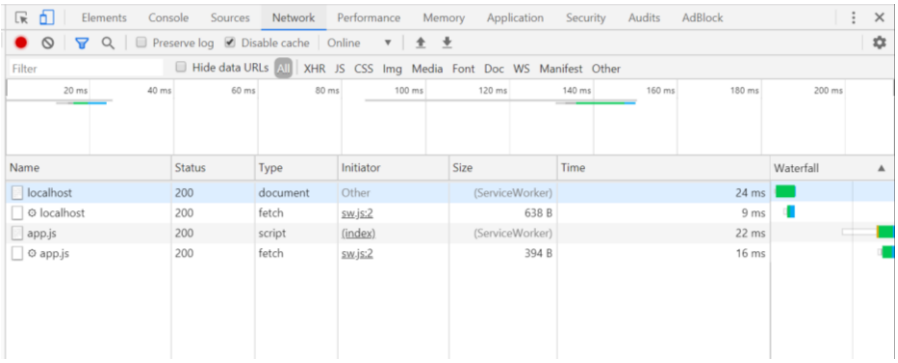

Imagen 3.2.8 Archivos dentro de la carpeta

Dentro de ***fetch*** es posible buscar la petición que hace la aplicación, es decir saber de qué archivo hace la petición para mostrar los estilos, los eventos y el servidor.

Para ver hacia donde está haciendo la petición se manda a la consola del navegador la ruta url de donde la hace, asi como se muestra en el siguiente código:

```
self.addEventListener('fetch', event => {
        console.log(event.request.url)
});
```

El código anterior da como resultado la imagen 3.2.9 que se muestra a continuación:

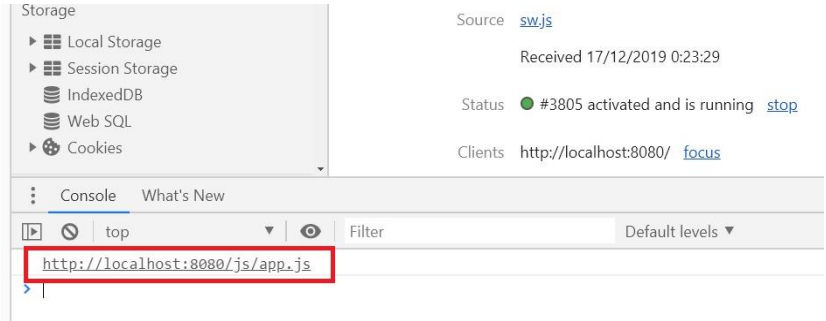

Imagen 3.2.9 Petición Fetch

Hasta el momento el ***fetch*** únicamente está haciendo la petición al archivo ***app.js***, por lo cual solo muestra esa unica peticion, como se muestra en la imagen 3.8.

3.3 CICLO DE VIDA DE UN SERVICE WORKER

De acuerdo con Jake Archibald (2019) en su artículo de fundamentos de la web "El ciclo de vida del *service worker* es la parte más complicada de este. Si desconoces lo que intenta hacer y los beneficios que ofrece, puede parecer que molesta. Sin embargo, una vez que conoces cómo funciona, puedes ofrecer actualizaciones discretas y fluidas a los usuarios, mezclando lo mejor de los patrones web y nativos."

Al el primer ***service worker*** ocurren una serie de eventos, el primero de ellos es el evento de instalación. Durante este evento pasa una función que señala la duración o si hubo algún error.

Si no hubo error en la instalación, el ***service worker*** pasa a la segunda fase de activado. En esta etapa se administrarán los caches y los elementos estáticos que estén almacenados.

Como tercera etapa entrará en **Espera** hasta que llegue un nuevo s***ervice worker***. Está etapa funcionara hasta la siguiente carga de información. Dentro de esta etapa se hacen las peticiones con ***fetch*** que controlará los eventos. Finalmente si no se recibe ningún evento o peticiones para cargar en la aplicación entrara en la etapa de terminado, por lo tanto regresará al estado o etapa de espera.

Para ilustrar lo anterior, se muestra la imagen 3.9 donde se observa cada una de las etapas en las que pasa el ***service worker***.

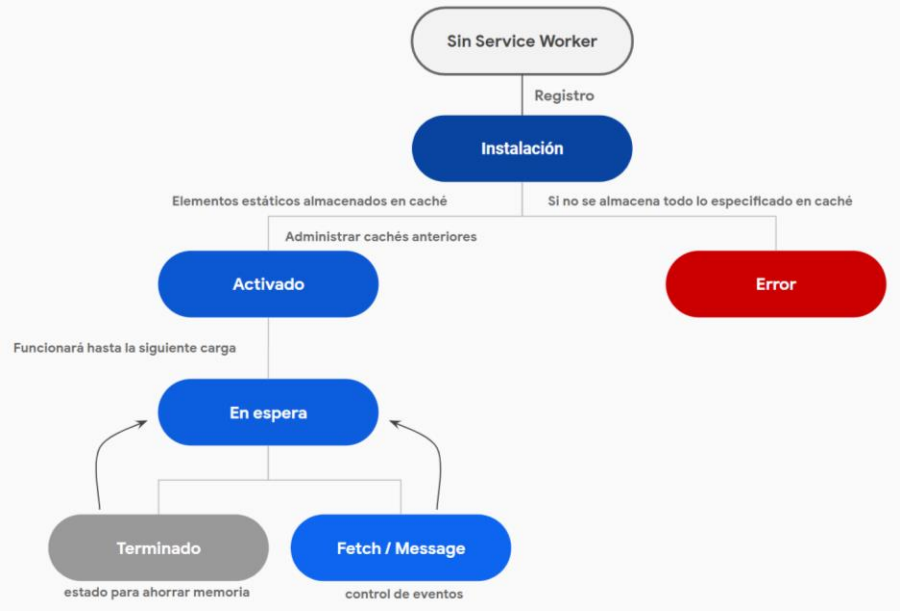

Imagen 3.3.1 Ciclo de vida de un service worker

Cuando es la primera vez que una página web carga, está envió una petición a internet, inmediatamente trae toda la información de los archivos, entre ellos el archivo ***sw.js*** que se registrará por medio de una instrucción del archivo ***app.js***.

Durante la instalación del ***service worker*** se pueden ejecutar instrucciones cómo descargar archivos de Javascript que componen la aplicación, se crea un caché entre otras cosas.

Para ver el evento en la consola del navegador se indica desde el archivo **sw.js** el siguiente código:

```
self.addEventListener("install", event => {
        console.log(event)
});
```

El resultado obtenido del código anterior se muestra en la siguiente imagen 3.3.2

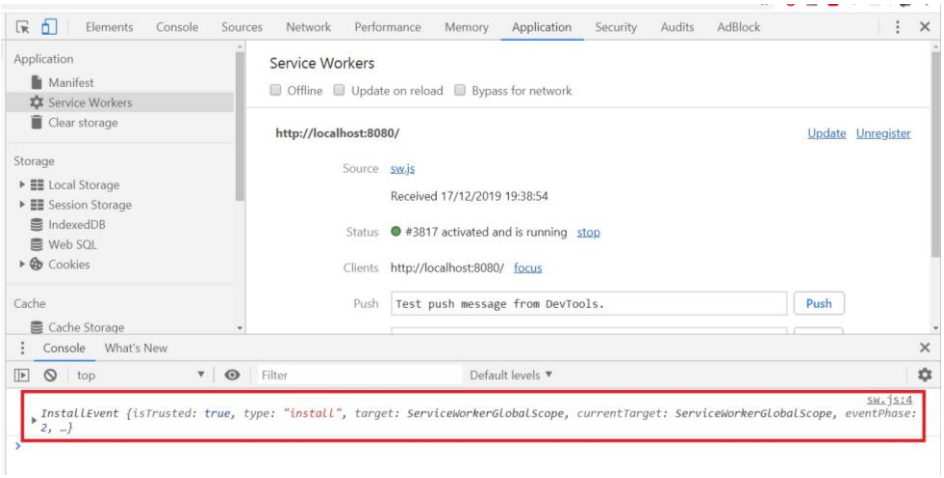

Imagen 3.3.2 Etapa de instalación mostrada en consola del navegador

Si el navegador se refresca o se recarga, este mensaje ya no aparecerá porque es el mismo ***service worker*** que ya está instalado en la aplicación.

Para observar cuando el ***service worker*** está tomando control de la aplicación o entra en la etapa de activación se agrega el siguiente código para imprimir en la consola del navegador el evento de activación.

```
self.addEventListener("activate", event => {
        console.log(event)
});
```

Cómo resultado se muestra en la consola del navegador la activación del ***service worker*** cómo se muestra en la imagen 3.3.3

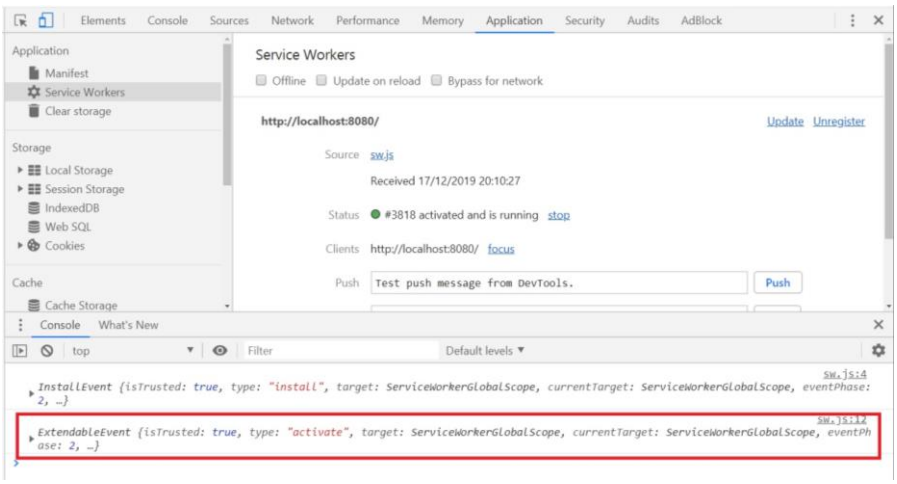

Imagen 3.3.3 Activación del service worker

Durante esta estaba de activación se limpia el ***cache*** viejo del navegador dando lugar a uno nuevo y actualizado para la activación. Para este momento el ***service worker*** tiene control total sobre la aplicación.

Cómo ejemplo, el ***service worker*** puede manipular los estilos de la aplicación y los eventos que entran, es por esto que la aplicación debe estar bajo un protocolo *HTTPS*

En la imagen 3.3.3 se muestra de nuevo el evento de instalación, como resultado que cuando se modifica el archivo sw.js para nuevos eventos, este ingresa a la aplicación como un nuevo ***service worker***.

3.4 ESTRATEGIAS DE CACHE PARA MODO OFFLINE

Saber que estrategias de cache se debe implementar en una aplicación web progresiva es un tema muy importante después del funcionamiento del s***ervice worker***. Es de suma importancia saber cómo funcionará la aplicación para poder aplicar una estrategia del manejo del cache eficiente y que sirva para brindarle al usuario final la mejor experiencia posible.

Existen 5 estrategias de cache comunes que se implementan en las aplicación progresivas

- Estrategia Solo red o *Network Only*
- Estrategia Red primero o *Network First*
- Estrategia Cache primero o *Cache First*
- Estrategia Solo Cache o *Cache Only*
- Estrategia Cache mientras válida o *Stale While Revalidate*

Para implementar alguna de las estrategias previamente mencionadas se debe enviar una petición para detectar cuando la conexión a la web sea nula o esa misma falle.

Dentro del archivo ***sw.js*** se crea un evento enviando una petición ***fetch*** el cual regresa una respuesta. La respuesta se recibe mediante el evento ***respondWith ()***, si esa respuesta es válida lo interpreta el navegador para finalmente si hay algún error se implementa un ***catch*** para controlarlo y poder regresar con el atributo ***Response ()*** un valor que diga cuándo se ha perdido la conexión o ha fallado.

Dentro del atributo ***Response ()*** se especifica una respuesta manual, en este caso se manda un mensaje cuando no haya conexión a internet.

Para ilustrarlo se puede observar el siguiente código:

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

    const offlineResp = new Response(`Debe haber conexión para
usar    esta página`)

    const resp = fetch(event.request).catch( () => offlineResp );
    event.respondWith(resp);

    });
```

En el código anterior se definen dos constantes. La primera ***offlineResp*** se le asigna el valor del atributo ***Response()*** el cual se envía una respuesta manual.

Para la segunda constante se le asigna el valor de la petición, seguido de un **catch()** donde se manda llamar la primera constante en caso de que no haya conexión a internet.

Al ver en el navegador se debe activar la opción ***offline*** que se encuentra en las herramientas de desarrollo del navegador, así como se muestra a continuación en la imagen 3.4.1

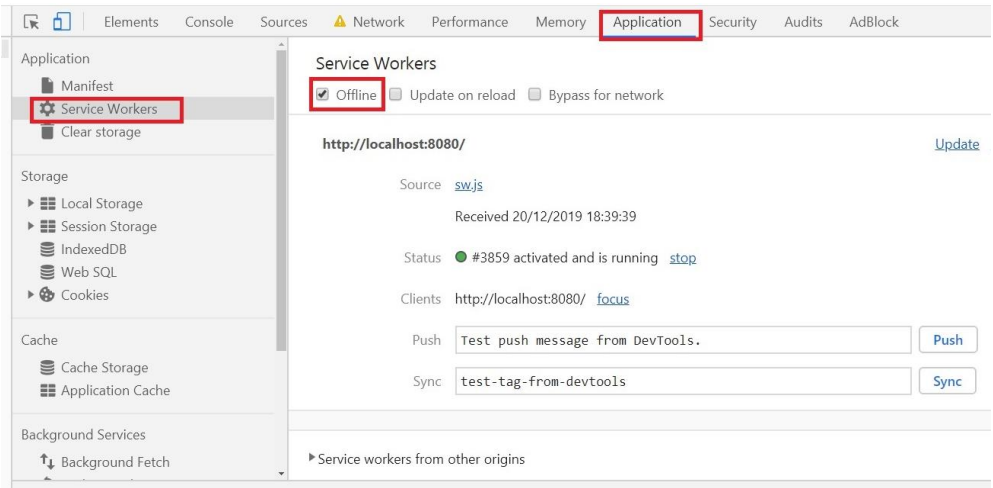

Imagen 3.4.1
Opción offline en herramientas de desarrollo

Una vez activada la opción offline en las herramientas de desarrollo del navegador se puede ver el resultado del código anterior en la siguiente imagen 3.4.2

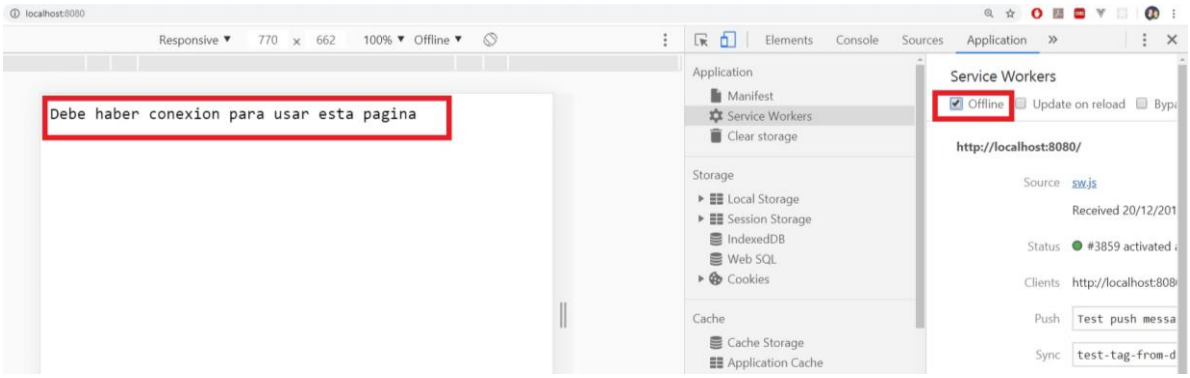

Imagen 3.4.2 Herramienta de modo offline activado

Para hacer uso de las estrategias cache se debe optimizar el mensaje que dará cuando no haya conexión a internet, para esto se hace uso del ***cache storage***.

El ***cache storage*** es una propiedad del objeto ***window***, esto se puede observar desde la consola del navegador mandando llamar ***window.caches***. Este ejemplo se muestra en la imagen 3.4.3.

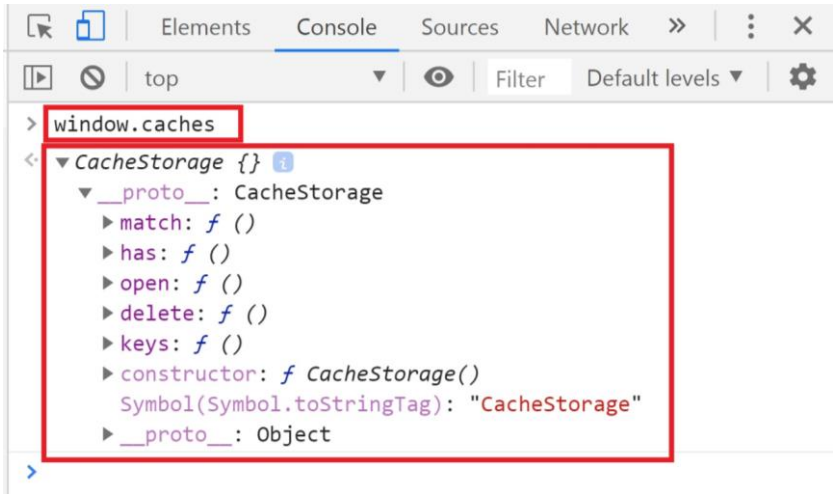

Imagen 3.4.3 Respuesta del evento window

Está propiedad no es soportada por algunos navegadores, de esta manera se debe indicar en el archivo app.js una validación en caso de que el cache se pueda o no usar en el navegador donde se abra la aplicación web. El código se representa de la siguiente manera:

