CORFO
BecasCapital Humano

dalid
aspasia
LA FORMACIÓN DE TU FUTURO

MÓDULO 6

DESARROLLO DE APLICACIONES WEB PROGRESIVAS (PWA)

Manual del Módulo 6: Desarrollo de aplicaciones Web Progresivas (PWA)

Introducción

En este módulo, aprenderás a implementar **Aplicaciones Web Progresivas (PWA)** utilizando **React**, una tecnología que combina lo mejor de las aplicaciones web tradicionales y las aplicaciones móviles. Las PWA son rápidas, seguras, y capaces de funcionar sin conexión, gracias a herramientas clave como **Service Workers** y almacenamiento local avanzado. Además, aprenderás a optimizar el rendimiento y la experiencia de usuario usando herramientas como **Lighthouse**. Al final del módulo, serás capaz de crear aplicaciones modernas, responsivas y escalables que ofrezcan una experiencia de usuario inmersiva y confiable.

1. Introducción a las Aplicaciones PWA

1.2 ¿Qué es una PWA (Progressive Web Application)?

Una **PWA (Progressive Web Application)** es una aplicación web que aprovecha las últimas tecnologías disponibles para ofrecer una experiencia similar a la de una aplicación nativa. Combina las mejores características de las aplicaciones web tradicionales y de las aplicaciones móviles, proporcionando capacidades avanzadas como la instalación en dispositivos, la ejecución offline, y el envío de notificaciones push. Esto permite que las aplicaciones funcionen incluso en condiciones de conectividad limitada o nula.

1.3 Características de una PWA

Las PWA destacan por las siguientes características fundamentales:

- **Progresiva**: Funciona para cualquier usuario, independientemente del navegador o dispositivo, gracias a los principios de mejora progresiva.
- **Responsiva**: Se adapta al tamaño de pantalla y a las capacidades de cualquier dispositivo, ofreciendo una experiencia de usuario coherente en móviles, tabletas y escritorios.
- **Adaptable**: Puede funcionar sin conexión a Internet o con conexiones inestables, gracias a los **Service Workers**, que permiten almacenar en caché los datos.
- **Segura**: Siempre se sirve a través de HTTPS para garantizar la privacidad y la integridad de los datos.
- **Instalable**: Puede ser añadida a la pantalla de inicio del dispositivo sin necesidad de pasar por una tienda de aplicaciones, permitiendo que el usuario acceda a ella como si fuera una aplicación nativa.
- **Reenganchable**: Soporta el envío de notificaciones push para mantener a los usuarios comprometidos, incluso cuando la aplicación no está abierta.

1.4 Beneficios de una PWA

Las PWA aportan numerosos beneficios tanto para los usuarios como para los desarrolladores:

- **Mejor rendimiento**: Al almacenar los recursos en caché, las aplicaciones cargan rápidamente incluso en redes lentas o sin conexión.
- **Menor consumo de datos**: Gracias al uso de estrategias de almacenamiento en caché y precaching, las PWA reducen el uso de datos móviles.
- **Compatibilidad multiplataforma**: Una sola base de código puede funcionar en diferentes dispositivos sin necesidad de desarrollar versiones específicas para cada uno.
- **Actualización automática**: Las PWA se actualizan automáticamente en segundo plano, gracias al uso de los **Service Workers**, lo que garantiza que los usuarios siempre tengan la última versión sin necesidad de intervenciones manuales.

1.5 Limitaciones de una PWA

Aunque las PWA son muy poderosas, tienen algunas limitaciones en comparación con las aplicaciones nativas:

- **Acceso limitado a hardware**: Aunque cada vez más APIs permiten acceder a características como la cámara o la geolocalización, algunas funcionalidades de hardware aún no están disponibles, como el acceso completo a Bluetooth, sensores avanzados o el almacenamiento nativo del dispositivo.
- **Interacción limitada con otras aplicaciones**: Las PWA no tienen la misma capacidad de integración con otras aplicaciones instaladas en el dispositivo, como podría tener una aplicación nativa.
- **Rendimiento en tareas intensivas**: Para aplicaciones que requieren mucho procesamiento o gráficos avanzados, una PWA puede no ser tan eficiente como una aplicación nativa desarrollada específicamente para el dispositivo.