```
if(window.caches) {
```

```
        caches.open('prueba1')
}
```

Es decir, el código anterior se indica que si existe la propiedad ***cache*** en el navegador, este buscara ese espacio de almacenamiento e intentara abrir un espacio llamado **prueba1,** de lo contrario si no existe, el cache creara ese espacio. Esto se puede observar en la siguiente imagen 3.4.4.

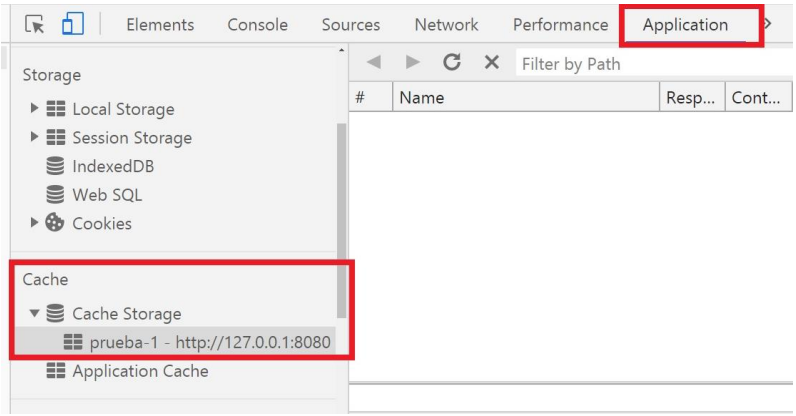

Imagen 3.4.4 Prueba almacenada en cache

Dentro del ***cache storage*** se pueden almacenar archivos, imágenes, documentos html, para ello se escribe el siguiente código:

```
if(window.caches) {
        caches.open('prueba1')
        cache.open('archivos-cache').then(cache => {
                cache.addAll([
                        '/index.html'
                ])
        })
}
```

La función del código anterior es crear un almacenamiento llamado ***archivos-cache***, esto regresa una promesa de Javascript, dentro de ella se abre un método para almacenar diversos archivos, este método es ***cache.addAll***. Dentro de este método se inicia un arreglo con las rutas a los archivos que se deseen almacenar. El resultado se muestra a continuación en la imagen 3.4.5.

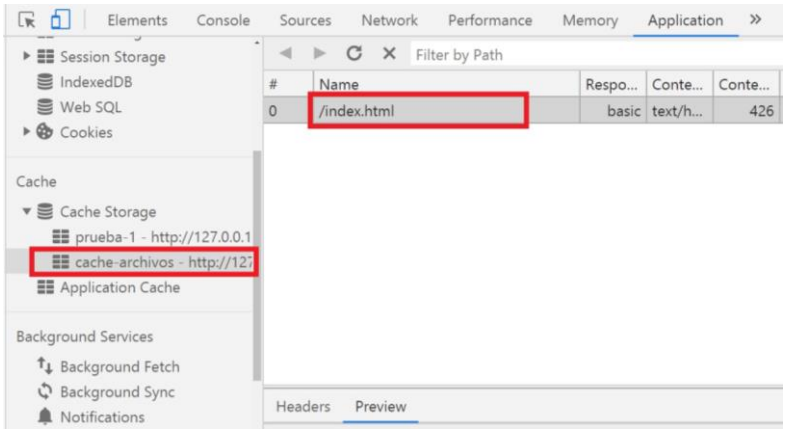

Imagen 3.4.5 Archivo index guardado en el cache storage

Dentro del cache se almacenan las peticiones y los archivos. De esta manera aunque se pierda la conexión a internet, se podrán consultar en un futuro esos archivos desde ese almacenamiento.

Para borrar algún archivo del cache se indica con un método ***delete*** que es propiedad del ***cache***, se puede ver en el siguiente ejemplo de código:

*cache.delete('**Archivo**');*

3.4.1 SHELL DE APLICACIÓN

De acuerdo con Addy Osmani informó en su artículo El modelo de shell App la describe cómo: "Una arquitectura de **shell de aplicación** (o ***shell de App***) es una forma de crear una Progressive Web App que se carga al instante y de manera confiable en la pantalla de tu usuario, en forma similar a lo que ves en las App nativas.(2019)

La "shell" de App es la mínima cantidad de HTML, CSS y JavaScript requeridos para activar la interfaz de usuario, y cuando se almacena en caché sin conexión puede asegurar un **rendimiento instantáneo y de alta confiabilidad** para los usuarios en las visitas repetidas. De esta manera, la shell de la App no se carga desde la red en cada visita del usuario. Solo se carga el contenido necesario de la red.

Para App de una sola página con arquitecturas con mucho código JavaScript, una shell de App es un enfoque acertado. Este enfoque se basa en almacenar la shell agresivamente en caché (utilizando un service worker para lograr que la App funcione. Luego, el contenido dinámico carga cada página a través de

JavaScript. Una shell de App es útil para enviar el HTML inicial a la pantalla en forma rápida y sin utilizar una red.

En otras palabras, la shell de App es similar al paquete de código que publicarías en una tienda de App al compilar una App nativa. Es el esqueleto de tu IU y contiene los componentes principales necesarios para poner en marcha tu App, pero probablemente no contenga los datos."(2017)

La aplicación debe guardar la ***shell*** *de aplicación* en el momento en que un ***service worker*** se esté instalando.

Para guardar los archivos que se usan en la ***shell*** *de aplicación* en el caché, se escribe el siguiente código dentro del archivo ***sw.js****:*

```
self.addEventListener("install", event => {

    const cacheSave = caches.open('cache-1')
        .then(cache => {

        return cache.addAll([
            '/index.html',
            '/css/styles.css',
            '/img/fes.png',
            '/js/app.js',
        ]);
    });

    event.waitUntil(cacheSave);
});
```

Es decir, en primer lugar lo que hace el código anterior es indicar que cuando se instala el ***service worker*** se inicia una variable ***cacheSave*** la cual guarda en el cache los archivos que se indican, enseguida regresa una promesa de ***javascript****,* en un arreglo se escriben las rutas donde se encuentran los archivos, los estilos y las imágenes. Consecuentemente de que la instalación del ***service worker*** es sumamente rápida, se debe dar una pausa para que el proceso de instalación y guardado en el ***cache*** termina. Por ende se inicia el evento ***waitUntil*** , esto hará que suceda primero el proceso de instalación y guardado para después ejecutar los siguientes pasos.

Aunque si en algún momento del guardado en el cache no se logra encontrar algún archivo de los que se han indicado dentro del arreglo, esto dará un error.

3.4.2 ESTRATEGIA DE SÓLO CACHE

La estrategia de sólo caché consiste en hacer una vez la instalación, descargar los recursos que la aplicación necesita. Una vez que los recursos han sido descargados, la aplicación no regresa hacer ninguna petición a la web. Para ilustrar esta estrategia se muestra la siguiente imagen 3.4.2.1.

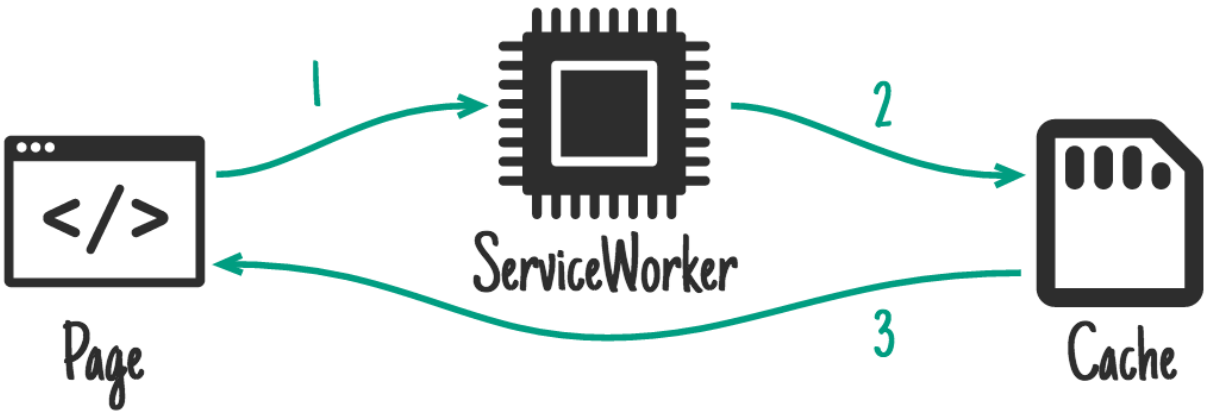

Imagen 3.4.2.1 Estrategia de solo cache

Está estrategia es usada cuando a la aplicacion web se le requiera leer solo los datos del ***cache storage***, es decir no habrá ninguna petición que acceda a la web, únicamente la información que es usada saldrá del ***cache***.

Para esta estrategia se escribe el siguiente código en el archivo *sw.js*:

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

        event.respondWith(caches.match(event.request));

});
```

Cada una de las estrategias ocurre cuando se hace alguna petición ***fetch***. Siguiendo el código anterior, dentro del evento ***listener*** se indica que cuando haga la petición ***fetch*** se inicia un nuevo evento que regrese la instrucción ***caches.match()***.

La instrucción ***caches.match()*** busca en el cache los archivos que esten guardados en el dominio que se encuentre la aplicación, en este caso es ***localhost***.

Ya implementada esta estrategia se puede ver el resultado a continuacion en la imagen 3.4.2.1

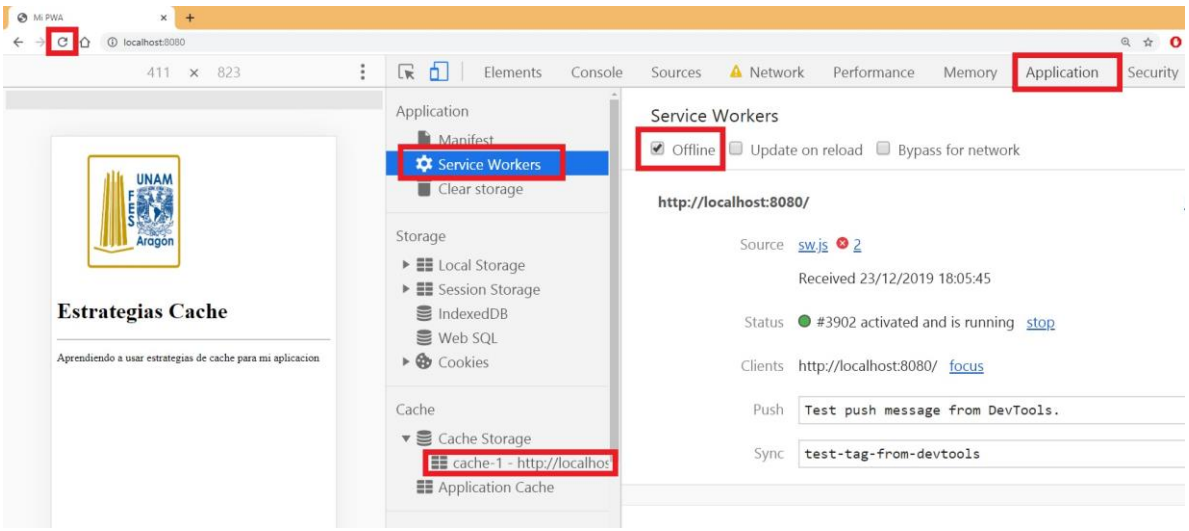

Imagen 3.4.2.2 Modo offline con solo cache

Estando dentro de la aplicación web, se debe ir a las herramientas de desarrollo del navegador y estar en la pestaña de aplicación. Ya estando ahi se puede ilustrar en la imagen 3.4.2.2 que hay archivos en el ***cache storage*** y está activando el modo ***offline***. Si la aplicación se recarga, se mostrara los recursos que ya estaban guardados, de esta manera no se perderá la información que se muestra.

3.4.3 ESTRATEGIA DE CACHE PRIMERO

Está estrategia consiste en verificar primero si la información que se desea consultar está en el ***cache***, sin embargo si algún recurso no está almacenado en el **cache**, esta misma consulta a la web para descargar la información. Enseguida almacena la información dentro del ***cache storage***. Cómo resultado se obtiene solo una petición a la web cada vez que algún recurso no se encuentre.

Para ilustrarlo se puede ver la imagen 3.4.3.1 a continuación:

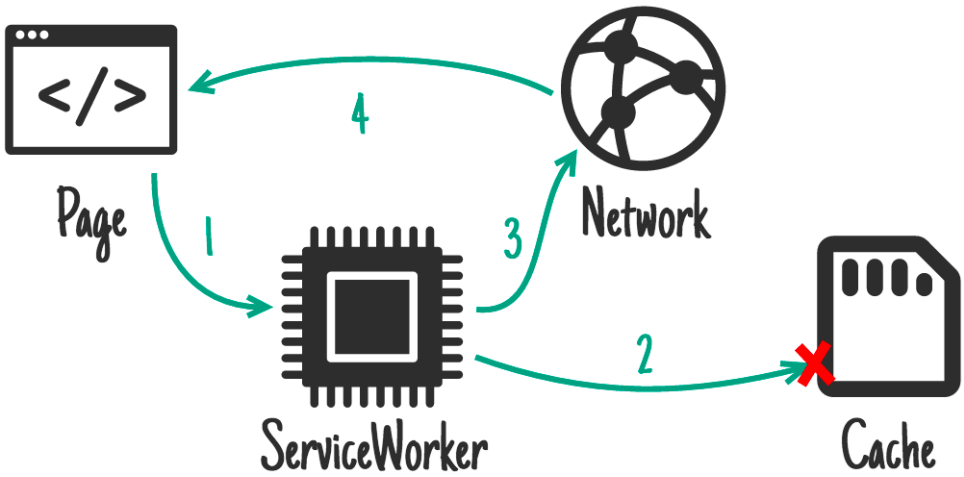

Imagen 3.4.3.1 Estrategia de cache primero y luego red

En el siguiente código se muestra su implementación:

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

    const respuesta = caches.match(event.request)
        .then(res => {

        return fetch(event.request).then(newResp => {

        //guardando en el cache cuando se encuentre el recurso
        caches.open('cache-1').then(cache => {
            // (solicitud, respuesta que regresa la petición)
            cache.put(event.request, newResp);
            });
            return newResp.clone();
        });
    });
    event.respondWith(respuesta);
});
```

En el código anterior se inicia una constante en la que tendrá la instruction ***caches.match***, con está misma se busca una petición ***request*** dando como resultado una promesa de ***Javascript***. Con la promesa se busca el archivo haciendo una petición a la web. Por último si el recurso se encontró, se guarda en el ***cache storage*** con la instrucción ***cache.put*** que contiene dos parámetros. El primero es la solicitud y la segunda es la información que contiene.

Para concluir se declara el evento ***respondWith()*** para ejecutar las instrucciones del código.

Una de las desventajas de esta estrategia, es al ejecutar el código combina la aplicación de ***Shell*** con recursos dinámicos. Esto quiere decir que recursos que pueden llegar a ser obsoletos en un futuro se quedan guardados junto con los recursos que constantemente se usan para la aplicación. Consecuentemente esto puede llegar hacer un mal desempeño de la aplicación web. Por lo tanto el código anterior se optimiza de la siguiente manera.

Se deben de crear tres tipos de cache para almacenar la información, el primero de ellos debe ser un ***cache dinámico****,* en el se almacenarán los recursos que la aplicación esté consultando constantemente en la web.

El segundo es un ***cache estático***, en él se almacenarán los recursos que no se llegan a consultar en la web y son modificados por el desarrollador de la aplicación. Por ejemplo los estilos, las imágenes estáticas, los archivos de Javascript y HTML.

Por último, un ***cache inmutable***. En él se almacenará las peticiones a recursos que no cambian a un futuro, por ejemplo la petición a librerías de diseño cómo ***bootstrap***.

Para implementar esta optimización se crean tres constantes, cada una de las con el tipo de cache a almacenar. El **cache estático** e **inmutable** se declaran cuando se inicia la instalación del service worker y **dinámico** se declara en las estrategias de cache cuando se envía alguna petición a la web.

El código de está optimización es el siguiente:

```
const cache_static = 'static-v1';
const cache_dynamic = 'dinamic-v1';
const cache_inmutable = 'inmutable-v1';

self.addEventListener("install", event => {

    const cacheSave = caches.open(cache_static)
        .then(cache => {

        return cache.addAll([
            '/index.html',
            '/css/styles.css',
            '/img/fes.png',
            '/js/app.js',
```

```
            ]);
        });

        const cacheInmutable = caches.open(cache_inmutable)
            .then(cache =>
cache.add(''https://bootstrapcdn.com/4.4.1))

        event.waitUntil(Promise.all([cacheSave, cacheInmutable  ]));
});

self.addEventListener("fetch", event => {

        const respuesta = caches.match(event.request)
            .then(res => {

            return fetch(event.request).then(newResp => {

            caches.open(cache_dynamic).then(cache => {

                cache.put(event.request, newResp);
                });
                return newResp.clone();
            });
        });
        event.respondWith(respuesta);
});
```

De esta manera se crearán tres caches, como se muestra en la siguiente imagen 3.4.3.2

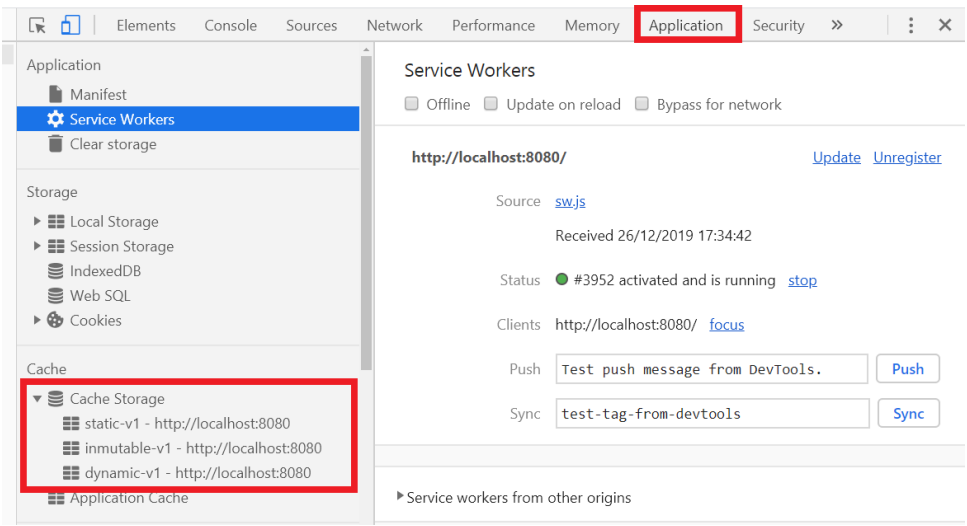

Imagen 3.4.3.1 Cache estatico, dinamico e inmutable

Cada vez que se haga alguna petición a la web se almacenarán los recursos dentro del cache dinámico. Sin embargo algunos navegadores tienen un límite de almacenamiento, por lo tanto el cache debe ser restaurado cada vez que se haga alguna petición. De esta manera se optimizará la aplicación web.

Para limpiar el cache se sigue en siguiente código:

```
function limpiarCache(cacheName, Items) {

    caches.open(cacheName).then(cache => {
        cache.keys().then(keys => {
            if(keys.length > Items){
                cache.delete(keys[0]).then(limpiarCache(cach
eName,items))
            }
        });
    });
}
```

El código anterior se inicia una función con el nombre de ***limpiarCache**,* dentro de ella se declaran dos atributos; el nombre del cache y el número de items que solo se guardaran en el cache dinámico. Posteriormente se inicia un método ***open ()*** que recibe el nombre del cache, después entra un método ***keys*** que son los registros que hay dentro del cache dinámico. Finalmente se debe comprobar si el límite de los registros alcanzó un máximo, en caso de ser así este limpiara el cache con el método ***delete()***.

Está función debe ser inicializada desde que se envía alguna solicitud a la web, de tal forma que cada vez que se inicie un nuevo registro se limpia el cache.

3.4.4 ESTRATEGIA DE RED PRIMERO

En esta estrategia consiste en enviar una petición a la web primero para obtener los recursos para la aplicación, de lo contrario si no existe ese recurso en la web intenta verificar si en el cache existe alguno relacionado.

En la imagen 3.4.4.1 se muestra el procedimiento de esta estrategia cache.

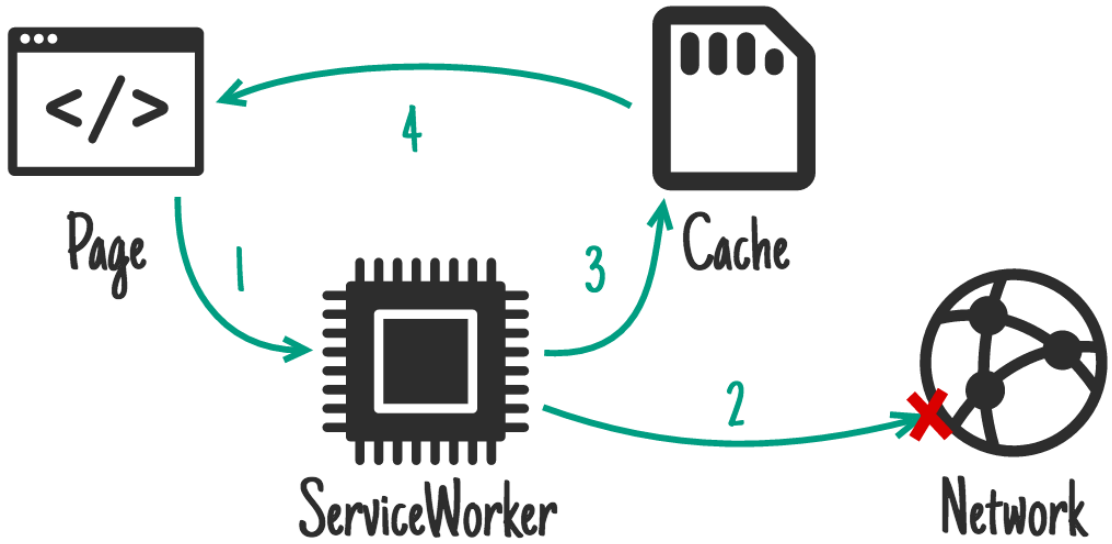

Imagen 3.4.4.1 Estrategia de red primero y luego cache

En el siguiente código se muestra la implementación de esta estrategia:

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

    const respuesta = fetch(e.request).then(res => {
        if(!res) return caches.match(event.request);

        caches.open(dinamic-v1)
            .then(cache => {
            cache.put(event.request, res);

            limpiarCache(dinamic-v1, 50);
        });
        return res.clone();
    }).catch(err => {
        return caches.match(event.request)
    });

    e.respondWith(respuesta)
});
```

En el código anterior se envía una petición ***fetch*** a la web, enseguida se comprueba si la respuesta que se recibió de la petición no fue correcta, se envía la petición al cache para buscar el recurso.

Por otro lado si la respuesta fue correcta se almacena en el cache la información obtenida de la web, además se hace una limpieza para evitar llenar el almacenamiento.

Por último si la petición a la web falla o no existe alguna conexión a internet, se envía nuevamente la petición al ***cache*** para buscar algún recurso que se relacione.

El código anterior da como resultado lo que se muestra a continuación en la imagen 3.4.4.3.

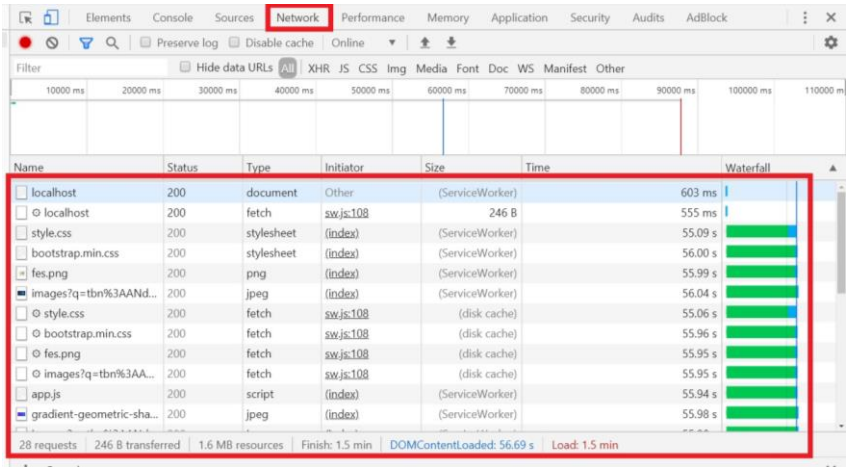

Imagen 3.4.4.2 Recursos almacenados en cache

Se observa que en la imagen 3.4.4.2 todas las peticiones han sido traídas de la web, posteriormente guardadas en el cache. En caso de no encontrar algún archivo en la web y en el cache mandara un error de petición.

Está estrategia siempre traerá la información más actualizada, por lo que lleva a dos problemas.

El primero problema que conlleva es la petición a la web, siempre se estará ejecutando, por lo tanto hay un consumo de datos constante, como resultado lleva a un segundo problema llamado velocidad de descarga.

Es una de las estrategias más lentas del cache por la razón de que siempre se tiene que hacer una descarga.

3.4.5 ESTRATEGIA DE CACHE MIENTRAS VALIDA

Está estrategia es utilizada cuando el rendimiento de la aplicación está en un estado crítico, por ejemplo cuando la conexión a internet es muy lenta.

Las actualizaciones que recibe la aplicación en esta estrategia siempre estarán una versión atrás, es decir que cualquier actualización que se haga en el archivo ***index o principal***, se observa exactamente igual, de manera que solo extrae la información que está guardada en el cache y posteriormente valida que la petición sea correcta. Cuando se haga una nueva recarga del sitio está hará una nueva petición al recurso desde la web, por lo tanto el usuario podrá consultar la información más actualizada.

Este proceso se ilustra en la imagen 3.4.5.1.

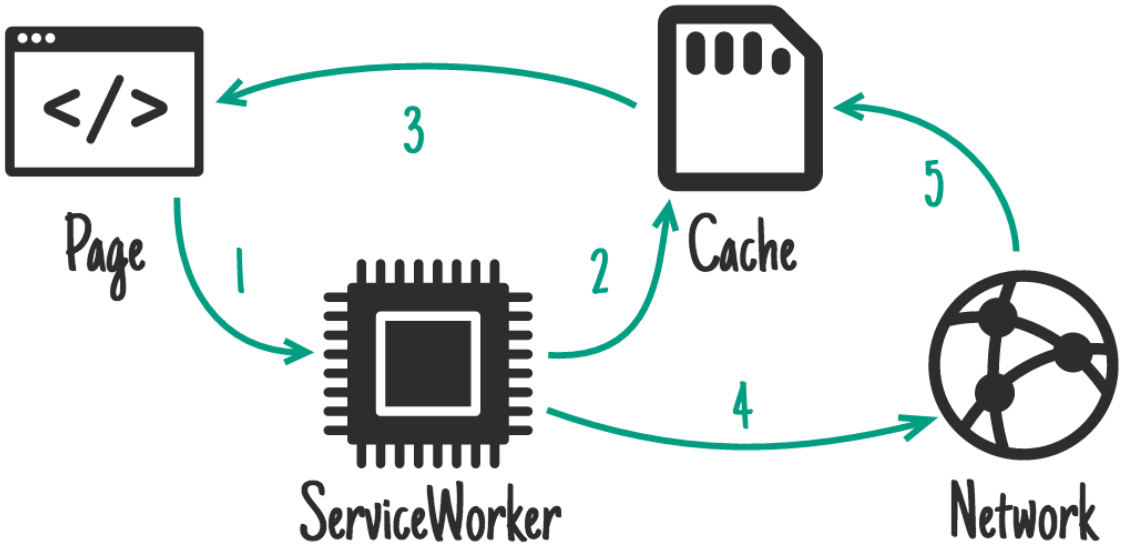

Imagen 3.4.5.1 Estrategia del cache mientras válida

Para implementar esta estrategia se sigue el siguiente código:

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

        const respuesta = caches.open('static-v1').then(cache => {
                fetch(e.request).then(newResp) =>
                        cache.put(e.request, newResp)
                return  cache.match(e.request);
        });
        e.respondWith(respuesta)
});
```

Se puede observar que en el código anterior se inicia recibiendo la información del cache estático como resultado de que la información que se va extraer únicamente es la que está guardada en el cache, no hay nada dinámico.

Posteriormente se actualiza la información y se recibe una nueva petición de la web. Si se actualiza algún archivo de diseño o el ***index***, está no hará ningún cómo resultado de traer primero la información del cache y a su vez actualizarla.

Está estrategia es bastante útil cuando el rendimiento de la aplicación web es crítico y se tiene que presentar lo más rápido posible al usuario para descargar actualizaciones más recientes.

3.4.6 ESTRATEGIA DE SOLO RED

Está estrategia es la más utilizada en la mayoría de las aplicaciones web. Todo es buscado a través de una petición a la web y presentados al usuario.

El proceso de esta estrategia se presenta en la siguiente imagen 3.4.6.1.

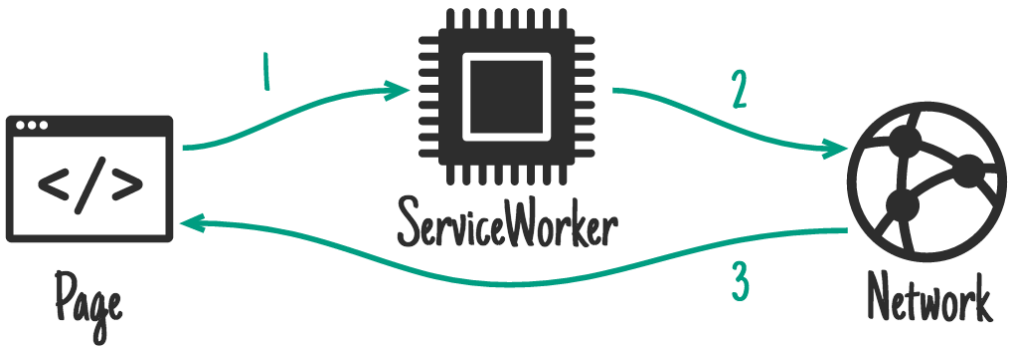

Imagen 3.4.6.1 Estrategia de solo red

La implementación es bastante sencilla, para eso se muestra el siguiente código:

```
self.addEventListener("fetch", event => {
        event.respondWith(
            fetch(event.request)
        )
});
```

Ninguna de la información se almacena en el cache.

3.4.7 ACTUALIZACIÓN Y OPTIMIZACIÓN DEL CACHE

Cada vez que la aplicación web traiga alguna petición para actualizar estéticamente la aplicación o nuevas funcionalidades se debe crear una nueva versión del caché.
Si el cache de una versión anterior no es eliminado, este se mantendrá a lo largo que la aplicación esté funcionando y consecuentemente habrá errores.

Por ejemplo si una aplicación recibe una actualización en la imagen principal, el usuario no podrá verla porque se mostrará la imagen principal de la versión anterior al cache.

Para actualizar y borrar la versión anterior al cache se ilustra en el siguiente código:

```
self.addEventListener("activate", event => {
        const respuesta = caches.keys().then(keys => {
            keys.forEach(key => {

                if(key !== CacheStatic &&
            key.includes('static')) {
                    return caches.delete(key);
                }
            })
        })
});
```

Este proceso se produce durante el evento de activación del ***service worker***, por ello dentro del evento se indica que se debe buscar una llave que traerá la versión anterior cuando se detecte una nueva versión de ese mismo. Si hay alguna actualización, la versión vieja del cache se elimina y es sustituida por la nueva versión actualizada.

3.5 DESPLIEGUES EN DISPOSITIVOS MÓVILES

Para desplegar una aplicación web progresiva se necesita una serie de configuración dentro del proyecto de la aplicación. En primer lugar se necesita crear un archivo manifiesto, este archivo permitirá la personalización de la aplicación a cómo lo verá el usuario final.

Matt Gaunt y Paul Kinlan (2018) explican el proceso del archivo manifiesto en su artículo y lo definen como "El manifiesto de las App web es un archivo JSON simple que permite que tú, el desarrollador, puedas controlar cómo se muestra tu App al usuario en áreas donde normalmente ven App nativas (por ejemplo, la pantalla de inicio de un dispositivo móvil), además de indicar lo que el usuario puede iniciar y definir su apariencia al iniciarse.

Los manifiestos de las App web permiten guardar un marcador de sitio en la pantalla de inicio de un dispositivo. Aunque los manifiestos de App web pueden usarse en cualquier sitio, son necesarios para las App web progresivas"

El archivo manifiesto puede llevar cualquier nombre, sin embargo la mayoría lo usa cómo ***manifest.json***. Este archivo se creará en la raíz del proyecto.

Para ilustrar el archivo manifiesto se puede ver el siguiente código:

```
{
        "short_name": "Mi PWA",
        "name": "Está es una PWA",
        "start_url": "index.html",
        background_color: "#3498db",
        "theme_color": "#3498db",
        "display": "standalone",
        "orientation": "portrait",
        "icons": [{
                "src":"img/icons/icon.png",
                "type": "img/png",
                "sizes": "72x72"
            }
            {
                "src":"img/icons/icon.png",
                "type": "img/png",
                "sizes": "72x72"
            }
            ]
}
```

Cómo se puede observar en el código anterior hay parámetros del archivo manifiesto que se deben especificar. A continuación se muestra una lista de los parámetros que se escribieron en el código anterior

- *short_name:* Es el texto que se muestra en la pantalla del usuario
- *name*: Es el texto que se usa en la instalación de la aplicación

- *start_url*: Es la página de inicio de la aplicación
- *background_color*: Color de fondo que usará la aplicación, este debe estar en hexadecimal
- *theme_color*: Color de la barra de navegación
- display: Se indica si se requiere mostrar la barra de navegación o los botones. Se tienen dos opciones:
    - *browser*: Se mostrará cómo si fuera una página web
    - *standalone*: Se muestra cómo si fuera una aplicación móvil
- *orientation*: Se especifica la orientación de la página , ya sea que solo se muestre de forma vertical u horizontal

Una vez que se ha creado el manifiesto de la aplicacion, despues se debe agregar una etiqueta link a todas las páginas que conformen la aplicación web, como se muestra a continuación:

<link rel=”manifest” href=”manifest.json”>

Para los dispositivos Android únicamente se requiere que se añada la siguiente línea de código en el archivo ***index.html*** indicando el mismo color que se añadió en el archivo manifiesto, esto cambiará el color de la barra de herramientas:

<meta name=”theme-color” content=”#3498db”>

Para dispositivos IOS la configuración es ligeramente diferente por razón que algunos dispositivos aún no tienen soporte para aplicaciones web progresivas.

La configuración en dispositivos IOS es la siguiente.

En primera, se añade la siguiente línea de código en el archivo index.html:

<meta name=”apple-mobile-web-App-capable” content=”yes”>

Después se añadirá la siguiente línea de código para especificar el icono de la aplicación

<link res=”apple-touch-icon” href=”icon.png”>

Algunos dispositivos IOS usan diferentes tamaños de para los iconos en la pantalla de inicio. Para especificar un icono para algún tamaño específico se usa el atributo *size.*

Posteriormente es importante indicar un ***splash screen****,* que no es más que una imagen que se mostrará cuando se inicie la aplicación, esto hará que la aplicación web progresiva sea más parecida a una nativa. Para ello se escribe el siguiente código en el archivo ***index.html.***

<link rel=”apple-touch-startup-image” href=”splash.png”>

Para personalizar la barra de estado en IOS se debe usar la siguiente etiqueta que se pondrá en el mismo archivo index.html.

<meta name=”apple-mobile-web-App-status-bar-style” content=”default”>

Existen solo tres maneras de personalizar la barra de estado, solo basta con cambiar el valor del atributo *content.*

Los valores que se le pueden dar son los siguientes:

- *default*: La barra de búsqueda se queda por defecto en color blanco y texto negro.
- *black*: La barra de búsqueda cambia a color negro con textos blancos.
- *black-translucent*: La barra de búsqueda es transparente y toma como color el mismo fondo del cuerpo de la aplicación.

EL resultado de estas tres opciones se muestra en la imagen 3.5.1:

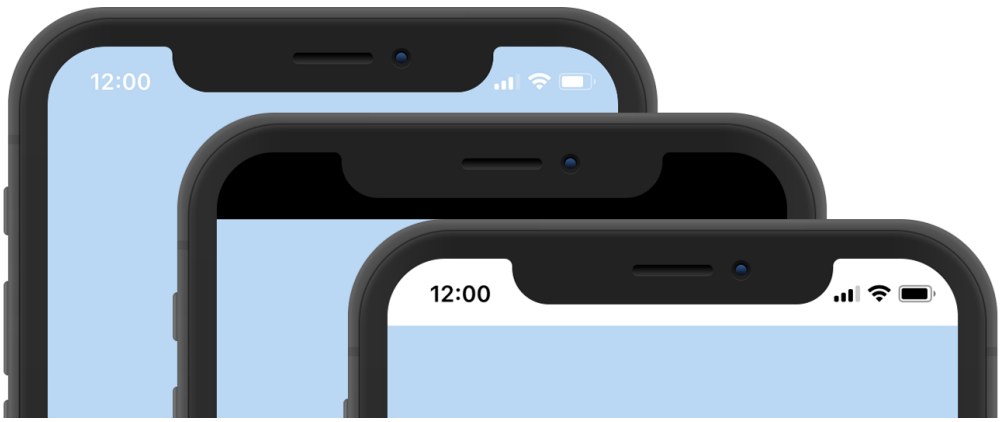

Imagen 3.5.1 Barra de estado con las configuraciones
https://miro.medium.com/max/1289/1*S3IV89JuxU07oiY6JjEkkA.png

Por último se deben desactivar algunas interacciones que el navegador permite, como por ejemplo la selección del texto, el resaltado en títulos, el despliegue de submenús de llamada. Como resultado la interacción con la aplicación web progresiva será una mejor experiencia parecida a una aplicación nativa

Para desactivar estas interacciones se coloca el siguiente código en el archivo de estilos *CSS* dentro del identificador del cuerpo ***body***

```
body {
        -webkit-user-select:none;
        -webkit-tap-highlight-color: transparent;
        -webkit-touch-callout:none:
}
```

Una vez configurada la aplicación web, el navegador detectara el manifest el cual se podrá observar desde las opciones de desarrollo.

Para ilustrarlo se muestra la siguiente imagen 3.5.2:

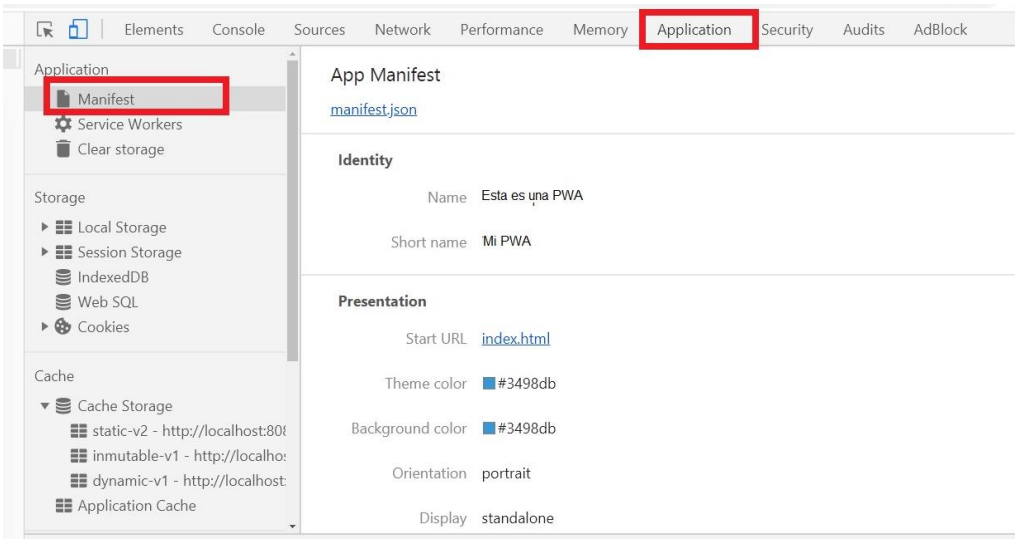

Imagen 3.5.2 Opción Manifest en navegador

Hay diferentes maneras de probar la aplicación en un dispositivo, la ideal es desplegar la aplicación en un sitio web con protocolo https. Otra manera es de manera local.

Para desplegar una aplicación de manera local, en el navegador en las opciones de desarrollo se muestran herramientas para desplegarla en un dispositivo remoto.

Estas herramientas se encuentran en la opciones con los tres puntos, seguido se observa la opción **más herramientas** donde desplegará un submenú y después habrá que seleccionar la opción **Dispositivos remotos**.

Así como se muestra en la imagen 3.5.3:

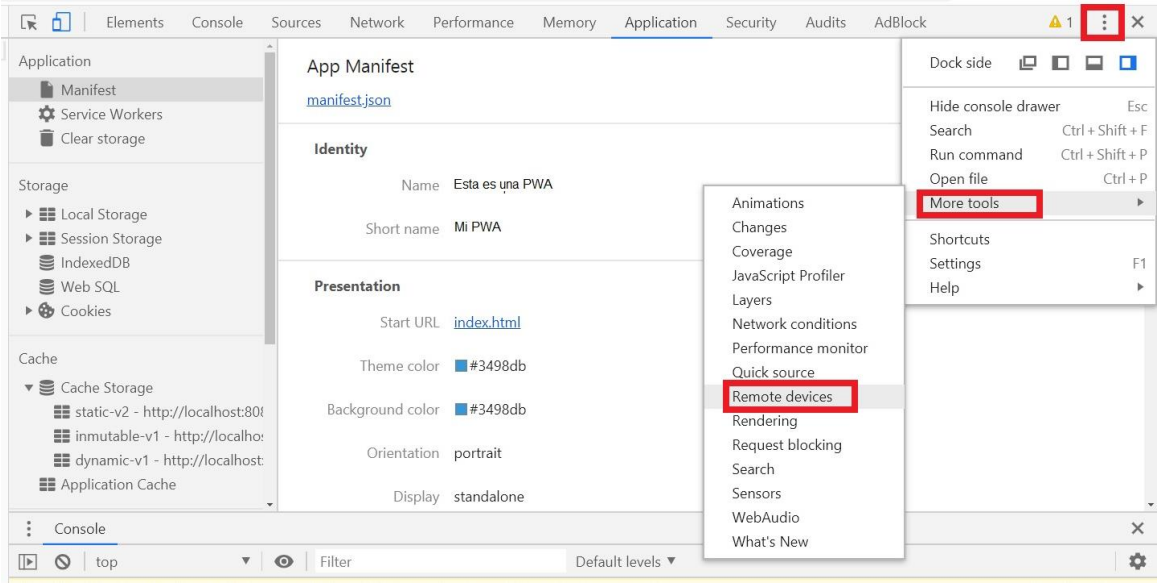

Imagen 3.5.3 Herramientas para dispositivos remotos

Posteriormente se mostrará la siguiente pantalla como se observa a continuación en la imagen 3.5.4:

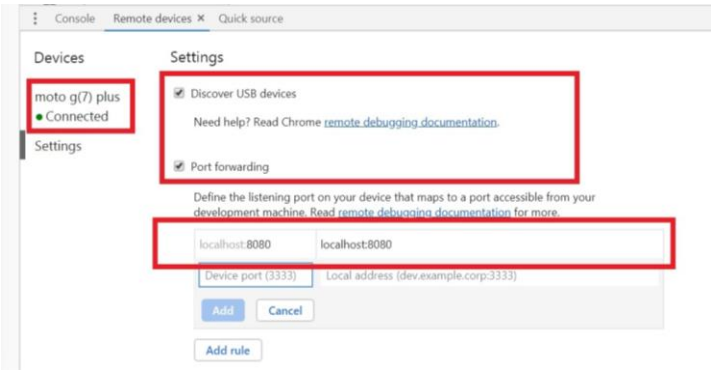

Imagen 3.5.4 Configuración de dispositivos remotos

En la configuración para dispositivos remotos se deben activar las dos opciones que aparecen.

La primera opción permitirá conectar el dispositivo por USB, posteriormente se observará la aplicación desde el dispositivo móvil.

La segunda opción permitirá desplegar la aplicación por web, sin necesidad de conectar el dispositivo físicamente, únicamente se debe colocar la dirección en el puerto que fue ejecutada la aplicación.

Unidad 4 DESARROLLO DE UNA APLICACIÓN PROGRESIVA EMPLEANDO TECNOLOGÍAS DE PROGRAMACIÓN WEB

4.1 INTRODUCCIÓ AL DESARROLLO

El siguiente proyecto se desarrollará con el objetivo de aplicar los conocimientos previamente explicados en los capítulos anteriores.

La aplicación consistirá de una página diseñada para dispositivos móviles que mostrará tarjetas de todas las carreras que se encuentran en la Facultad de Estudios Superiores Aragón. Estas tarjetas se encontrarán con un diseño de carrusel para que el usuario cuando abra la aplicación pueda desplazarse de izquierda a derecha. Cada tarjeta tendrá el logotipo de cada carrera y al presionarlas abrirá una modal o ventana emergente que se podrá observar la misión u objetivo general de cada una de ellas.

Tendrá todos los estándares que consta una aplicación web progresiva lo que significa que actuará como una aplicación nativa instalada en el dispositivo y funcionalidad offline cuando no se tenga conexión a internet.

Las tecnologías que se usarán para el desarrollo del proyecto son las siguientes:

HTML
CSS
Javascript

4.2 ESTRUCTURA DE LA APLICACIÓN

El proyecto está formado por 2 carpetas y un archivo principal. La primera carpeta tendrá el nombre img y dentro de ella se tendrá tres carpetas que para cada una de las áreas en oferta académica que se ofrece en la FES Aragón con el objetivo de tener una estructura ordenada, finalmente cada una de esas carpetas tendrá los logotipos de cada licenciatura e ingeniería con la que se cuenta.

La segunda carpeta tendrá el nombre de CSS, dentro de ella se almacenarán los estilos que se darán a la página.

El archivo principal lleva el nombre de index.html y dentro de él se escribirá la estructura del proyecto para la parte visual hacia el usuario.

La estructura finalmente se muestra en la siguiente imagen 4.2.1:

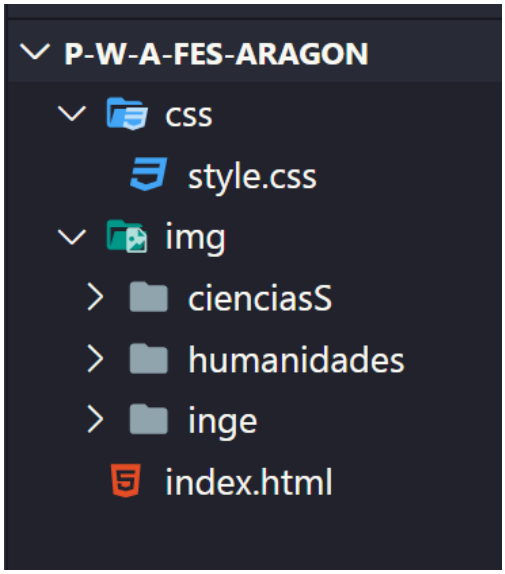

Imagen 4.2.1 Estructura inicial

El siguiente paso en el desarrollo es darle diseño a la aplicación, empezando por estructurarlo en el archivo principal index.html el cual llevará las siguientes líneas de código dentro del cuerpo del documento.

La cabecera se define en una etiqueta header que tendrá el título de Bienvenido dentro de otra etiqueta **<h2>**, como se muestra en el siguiente ejemplo 1:

Ejemplo 1