1.6 Diferencias entre una PWA, una aplicación web tradicional y una aplicación nativa

- **Aplicación Web Tradicional**: Se accede a través de un navegador y está limitada por la conectividad de red. No se instala en el dispositivo y no tiene capacidades offline ni de notificaciones push.
- **Aplicación Nativa**: Se desarrolla específicamente para un sistema operativo (iOS, Android, etc.), lo que permite un acceso más profundo al hardware y una mejor integración con el sistema. Sin embargo, requiere múltiples versiones para cada plataforma.
- **PWA**: Se accede como una aplicación web, pero puede instalarse en el dispositivo como una aplicación nativa. Funciona sin conexión y ofrece funcionalidades avanzadas como notificaciones push, aunque con acceso limitado al hardware en comparación con las aplicaciones nativas.

1.7 Arquitectura y componentes de una PWA (Service Workers, Manifiesto, Shell de la aplicación)

Una PWA consta de varios componentes clave que la hacen posible:

1. **Service Workers**: Son scripts que se ejecutan en segundo plano, permitiendo la interceptación de solicitudes de red, el almacenamiento en caché de recursos y la gestión de la funcionalidad offline.

2. **Manifiesto**: Un archivo JSON que proporciona metadatos sobre la aplicación, como su nombre, iconos, colores y cómo debe comportarse cuando se añade a la pantalla de inicio.
3. **Shell de la aplicación**: Es la estructura mínima necesaria para mostrar la interfaz de usuario, que se almacena en caché y se carga rápidamente. El contenido dinámico puede actualizarse independientemente de la estructura del shell.

1.8 Frameworks que soportan el desarrollo de PWA's

Existen varios **frameworks** que facilitan el desarrollo de **PWA** al proporcionar herramientas y funcionalidades preconfiguradas para implementar Service Workers, gestionar el almacenamiento en caché y optimizar la carga de la aplicación:

- **React**: Utilizando bibliotecas que incluye el soporte para PWA con Service Workers preconfigurados.
- **Angular**: A través del módulo Angular Service Worker que proporciona características de PWA listas para usar.
- **Vue.js**: El plugin Vue CLI PWA permite generar una aplicación web progresiva con configuraciones mínimas.
- **Ionic**: Muy popular para el desarrollo de aplicaciones híbridas, incluye soporte para PWA en su conjunto de herramientas.

Cada uno de estos frameworks facilita la creación de aplicaciones responsivas, escalables y que aprovechan las capacidades avanzadas de las PWA.

2. El Manifiesto

2.1 ¿Qué es el Manifiesto?

El **Manifiesto de una PWA** es un archivo en formato **JSON** que contiene metadatos esenciales sobre la aplicación web, como el nombre, los iconos, el color de fondo, y la URL de inicio. Este archivo define cómo debe comportarse la PWA cuando se instala en el dispositivo del usuario, similar a una aplicación nativa. El **manifiesto** permite que la PWA sea identificada como instalable por el navegador y proporciona una experiencia más personalizada para el usuario.

2.2 ¿Para qué se usa el Manifiesto?

El archivo **manifest.json** se utiliza para:

1. **Definir cómo se verá y se comportará la PWA** cuando sea instalada en el dispositivo del usuario.
2. **Configurar la apariencia de la PWA** en la pantalla de inicio del dispositivo, incluyendo el nombre, iconos y colores temáticos.
3. **Determinar el modo de visualización** de la aplicación (por ejemplo, si se muestra a pantalla completa o en un marco de navegador).
4. **Especificar la URL de inicio** que la aplicación cargará cuando sea abierta desde el icono en la pantalla de inicio.
5. **Aumentar la coherencia visual y de experiencia** al hacer que la PWA se asemeje a una aplicación nativa en términos de comportamiento y aspecto.

2.3 Estructura de un archivo de Manifiesto

Un archivo de manifiesto estándar contiene varias propiedades clave que personalizan la experiencia del usuario. Aquí te muestro un ejemplo típico de un archivo `manifest.json`:

```
{

  "name": "Mi Aplicación PWA",
  "short_name": "PWA",
  "description": "Una aplicación web progresiva de ejemplo.",
  "start_url": "/index.html",
  "display": "standalone",
  "background_color": "#ffffff",
  "theme_color": "#000000",
  "icons": [
    {
      "src": "icon-192x192.png",
      "sizes": "192x192",
```

```
      "type": "image/png"
    },
    {
      "src": "icon-512x512.png",
      "sizes": "512x512",
      "type": "image/png"
    }
  ]
}
```