```
<header>
        <h1 class=””titulo-page>
                Bienvenido
        </h1>
</header>
```

Para dar estilo a la cabecera se da referencia a la clase de la etiqueta <h1> en el archivo de estilos CSS.
El estilo que llevará la cabecera será la siguiente:

```
.titulo-page {
        background: #F5A705;
        color: white;
        position: relative
}
```

Las propiedades que se da son un fondo de color naranja escrito en hexadecimal, con letra de color blanco y con una posición relativa, el resultado de este diseño de cabecera se muestra a continuación en la imagen 4.2.2:

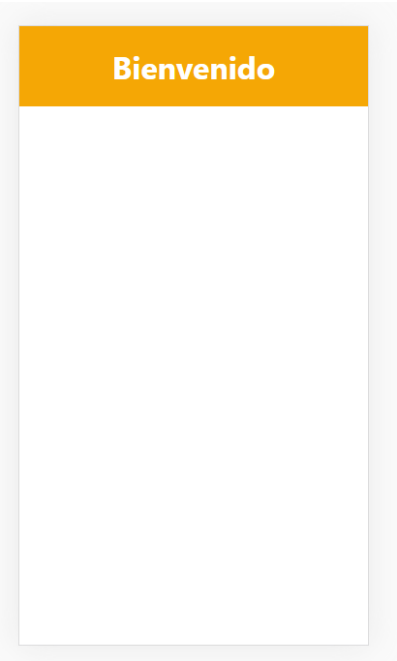

Imagen 4.2.2 Cabecera de la aplicación

A continuación para construir un carrusel de imágenes dentro de la aplicación se necesita de varios elementos, sin embargo la funcionalidad se dará en archivo de estilos. La estructura del carrusel para la sección de imágenes es la siguiente:

```
<div class="carrusel">
    <h1>Ingenierías</h1>
    <section class="carousel">
        <div class="carousel__container">
            <div class="carousel-item">
                <a href="#modal-ico" id="show-modal">
                    <img class="carousel-item__img" src="img/inge/ico.jpeg">
                </a>
            </div>
            <div class="carousel-item">
```

```
                    <a href="#modal-iee" id="show-modal">
                        <img class="carousel-item__img" src="img/inge/iee.jpg">
                    </a>
                </div>
                <div class="carousel-item">
                    <a href="#modal-iciv" id="show-modal">
                        <img class="carousel-item__img"
        src="img/inge/iciv.jpeg">
                    </a>
                </div>
                <div class="carousel-item">
                    <a href="#modal-ime" id="show-modal">
                        <img class="carousel-item__img" src="img/inge/ime.jpeg"
        >
                    </a>
                </div>
            </div>
        </section>
    </div>
```

El anterior código se tiene una etiqueta ** ** principal que almacena todo el carrusel, dentro de ella se define un título que llevara el nombre de la área. Dentro de la etiqueta **<section>** contendrá dos diferentes etiquetas ** ** que su principal función será dar el movimiento al carrusel. Finalmente dentro de las etiquetas de enlaces **<a>** se especifica la imagen de cada una de las carreras del área.

Dentro del archivo de estilos css se escribe el siguiente código para dar estilo al carrusel:

```
.carrusel {
    overflow: hidden;
    min-height: calc(50vh);
    background-image: linear-gradient(#006E90, #74b8d4);
}

.carrusel h1 {
    margin: 0;
    padding: 10px;
    color: #ffffff;
    text-align: center;
}
```

```
.carousel {
    width: 100%;
    overflow-y: scroll;
    position: relative;
    padding-left: 30px;
}

.carousel__container {
    margin: 10px 0px;
    white-space: nowrap;
    padding-bottom: 10px;
}

.carousel-item {
    width: 200px;
    height: 250px;
    cursor: pointer;
    overflow: hidden;
    position: relative;
    margin-right: 10px;
    border-radius: 20px;
    display: inline-block;
    transition: 450ms all;
    background-color: #0D3B66;
    transform-origin: center left;
}

.carousel-item__img {
    width: 200px;
    height: 250px;
    object-fit: cover;
}
```

Entre todos atributos que se dan para el diseño del carrusel, los más importantes son **overflow: hidden** que permite el desplazamiento sin que la pantalla se salga de las coordenadas, **overflow-y: scroll** que permite el movimiento de izquierda a derecha. El resto solo da estilo a las fuentes, tamaños y colores para el carrusel.

El resultado se muestra a continuación en la imagen 4.2.3:

Imagen 4.2.3 Primer carrusel de imágenes

Para las dos áreas restantes se usa el mismo carrusel con el mismo diseño, lo que da la siguiente imagen 4.2.4:

Imagen 4.2.4 Carrusel de imágenes

Lo siguiente es crear un modal que aparece con la información de cada carrera, para ello se define de la siguiente manera:

```
<aside id="modal-ico" class="modal">
    <div class="content-modal">
        <header>
            <a href="#" class="close-modal">X</a>
            <h2>Ingenieria en computacion</h2>
        </header>
        <article>
            <h3>Misión</h3>
            <p> Nuestra misión es formar profesionales </p>
        </article>
    </div>
</aside>
```

Se usa la etiqueta **<asisde>** para crear este modal, dentro del archivo de estilos se le dará el siguiente estilo para que aparezca a la hora de presionar alguna de las tarjetas del carrusel.

```
h2 {
    color: #0D3B66;
    padding: 1rem;
}

#show-modal {
    text-align: center;
    color: white;
    font-weight: 500;
    font-size: 1.5rem;
    text-decoration: none;
}

#show-modal:hover {
    cursor: pointer;
    /* background: rgb(72, 126, 29); */
}

.modal {
    position: fixed;
    top: -100vh;
    left: 0;
```

```
    z-index: 9999999;
    background: rgba(0, 0, 0, 0.75);
    width: 100vw;
    height: 100vh;
    opacity: 0;
    transition: opacity 0.35s ease;
}

.modal .content-modal {
    width: 100%;
    max-width: 500px;
    position: fixed;
    left: 50%;
    top: -100vh;
    transition: top 0.35s ease;
    margin-left: -250px;
    background: white;
    box-shadow: 0 1px 2px rgba(0, 0, 0, 0.1);
    border-radius: 2px;
    z-index: 9999999;
}

.modal h2 {
    padding: 1.2rem;
    text-align: center;
    color: #000000;
    margin: 0;
}

.modal h3 {
    text-align: start;
    padding-left: 15px;
    padding-top: 15px;
}

.modal p {
    text-align: justify;
    padding-left: 8px;
    padding-right: 8px;
}

.modal article {
```

```
    height: 350px;
    background: #b8b8b8;
}

.close-modal {
    color: #e74c3c;
    position: absolute;
    top: 0.2em;
    right: 0.375em;
    margin: 0;
    padding: 5px;
    font-weight: bold;
    font-size: 1.5em;
    text-decoration: none;
}

.modal a:hover {
    color: #c0392b;
}

.modal:target {
    opacity: 1;
    top: 0;
}

.modal .btn-close-modal {
    position: absolute;
    left: 0;
    width: 100%;
    height: 100%;
    z-index: 9999999;
}

.modal:target .content-modal {
    top: 50px;
    transition: top 0.35s ease;
}
```

Cada uno de los atributos le dará estilo al modal, esto se muestra a continuación en la siguiente imagen 4.2.5:

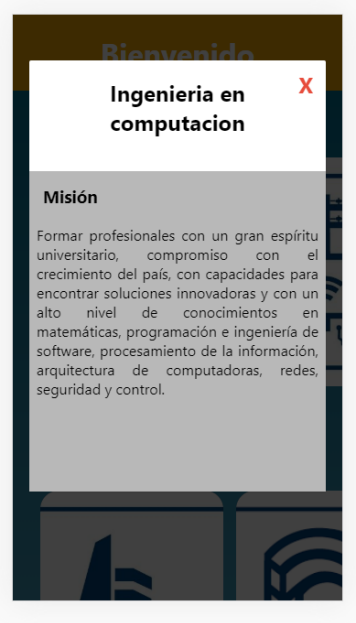

Imagen 4.2.5 Modal

4.3 CONFIGURACIÓN DEL SERVICE WORKER

Es fundamental dar una estructura adecuada al hacer la configuración inicial en el archivo donde se encuentra el service worker. Para esta primera configuración se establecerán dos tipos de cache.

El primero será el cache estático que será el principal para la aplicación y se almacenarán los documentos creados por el desarrollador.

El segundo es el cache inmutable que tendrá como principal función almacenar los documentos desarrollados por librerías externas o de terceros.

Cómo primer paso se crean dos archivos Javascript, el primero con nombre *app.js* que tendrá la función registrar al service worker llamando al archivo *sw.js* que es donde estará toda la configuración del mismo *service worker*.

El segundo archivo lleva el nombre de *sw.js* que tendrá las configuraciones y las estrategias de cache para cuando la aplicación web esté en producción. Finalmente la estructura del proyecto se muestra en la siguiente imagen 4.3.1

Imagen 4.3.1 Estructura final

Dentro del archivo *app.js* contendrá las siguientes líneas de código:

```
if(navigator.serviceWorker) {
        navigator.ServiceWorker.register('/sw.js')
}
```

Para el archivo *sw.js* se tendrán las siguientes configuraciones:

Cómo primera configuración se crean tres constantes para los diferentes caches que estarán almacenados en la aplicación, estos son:

Cache estático
Cache dinámico
Cache inmutable

Enseguida se establecen los archivos que pertenecen a la aplicación shell, algunos de ellos son las imágenes, el archivo principal de la aplicación y el archivo de javascript app.js. También se define la aplicación de shell inmutable que son las librerías de terceros o externas, por ejemplo las fuentes para los párrafos y títulos, las animaciones e iconos.

Por otro lado se configura la instalación del service worker llamando a un evento listener y un segundo evento de activación. Está configuracion se muestra en el siguiente ejemplo 1:

Ejemplo 1

```
self.addEventListener('install', event => {
        event.waitUntil();
})
```

```
self.addEventListener('activate', event => {
        event.waitUntil();
})
```

En el evento de activación del *service worker* se hace una configuración adicional para que el caché almacenado se elimine cada vez que se modifique algún documento. Está segunda configuracion se muestra en el siguiente ejemplo 2:

Ejemplo 2

```
const respuesta = caches.keys().then(keys => {
        keys.forEach(key => {
                if(key !== STATIC_CACHE && includes('static')) {
                        return caches.delete(key);
                }
        });
});
```

Por último hay que recordar que para ejecutar la aplicación se debe hacer sobre un servidor, para está ocasión se hará con ayuda de la librería http-server de node js en el puerto 8080.

Al ejecutar la aplicación se puede observar que están almacenados los archivos en el cache del ordenador, esto muestra a continuación en la imagen 4.3.2

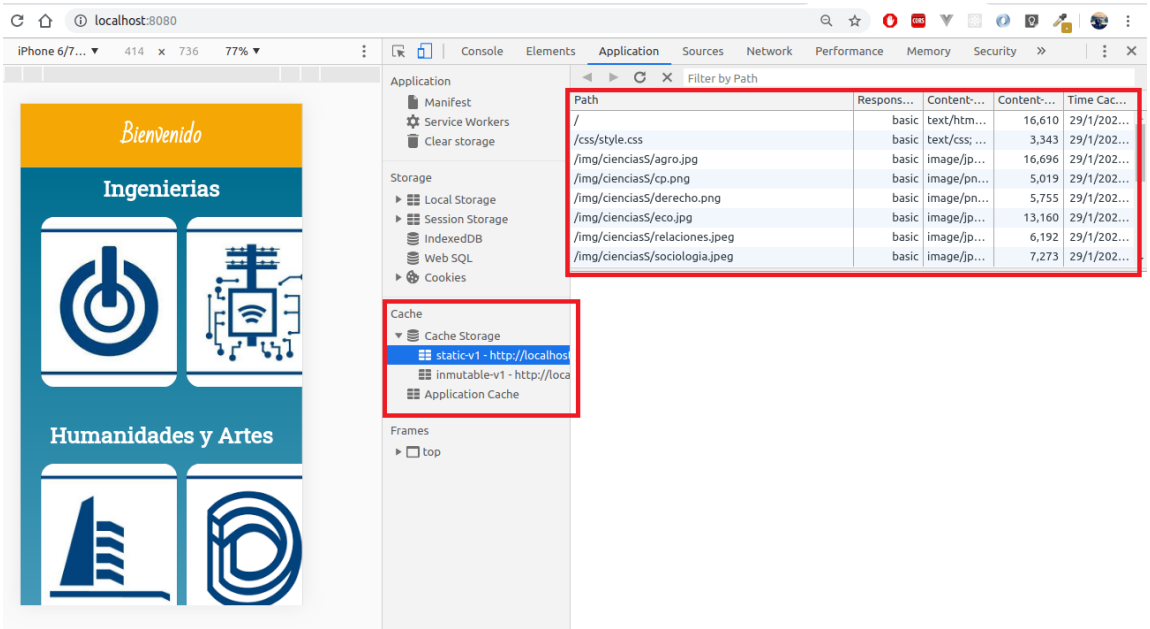

Imagen 4.3.2 Archivos almacenados en cache estático

4.4 IMPLEMENTACIÓN DE LA ESTRATEGIA DE CACHE

La estrategia que se utilizara para este proyecto sea la que haga una petición a la red red primero y después almacene esa información en el cache, también denominada estrategia de caché primero y luego red.

Su implementación es la siguiente:

Primero se crea un evento listener llamando a la función fetch seguido de una respuesta, cómo se puede observar en el siguiente ejemplo 1:

Ejemplo 1