- **name**: El nombre completo de la aplicación, que se mostrará al usuario, por ejemplo, en la pantalla de inicio.
- **short_name**: Una versión más corta del nombre que se usará cuando no haya espacio suficiente para mostrar el nombre completo.
- **description**: Una breve descripción de la aplicación que los navegadores pueden utilizar para dar más contexto al usuario.
- **start_url**: Define la URL que se abrirá cuando el usuario inicie la aplicación desde la pantalla de inicio. Esta URL generalmente apunta al punto de entrada de la aplicación.
- **display**: Define cómo se debe mostrar la PWA. Los valores posibles incluyen:
    - **standalone**: La aplicación se comporta como una aplicación nativa, sin barra de navegador.
    - **fullscreen**: La aplicación ocupa toda la pantalla.
    - **minimal-ui**: Similar a standalone pero con una pequeña barra de navegación.
- **background_color**: El color de fondo que se mostrará durante la carga inicial de la aplicación, mientras se está cargando la página principal.
- **theme_color**: El color del tema de la aplicación, que se refleja en la interfaz del dispositivo, como la barra de estado en móviles.
- **icons**: Una lista de iconos en diferentes tamaños que el dispositivo utilizará para mostrar la aplicación en la pantalla de inicio.

Este archivo es fundamental para garantizar que la PWA se vea y se sienta como una aplicación nativa, proporcionando una experiencia de usuario consistente y mejorada.

3. El Service Worker

3.1 ¿Qué es el Service Worker?

El **Service Worker** es un script que el navegador ejecuta en segundo plano, separado del hilo principal de la aplicación. Permite que una aplicación web tenga características avanzadas como el **trabajo sin conexión**, el **almacenamiento en caché** y las **notificaciones push**. Actúa como un intermediario entre la aplicación y la red, permitiendo interceptar y gestionar las solicitudes de red y almacenar los recursos para su uso futuro.

3.2 ¿Para qué se usa el Service Worker?

El **Service Worker** se utiliza principalmente para:

- **Habilitar el funcionamiento sin conexión (offline)** mediante el almacenamiento en caché de recursos.
- **Mejorar el rendimiento** cargando recursos directamente desde el caché en lugar de hacer solicitudes constantes a la red.
- **Gestionar notificaciones push** y la sincronización en segundo plano.
- **Aumentar la confiabilidad** de las aplicaciones web al garantizar que funcionen incluso cuando la conexión a internet sea inestable o no esté disponible.

3.3 Ventajas de usar un Service Worker

1. **Funciona offline**: Una de las mayores ventajas de un Service Worker es la capacidad de hacer que las aplicaciones funcionen sin conexión, lo que mejora significativamente la experiencia del usuario.
2. **Mejora del rendimiento**: El Service Worker puede almacenar en caché archivos clave, permitiendo que la aplicación cargue mucho más rápido en visitas subsecuentes.
3. **Interacción en segundo plano**: Al funcionar en segundo plano, los Service Workers permiten ejecutar tareas como la sincronización de datos y el envío de notificaciones sin que el usuario tenga la aplicación abierta.
4. **Aislamiento del hilo principal**: Como el Service Worker se ejecuta fuera del hilo principal, no afecta el rendimiento de la interfaz de usuario, lo que mejora la capacidad de respuesta.

3.4 Descripción general de un Service Worker

Un **Service Worker** tiene varias características clave:

- **API asíncrona**: El Service Worker utiliza promesas y eventos asíncronos para realizar sus tareas sin bloquear el hilo principal de la aplicación.

- **API basada en eventos**: El Service Worker reacciona a eventos como `fetch`, `install` y `activate`. Por ejemplo, intercepta peticiones de red (`fetch`) y puede responder con recursos almacenados en caché.

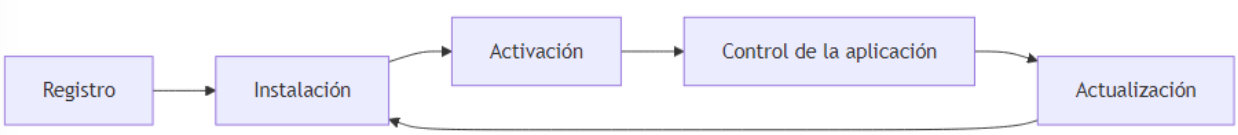
- **Precaching**: Durante la instalación del Service Worker, puedes pre-cargar ciertos recursos para que estén disponibles offline, mejorando el tiempo de carga de la aplicación en futuras visitas.

- **Aislamiento del hilo principal**: El Service Worker se ejecuta en un hilo independiente, lo que significa que no puede manipular el DOM directamente, lo que mejora el rendimiento general.

3.5 Ciclo de vida de un Service Worker

El ciclo de vida de un Service Worker incluye las siguientes etapas:

1. **Registro**: Se registra el Service Worker en el navegador cuando la aplicación se carga por primera vez.
2. **Instalación**: El Service Worker se descarga e instala, cargando en caché los archivos especificados.
3. **Activación**: El Service Worker toma control de la aplicación y empieza a gestionar las solicitudes de red.
4. **Actualización**: Si hay cambios en el código del Service Worker, se activa una nueva versión y el ciclo de vida comienza de nuevo.



3.6 Cómo interactúa el Service Worker con el caché y el acceso a la red

El Service Worker actúa como un intermediario entre la aplicación y la red, interceptando las solicitudes de red. Puede optar por:

- **Buscar en caché** si los recursos solicitados ya están almacenados.
- **Realizar la solicitud a la red** y almacenar la respuesta en caché para futuras solicitudes.

Esto permite que la aplicación funcione sin conexión y mejore el rendimiento al cargar los recursos almacenados en caché en lugar de hacer solicitudes innecesarias a la red.

3.7 Cómo configurar Service Worker para ReactJs

En aplicaciones React creadas con `create-react-app`, el Service Worker ya está integrado. Para activarlo, solo necesitas registrar el Service Worker en el archivo `index.js`:

```
import * as serviceWorkerRegistration from './serviceWorkerRegistration';

// Activar Service Worker
serviceWorkerRegistration.register();
```

Este registro habilitará el uso del Service Worker y proporcionará funcionalidades offline para la aplicación React.

3.8 Funcionamiento de una PWA con HTTPS

Los **Service Workers** requieren que la aplicación esté servida a través de **HTTPS** por razones de seguridad. Esto garantiza que las comunicaciones sean seguras y protege contra ataques como el "man-in-the-middle". En desarrollo local, puedes usar `localhost`, pero en producción, la PWA debe estar servida en un servidor HTTPS.

3.9 Estrategias de almacenamiento en caché de Service Worker

Existen varias estrategias para gestionar el almacenamiento en caché de recursos con un Service Worker, cada una adecuada para diferentes casos de uso:

3.9.1 Stale-While-Revalidate

El Service Worker carga primero los recursos del caché, mientras actualiza la versión en segundo plano desde la red.

```
self.addEventListener('fetch', (event) => {
  event.respondWith(
    caches.open('app-cache').then((cache) => {
      return cache.match(event.request).then((response) => {
        const fetchPromise = fetch(event.request).then((networkResponse) =>
{
          cache.put(event.request, networkResponse.clone());
          return networkResponse;
        });
        return response || fetchPromise;
      });
    })
  );
});
```

3.9.2 Cache-First

Se priorizan los recursos del caché, solo se realiza una solicitud a la red si no se encuentra el recurso en caché.

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

```
    caches.match(event.request).then((response) => {
      return response || fetch(event.request);
    })
  );
});
```

3.9.3 Network-First

Se intenta primero obtener los recursos de la red, y si la solicitud falla (por ejemplo, si no hay conexión), se busca en el caché.

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

3.9.4 Cache-Only

Solo se recuperan los recursos desde el caché, sin hacer solicitudes a la red.

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

3.9.5 Network-Only

Solo se realizan solicitudes a la red, sin utilizar el caché.

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

3.9.6 Cache-And-Network

Similar a Network-First, pero actualiza el caché con la respuesta de la red, para estar siempre sincronizado.

4. Almacenamiento en una PWA

4.1 Cómo se administra el Almacenamiento Web en una PWA

El almacenamiento en una **PWA** es crucial para permitir que la aplicación funcione sin conexión, manteniendo datos persistentes o temporales entre sesiones. Una PWA puede utilizar diferentes técnicas y tecnologías para almacenar información en el navegador del usuario:

- **Cache API**: Es utilizada principalmente por el **Service Worker** para almacenar archivos estáticos como HTML, CSS, JavaScript y recursos multimedia. Esto permite que la aplicación funcione sin conexión o con una carga más rápida en visitas posteriores.

- **LocalStorage y SessionStorage**: Se utilizan para almacenar pequeños volúmenes de datos, como preferencias de usuario o sesiones activas. A diferencia de Cache API, estos almacenamientos son más adecuados para datos que cambian de manera dinámica.

- **IndexedDB**: Es una base de datos en el navegador, ideal para almacenar grandes volúmenes de datos estructurados como listas, tablas o archivos complejos. A diferencia de LocalStorage, es más robusta y permite consultas complejas.