```
self.addEventListener('fetch', e => {
        e.respondWith(e)
});
```

Para este proyecto se manda llamar una respuesta de la información que traerá de la web y así almacenarla en un caché dinámico, esto es únicamente para las librerías externas como las fuentes de letras o los iconos. Es posible que el evento no contenga información, por lo que ocurrirá un error como el que se muestra a continuación en la imagen 4.3.1

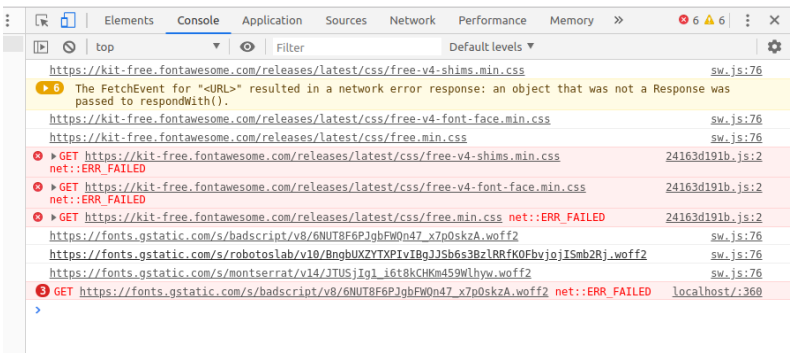

Imagen 4.4.1 Error en las peticiones externas

Para controlar el error mostrado en la imagen 4.4.1 se crea un archivo auxiliar para el service worker. Para este proyecto el archivo lleva el nombre de *sw-utils.js.* Este archivo contiene una función que actualiza el cache dinámico y recibe tres parámetros.

El primer parámetro es el cache dinámico, el segundo es un requerimiento y como tercer parámetro es la respuesta que se recibe.

Cuando se haga alguna petición y detecte que hay nueva información, esta función almacenará estos datos en el cache dinámico, sin embargo si no hay información nueva no se ejecutará la función.

El código que lleva el archivo sw-utils.js es el siguiente:

```
function actualizarCacheDinamico(dynamicCache, req, res) {
        if(res.ok){
                return caches.open(dynamicCache).then(cache => {
                        cache.put(req, res.clone());
                        return res.clone();
                });
        } else {
                return res;
        }
}
```

Finalmente esta función se manda llamar en el archivo de configuración del service worker al momento de hacer la estrategia de cache. Para mandar llamar la función se escribe el siguiente código:

```
importScripts('js/sw-utils.js');
```

Está función es implementada en la estrategia de cache como se muestra a continuación en las siguientes líneas de código:

```
self.addEventListener('fetch', e => {
   const respuesta = caches.match(e.request).then(res => {

      if (res) {
         return res
      } else {
         return fetch(e.request).then(newRes => {
            return actualizaCacheDinamico(DYNAMIC_CACHE, e.request,
newRes);
         });
      }
   });

   e.respondWith(respuesta)
});
```

Con está implementación evita que aparezca errores al enviar peticiones a librerías externas así como se mostró anteriormente en la en la imagen 4.4.1

4.5 CONFIGURACIÓN DEL ARCHIVO MANIFIESTO

El archivo manifest es un archivo con extensión json para configurar la aplicación al momento de abrirla desde el dispositivo, este archivo se crea en la raíz del proyecto como se muestra a continuación en la imagen 4.5.1.

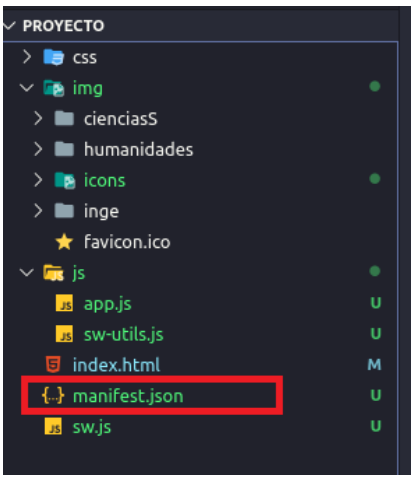

Imagen 4.5.1 Archivo manifest en la raíz del proyecto

Enseguida se crea una carpeta icons dentro de la carpeta de imágenes para almacenar los iconos para los diferentes dispositivos. Las dimensiones de los iconos usados para esta aplicación son los siguientes:

72x72 px
96x96 px
120x120px
144x144 px
152x152 px
192x192 px
310x310 px

La configuración que se usa en el archivo manifest es la siguiente:

Nombre de la aplicación: FESAPWA
Descripción: PWA informativa de FES Aragón
Archivo principal: index.html
Color de fondo en hexadecimal al iniciar la aplicación: #212d40
Tipo de barra de búsqueda: Sin mostrar barra de búsqueda
Orientación de la aplicación: Vertical

Enseguida se indica la ruta de cada uno de los iconos correspondientes. Después de conocer la configuración se declara en el archivo manifest cómo se muestra a continuación:

```
{
   "short_name": "FESAPWA",
   "name": "PWA informativa de FES Aragón",
   "start_url": "index.html",
   "background_color": "#212d40",
   "display": "standalone",
   "orientation": "portrait",
   "theme_color": "#212d40",
   "icons": [{
         "src": "img/icons/icon-72x72.png",
         "type": "image/png",
         "sizes": "72x72"
      },
      ….
   ]
}
```

Finalmente dentro del navegador se puede comprobar si la aplicación aceptó el archivo manifest, de lo contrario no mostrará nada.

Al aceptar el archivo manifest aparecerá la información como se muestra en la siguiente imagen 4.5.2

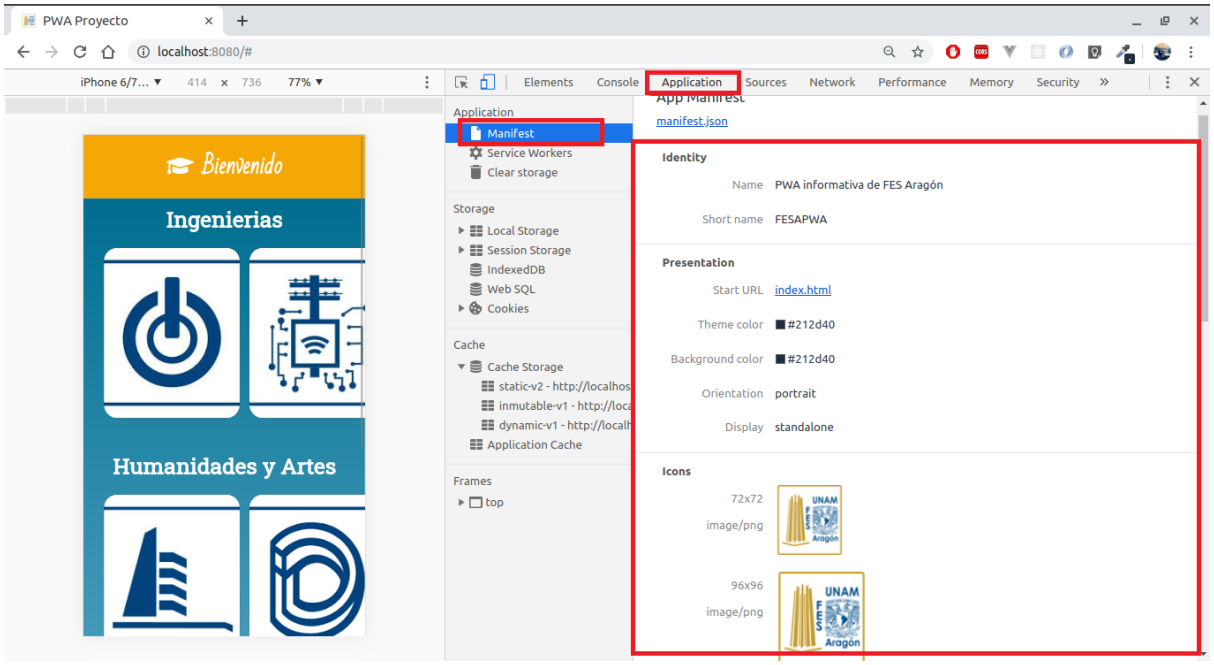

Imagen 4.5.2 Archivo manifest en las herramientas del navegador

4.6 EJECUTAR LA APLICACIÓN WEB PROGRESIVA EN UN DISPOSITIVO MÓVIL

Antes de lanzar a producción alguna aplicación se debe ejecutar y depurar en un dispositivo de prueba. Para ello el navegador Google Chrome permite ejecutar este tipo de aplicaciones bajo un servidor local en dispositivos reales.

En la imagen 4.6.1 se muestra en donde se encuentra la configuración anteriormente mencionada:

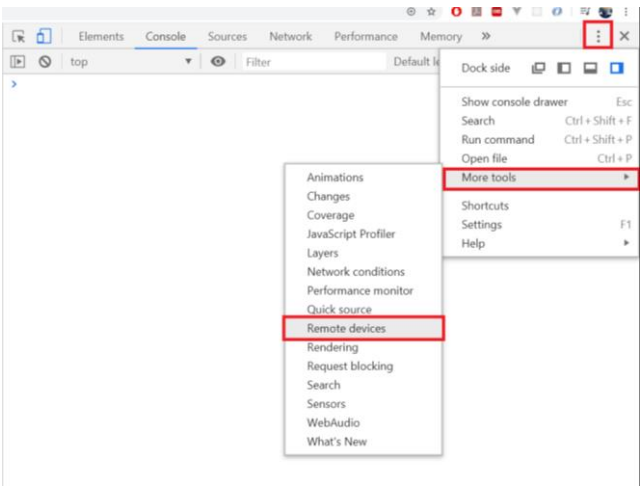

Imagen 4.6.1 Herramientas para desarrollo en dispositivos

Dentro de la configuración se muestra la opción para especificar el puerto para que él pueda ser desplegado al dispositivo móvil, además si el dispositivo móvil está conectado al ordenador, este lo de detectar así como se muestra en la imagen 4.6.2

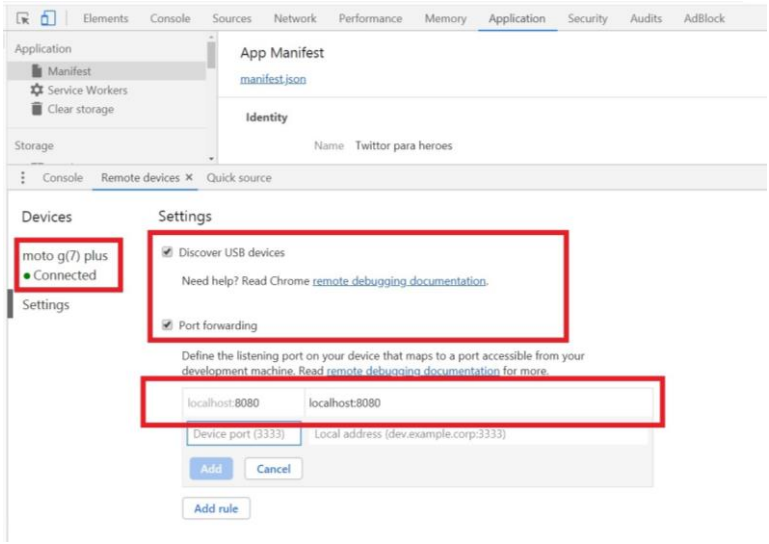

Imagen 4.6.2 Depuración en dispositivos

El dispositivo debe estar conectado mediante USB para que pueda visualizarse la aplicación, enseguida el navegador enviará una petición al dispositivo para que la aplicación puede visualizarse en el dispositivo móvil.

De igual importancia ya dentro de la aplicación se observará un mensaje para añadir la aplicación a la pantalla principal, así como se muestra en la imagen 4.6.3

Imagen 4.6.3 Aplicación web en dispositivo real

AL dar click en el mensaje emergente la aplicación se colocará en la pantalla principal del dispositivo, así como se muestra a continuación en la imagen 4.6.4

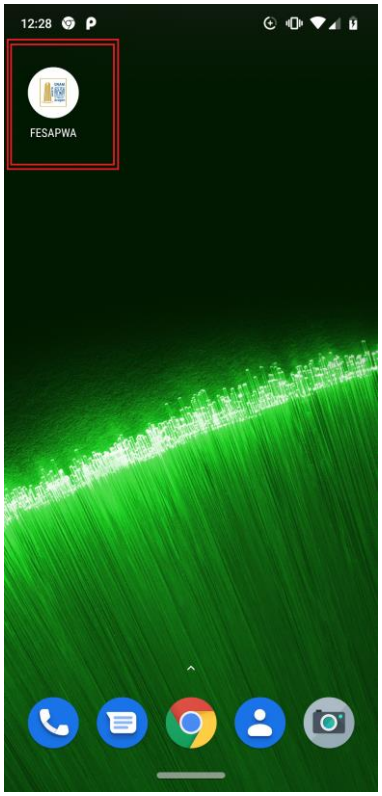

Imagen 4.6.4 Aplicación en la pantalla principal del dispositivo móvil

4.7 CONFIGURACIÓN PARA DESPLEGAR UNA APLICACIÓN CON AYUDA DE GITHUB PAGES

GitHub es una plataforma para el desarrollo colaborativo entre desarrolladores la cual provee herramientas para publicar páginas web o aplicaciones todo bajo protocolo HTTPS. Es importante tener una cuenta para poder usar estas herramientas.

A continuación en la imagen 4.7.1 se muestra la página principal.

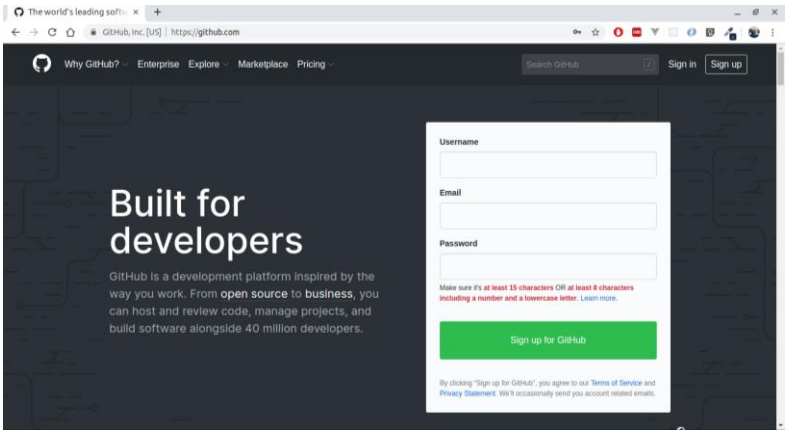

Imagen 4.7.1 Página principal de GitHub

Antes de iniciar un proyecto en GitHub es necesario inicializar un repositorio en la carpeta del proyecto. Para inicializarlo se escribe el siguiente comando desde la terminal del sistema operativo siempre estando en la ruta donde se creó el proyecto.

git init

Posteriormente se deben agregar los cambios a un repositorio local, para ello se escriben los siguientes dos comandos:

git add -A
git commit -m “Mensaje”

Enseguida en la página de GitHub se registra un nuevo proyecto indicando el nombre y una descripción del proyecto, así como se muestra en la siguiente imagen 4.7.2

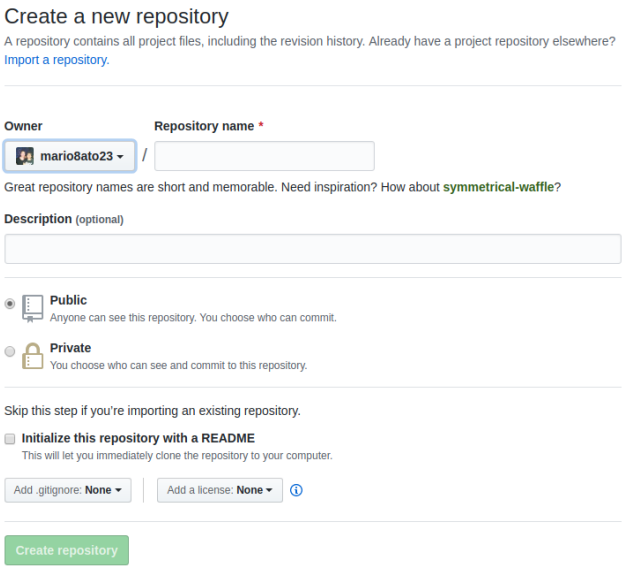

Imagen 4.7.2 Registro del proyecto en GitHub

Finalmente al finalizar el registro aparecerá la siguiente ventana como se muestra a continuación en la imagen 4.7.3

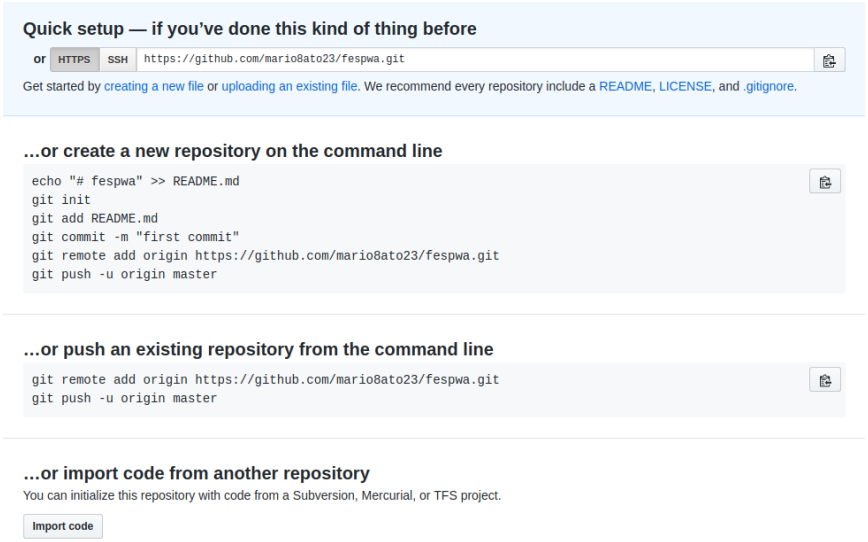

Imagen 4.7.3 Ventana para agregar el proyecto a GitHub

Posteriormente en la terminal del sistema operativo se colocan los siguientes comandos para subir el proyecto a GitHub.

git remote add origin https://github.com/usuario/nombreProyecto.git
git push -u origin master

Inmediatamente el código y las imágenes del proyecto estarán disponibles dentro del repositorio de GitHub, esto se puede observar en la siguiente imagen 4.7.4

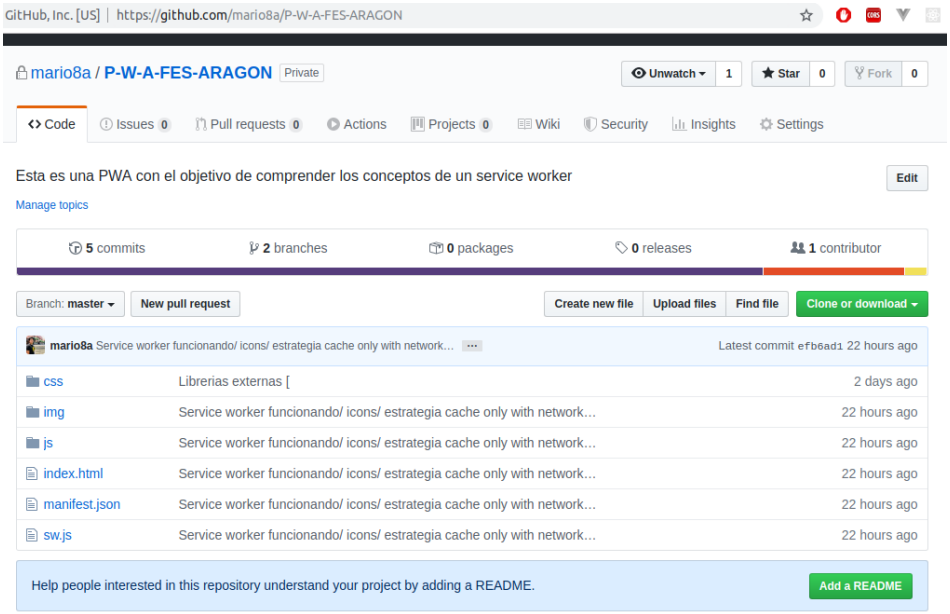

Imagen 4.7.4 Repositorio de GitHub del proyecto

Lo siguiente es habilitar GitHub Pages para desplegar el proyecto en una página web, para ello esta opción se encuentra en la opción de herramientas, esta se encuentra en el lado derecho de la página, así como se muestra en la siguiente imagen 4.7.5

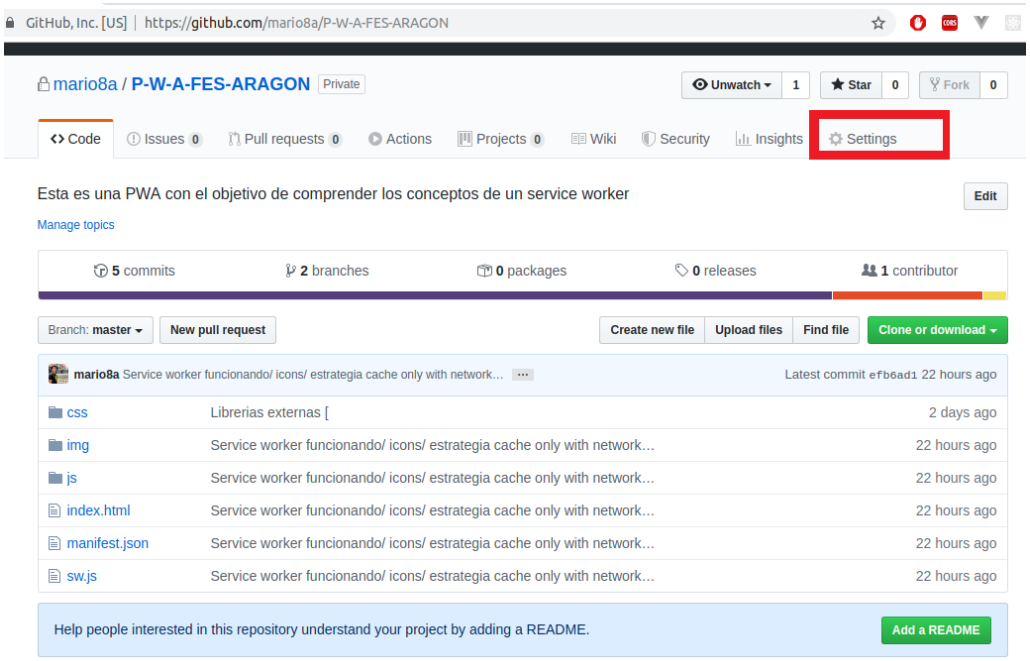

Imagen 4.7.5 Opción de herramientas del repositorio en GitHub

Posteriormente se habilita la opción de GitHub pages seleccionando el archivo principal que se generó en la rama maestra, esta opción se puede observar a continuación en la imagen 4.7.6

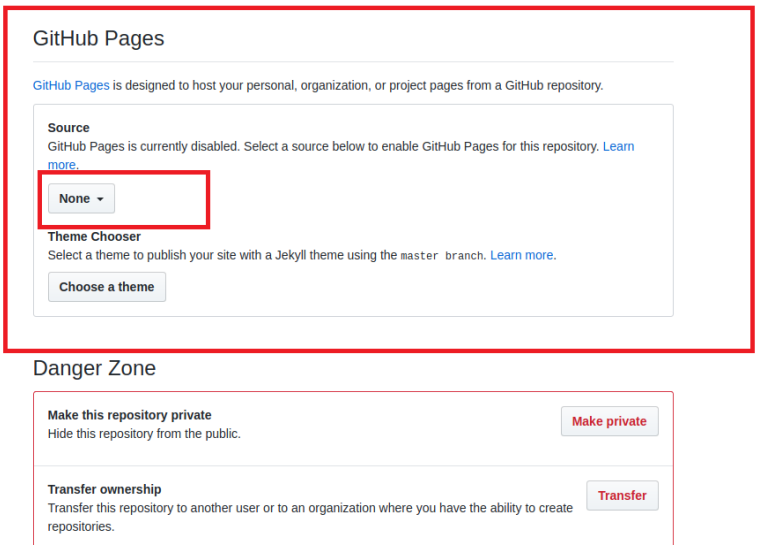

Imagen 4.7.6 Herramientas de GitHub Pages

Finalmente al seleccionar la opción de rama maestra se habilitará un enlace para abrir la aplicación web, así como se muestra en la siguiente imagen 4.7.7

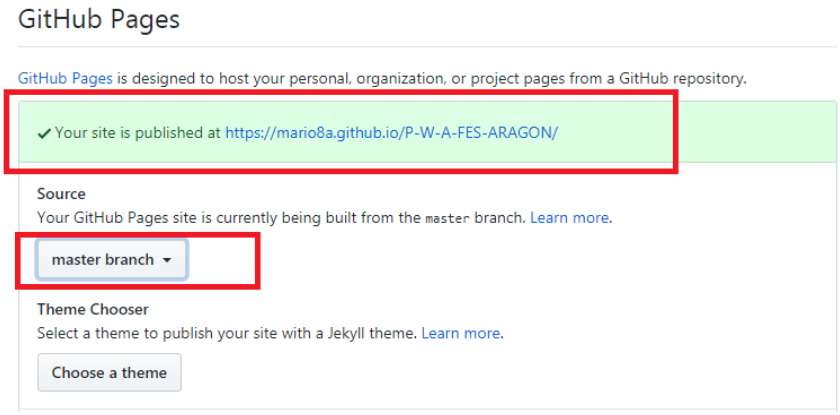

Imagen 4.7.7 Proyecto habilitado en GitHub Pages

4.8 ESTADO DE LA APLICACIÓN WEB PROGRESIVA

El estado de la aplicación web progresiva es importante tenerlo en cuenta, de lo contrario si no cumple con los estándares esta puede llegar a fallar en algunos dispositivos.

Para saber el estado en que se encuentra, google chrome cuenta con una herramienta de nombre **Audits,** está se encuentra en la barra de herramientas del navegador, en ella se deben habilitar las opciones que se muestran en la siguiente imagen 4.8.1

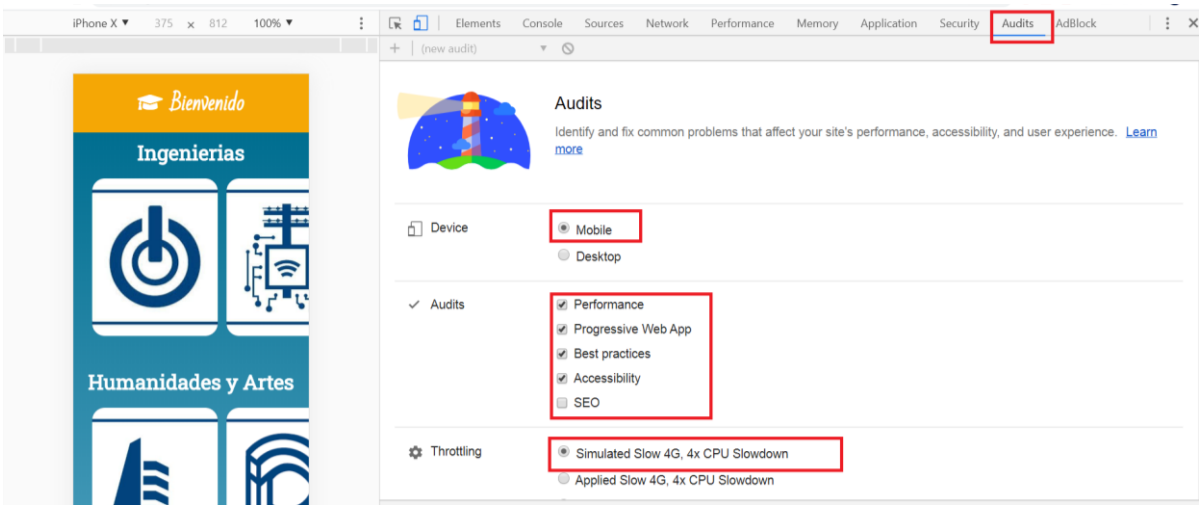

Imagen 4.8.1 Herramienta Audits

Al habilitar esta opción se mostrará la siguiente ventana que se muestra a continuación en la imagen 4.8.2

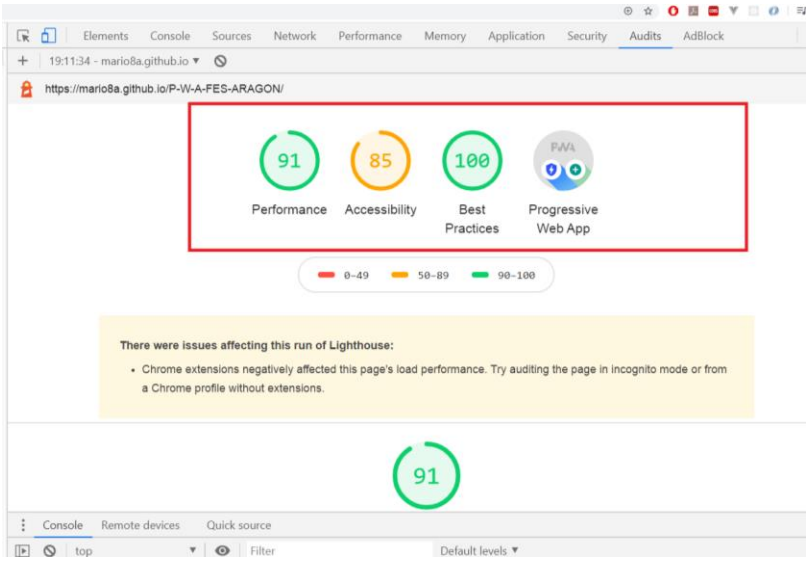

Imagen 4.8.2 Estado de la aplicación web

Cómo se puede observar en la imagen 4.8.2 hay 5 estados, sin embargo la más importante a seguir es la última de Progressive Web App, al dar click sobre esta se mostrará que puntos cumple como aplicación web progresiva. En la siguiente imagen 4.8.3 se muestra el resultado de la aplicación desarrollada en este capítulo.

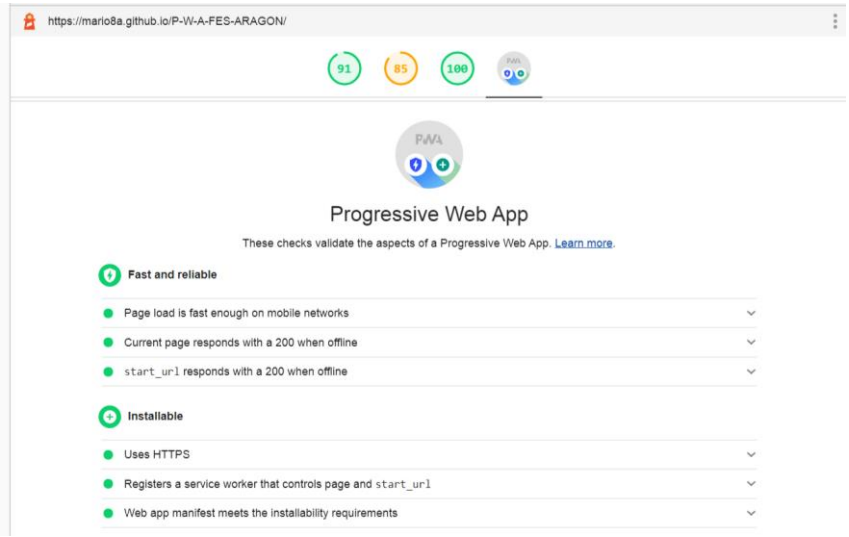

Imagen 4.8.3 Estado de la aplicación web progresiva

Como conclusión esta aplicación cumple con todos los estándares de una aplicación web progresiva, es rápida, funciona sin conexión a internet y está bajo un protocolo seguro HTTPS.

La idea detrás de este tipo de aplicaciones es que tengan similitud a las nativas, que sean útiles para el usuario, que ofrezcan una experiencia mejorada y que aprovechen las nuevas herramientas de la tecnología.

CONCLUSÓN

Hoy en día son más los usuarios que acceden a páginas web vía Smartphone o teléfonos que los que lo hacen a través de computadoras de escritorio. Actualmente más del 50% de los usuarios accede a internet con teléfonos, poco más del 4% por tabletas y el resto por computadoras de escritorio.

Cuando los Smartphone empezaron a popularizarse la estrategia que usaban los desarrolladores de los principales sitios web era adaptar las páginas de computadora a teléfono, el resultado eran páginas mal hechas y adaptadas

Las Aplicaciones Web Progresivas siguen siendo una tendencia en desarrollo desde el año 2015, sin embargo probablemente tome un par de años más descubrir si esta misma tendencia se mantiene. El concepto es muy interesante para muchos desarrolladores y a pesar de estar diseñadas con tecnologías de navegadores modernos, cada vez más personas van a poder tener acceso a este tipo de aplicaciones.

El desarrollo de este proyecto presentó ciertas dificultades, entre ellas analizar ciertas herramientas para el desarrollo de la misma aplicación y la falta de algunos conocimientos. Al trabajar en este proyecto desde su diseño hasta el funcionamiento básico, ha contribuido no solo en mis habilidades de programación, sino también a la capacidad de poder investigar y buscar información en diferentes fuentes.

Al final, con todo lo trabajado y presentado, puede concluir que los objetivos planteados al inicio de este trabajo han sido cubiertos satisfactoriamente y tanto la metodología cómo las herramientas planteadas fueron acertadas para resolver la problemática planteada en la introducción de este trabajo, a su vez difundir u compartir los conocimientos necesarios, explicados con detalle sobre cómo llegar a poder desarrollar este tipo de aplicaciones.

No solo se brinda el conocimiento teórico y acontecido, sino que también el conocimiento práctico y visual del código, así como detallar su funcionamiento y cómo usarlo en el plano web.

REFERENCIAS BIBLIOGRAFICAS

Artica Navarro, Robertho. *Desarrollo de aplicaciones móviles*. Quito. 2014, 64 p. Tesis (Ingeniero en sistemas e informática). Universidad Nacional de la Amazonía Peruana

ROLDÁN MORALES, M., & NEIL THOMPSON, D. (2013). *Aplicaciones móviles nativas orientadas a servicios y recursos de bibliotecas universitarias*. Recuperado de https://www.uned.ac.cr/academica/edutec/memoria/ponencias/morales_donoval_114.pdf

Gonzalo de Lucas (2014). *EVOLUCIÓN DE LAS APLICACIONES PARA MÓVILES*. Recuperado de https://empresarias.camara.es/estaticos/upload/0/007/7438.pdf

Ponce de León, D. (2016). Tutoriales HTML. Recuperado 23 marzo, 2020, de https://www.htmlquick.com/es/tutorials.html

Cuello Javier & Vittone Jose (2013). *Diseñando apps para móviles* (2ª ed.). Recuperado de http://appdesignbook.com/es/contenidos/las-aplicaciones

Florido Benitez L. (2016, diciembre). LAS APLICACIONES MÓVILES CONTRIBUYEN A MEJORAR LOS NIVELES DE SATISFACCIÓN DEL PASAJERO [Publicación en un blog]. Recuperado 23 marzo, 2020, de https://www.researchgate.net/publication/312119711_LAS_APLICACIONES_MOVILES_CONTRIBUYEN_A_MEJORAR_LOS_NIVELES_DE_SATISFACCION_DEL_PASAJERO

[Fazt]. (2019, Julio 6). 5 Tecnologías de Desarrollo de Aplicaciones Moviles Multiplataforma [Archivo de video]. Recuperado de https://www.youtube.com/watch?v=qC--aHT2c7M&t=767s

[Platzi]. (2018, Noviembre 2). Cómo crear tu primera app | PlatziLive [Archivo de video]. Recuperado de https://www.youtube.com/watch?v=6MfO1uU8Avk&t=37s

Carles, M. (2004). *Desarrollo de aplicaciones web* (Ed. rev.). Recuperado de https://libros.metabiblioteca.org/bitstream/001/591/1/004%20Desarrollo%20de%20aplicaciones%20web.pdf

[Fazt]. (2018, Enero 26). PWA | Progressive Web Apps, Introducción a las Aplicaciones Web Progresivas [Archivo de video]. Recuperado de https://www.youtube.com/watch?v=QSb4VZnF9q4&t=156s

[Autentia]. (2017, Febrero 15). Progressive Web Apps - Enrique García en T3chFest 2017 [Archivo de video]. Recuperado de https://www.youtube.com/watch?v=VK2i-osOQPw

¿Que es HTML y para que sirve? (2015). Recuperado 22 marzo, 2020, de http://www.acercadehtml.com/index.html

Estructura del cuerpo en HTML5. Etiqueta <footer>. (2014). Recuperado 22 marzo, 2020, de http://www.edu4java.com/es/index.html

Formularios basicos en HTML. (2006). Recuperado 23 marzo, 2020, de https://www.htmlquick.com/es/tutorials/forms.html

Ferrer, F., Garcia, V., & Garcia, R. (2003). *Curso completo de HTML* (Ed. rev.). Recuperado de http://es.tldp.org/Manuales-LuCAS/doc-curso-html/doc-curso-html.pdf

API fetch, el nuevo estándar que permite hacer llamadas HTTP. (2016, 28 octubre). Recuperado 22 marzo, 2020, de https://www.todojs.com/api-fetch-el-nuevo-estandar-que-permite-hacer-llamadas-http/

Archibald, Jake. (2019). El ciclo de vida del service worker. Recuperado 22 marzo, 2020, de https://developers.google.com/web/fundamentals/primers/service-workers/lifecycle?hl=es

Designing Native-Like Progressive Web Apps For iOS. (2018, 5 julio). Recuperado 22 marzo, 2020, de https://medium.com/appscope/designing-native-like-progressive-web-apps-for-ios-1b3cdda1d0e8

Grados Caballero, J. G. (2015). Manejo de Arrays en JavaScript. Recuperado 22 marzo, 2020, de https://devcode.la/tutoriales/manejo-de-arrays-en-javascript/

LePage, Pete., & Beaufort, Francois. (2018, 5 noviembre). Add a web app manifest. Recuperado 22 marzo, 2020, de https://web.dev/add-manifest/
Osmani, A. (2019). El modelo de "shell de app". Recuperado 22 marzo, 2020, de https://developers.google.com/web/fundamentals/architecture/app-shell?hl=es

Sánchez, G. (2015, 9 junio). JAVASCRIPT EN EL HEAD O EN EL CIERRE DEL BODY? ESTÁS EQUIVOCADO. Recuperado 22 marzo, 2020, de http://blog.desafiolatam.com/javascript-en-el-head-o-en-el-cierre-del-body-estas-equivocado/

ANEXO

Estructura Carrusel de imágenes

```
<div class="carrusel">
        <h1>Ingenierías</h1>
        <section class="carousel">
            <div class="carousel__container">
                <div class="carousel-item">
                    <a href="#modal-ico" id="show-modal">
                        <img class="carousel-item__img" src="img/inge/ico.jpeg">
                    </a>
                </div>
                <div class="carousel-item">
                    <a href="#modal-iee" id="show-modal">
                        <img class="carousel-item__img" src="img/inge/iee.jpg">
                    </a>
                </div>
                <div class="carousel-item">
                    <a href="#modal-iciv" id="show-modal">
                        <img class="carousel-item__img"
src="img/inge/iciv.jpeg">
                    </a>
                </div>
                <div class="carousel-item">
                    <a href="#modal-ime" id="show-modal">
                        <img class="carousel-item__img" src="img/inge/ime.jpeg"
>
                    </a>
                </div>
            </div>
        </section>
</div>
```

Diseño del carrusel de imágenes

```
.carrusel {
    overflow: hidden;
    min-height: calc(50vh);
    background-image: linear-gradient(#006E90, #74b8d4);
}

.carrusel h1 {
    margin: 0;
    padding: 10px;
    color: #ffffff;
    text-align: center;
```

```
}

.carousel {
    width: 100%;
    overflow-y: scroll;
    position: relative;
    padding-left: 30px;
}

.carousel__container {
    margin: 10px 0px;
    white-space: nowrap;
    padding-bottom: 10px;
}

.carousel-item {
    width: 200px;
    height: 250px;
    cursor: pointer;
    overflow: hidden;
    position: relative;
    margin-right: 10px;
    border-radius: 20px;
    display: inline-block;
    transition: 450ms all;
    background-color: #0D3B66;
    transform-origin: center left;
}

.carousel-item__img {
    width: 200px;
    height: 250px;
    object-fit: cover;
}
```

Diseño del modal

```
h2 {
    color: #0D3B66;
    padding: 1rem;
}

#show-modal {
    text-align: center;
    color: white;
    font-weight: 500;
    font-size: 1.5rem;
```

```
    text-decoration: none;
}

#show-modal:hover {
    cursor: pointer;
    /* background: rgb(72, 126, 29); */
}

.modal {
    position: fixed;
    top: -100vh;
    left: 0;
    z-index: 9999999;
    background: rgba(0, 0, 0, 0.75);
    width: 100vw;
    height: 100vh;
    opacity: 0;
    transition: opacity 0.35s ease;
}

.modal .content-modal {
    width: 100%;
    max-width: 500px;
    position: fixed;
    left: 50%;
    top: -100vh;
    transition: top 0.35s ease;
    margin-left: -250px;
    background: white;
    box-shadow: 0 1px 2px rgba(0, 0, 0, 0.1);
    border-radius: 2px;
    z-index: 9999999;
}

.modal h2 {
    padding: 1.2rem;
    text-align: center;
    color: #000000;
    margin: 0;
}

.modal h3 {
    text-align: start;
    padding-left: 15px;
    padding-top: 15px;
}

.modal p {
```

```
    text-align: justify;
    padding-left: 8px;
    padding-right: 8px;
}

.modal article {
    height: 350px;
    background: #b8b8b8;
}

.close-modal {
    color: #e74c3c;
    position: absolute;
    top: 0.2em;
    right: 0.375em;
    margin: 0;
    padding: 5px;
    font-weight: bold;
    font-size: 1.5em;
    text-decoration: none;
}

.modal a:hover {
    color: #c0392b;
}

.modal:target {
    opacity: 1;
    top: 0;
}

.modal .btn-close-modal {
    position: absolute;
    left: 0;
    width: 100%;
    height: 100%;
    z-index: 9999999;
}

.modal:target .content-modal {
    top: 50px;
    transition: top 0.35s ease;
}
```

Configuración del service worker archivo sw.js

```
const STATIC_CACHE = 'static-v4';
```

```
const DYNAMIC_CACHE = 'dynamic-v2';
const INMUTABLE_CACHE = 'inmutable-v1';

const APP_SHELL = [
    // '/',
    'index.html',
    'css/style.css',
    'img/favicon.ico',
    'img/inge/ico.jpeg',
    'img/inge/iee.jpg',
    'img/inge/iciv.jpeg',
    'img/inge/ime.jpeg',
    'img/humanidades/arqui.jpg',
    'img/humanidades/di.png',
    'img/humanidades/pedagogia.jpg',
    'img/cienciasS/agro.jpg',
    'img/cienciasS/cp.png',
    'img/cienciasS/derecho.png',
    'img/cienciasS/eco.jpg',
    'img/cienciasS/relaciones.jpeg',
    'img/cienciasS/sociologia.jpeg',
    'js/app.js',
    'js/sw-utils.js'
];

const APP_SHELL_INMUTABLE = [
    'https://fonts.googleapis.com/css?family=Bad+Script&display=swap',
    'https://fonts.googleapis.com/css?family=Roboto+Slab&display=swap',
    'https://fonts.googleapis.com/css?family=Montserrat&display=swap',
    'https://use.fontawesome.com/releases/v5.3.1/css/all.css',
    'css/animated.css'
];

self.addEventListener('install', e => {


    const cacheStatic = caches.open(STATIC_CACHE).then(cache =>
        cache.addAll(APP_SHELL));

    const cacheInmutable = caches.open(INMUTABLE_CACHE).then(cache =>
        cache.addAll(APP_SHELL_INMUTABLE));

    e.waitUntil(Promise.all([cacheStatic, cacheInmutable]));

});


self.addEventListener('activate', e => {
```

```
    const respuesta = caches.keys().then(keys => {

        keys.forEach(key => {

            if (key !== STATIC_CACHE && key.includes('static')) {
                return caches.delete(key);
            }

            if (key !== DYNAMIC_CACHE && key.includes('dynamic')) {
                return caches.delete(key);
            }

        });

    });

    e.waitUntil(respuesta);

});

// Comenzando con un cache only, siempre se debe dar una respuesta del
e.respondWith

self.addEventListener('fetch', e => {

    // verificar en el cache si existe la request
    const respuesta = caches.match(e.request).then(res => {

        if (res) {
            return res
        } else {
            return fetch(e.request).then(newRes => {
                return actualizaCacheDinamico(DYNAMIC_CACHE, e.request,
newRes);
            });
        }
    });

    e.respondWith(respuesta)
});
```

Configuración del archivo manifest.json

```
{
    "short_name": "FESAPWA",
```

```
    "name": "PWA informativa de FES Aragón",
    "start_url": "index.html",
    "background_color": "#212d40",
    "display": "standalone",
    "orientation": "portrait",
    "theme_color": "#212d40",
    "icons": [{
          "src": "img/icons/icon-72x72.png",
          "type": "image/png",
          "sizes": "72x72"
        },
        {
          "src": "img/icons/icon-96x96.png",
          "type": "image/png",
          "sizes": "96x96"
        },
        {
          "src": "img/icons/icon-120x120.png",
          "type": "image/png",
          "sizes": "120x120"
        },
        {
          "src": "img/icons/icon-144x144.png",
          "type": "image/png",
          "sizes": "144x144"
        },
        {
          "src": "img/icons/icon-152x152.png",
          "type": "image/png",
          "sizes": "152x152"
        },
        {
          "src": "img/icons/icon-192x192.png",
          "type": "image/png",
          "sizes": "192x192"
        },
        {
          "src": "img/icons/icon-310x310.png",
          "type": "image/png",
          "sizes": "310x310"
        }
    ]
}
```