- **File System Access API**: Permite acceder a archivos locales desde el navegador (aunque con soporte limitado) y es útil para PWA que necesiten trabajar con archivos grandes, como aplicaciones de edición de fotos o vídeos.

4.2 El uso de LocalStorage y SessionStorage

LocalStorage y **SessionStorage** son dos APIs del navegador que permiten almacenar datos clave-valor de forma simple, pero con algunas diferencias importantes:

- **LocalStorage**: Los datos almacenados en LocalStorage son persistentes y permanecen disponibles incluso después de que el usuario cierra la pestaña o el navegador.

  Ejemplo de LocalStorage:

```
// Guardar un valor en LocalStorage
localStorage.setItem('nombreUsuario', 'Juan');

// Recuperar un valor de LocalStorage
let nombre = localStorage.getItem('nombreUsuario');

// Eliminar un valor de LocalStorage
localStorage.removeItem('nombreUsuario');
```

- **SessionStorage**: Similar a LocalStorage, pero los datos solo persisten durante la sesión actual del navegador. Una vez que se cierra la pestaña, los datos almacenados en SessionStorage se eliminan.

  Ejemplo de SessionStorage:

```
// Guardar un valor en SessionStorage
sessionStorage.setItem('autenticado', 'true');

// Recuperar un valor de SessionStorage
let autenticado = sessionStorage.getItem('autenticado');

// Eliminar un valor de SessionStorage
sessionStorage.removeItem('autenticado');
```

Ambos métodos son útiles para almacenar pequeñas cantidades de datos, pero no son adecuados para grandes volúmenes de información o para datos que requieran operaciones complejas como búsquedas o filtrados.

4.3 Acerca de WebAssembly y el código precompilado

WebAssembly (Wasm) es una tecnología que permite ejecutar código de bajo nivel en el navegador, proporcionando un rendimiento casi nativo. Aunque **JavaScript** es el lenguaje principal en las PWA, **WebAssembly** permite ejecutar código precompilado escrito en lenguajes como C, C++ o Rust, que puede ser utilizado en tareas que requieren un alto rendimiento.

Ventajas de usar WebAssembly en PWA:

- **Alto rendimiento**: Permite ejecutar cálculos complejos de forma eficiente, como en juegos o aplicaciones científicas.
- **Interoperabilidad**: Se puede combinar con JavaScript para potenciar las aplicaciones web.
- **Portabilidad**: WebAssembly puede ejecutarse en cualquier navegador moderno sin depender de la plataforma.

Ejemplo básico de uso de WebAssembly:

```
// Cargar y ejecutar un módulo WebAssembly
fetch('modulo.wasm').then(response =>
  response.arrayBuffer()
).then(bytes =>
  WebAssembly.instantiate(bytes)
).then(results => {
  console.log('Módulo WebAssembly cargado:', results);
});
```

4.4 Bibliotecas de bases de datos con soporte para PWA (IndexedDB, PouchDB, RxDB, GunDB)

Para almacenar grandes volúmenes de datos en una PWA, existen varias bases de datos basadas en el navegador, que son especialmente útiles para aplicaciones que necesitan trabajar sin conexión y sincronizar datos cuando vuelve la conectividad:

1. **IndexedDB**:

    - Es una base de datos en el navegador que permite almacenar datos estructurados, realizar búsquedas y almacenar grandes cantidades de datos.
    - A diferencia de LocalStorage, IndexedDB es asíncrona y permite almacenar datos más complejos.

    Ejemplo de uso de IndexedDB:

```
let db;
let request = indexedDB.open('miBaseDeDatos', 1);

request.onupgradeneeded = function(event) {
  db = event.target.result;
  db.createObjectStore('usuarios', { keyPath: 'id' });
};

request.onsuccess = function(event) {
  db = event.target.result;
  let transaction = db.transaction(['usuarios'], 'readwrite');
  let store = transaction.objectStore('usuarios');

  store.put({ id: 1, nombre: 'Juan' });

};
```

2. **PouchDB**:

    - Es una base de datos de código abierto que puede sincronizar datos con bases de datos CouchDB en la nube, lo que permite a las PWA almacenar datos localmente y sincronizarlos cuando vuelva la conectividad.
    - Perfecta para aplicaciones que requieren sincronización en tiempo real.

    Ejemplo de uso de PouchDB:

```
const db = new PouchDB('miBaseDeDatos');
```

```
db.put({
  _id: 'usuario_1',
  nombre: 'Juan',
  edad: 30
}).then(function(response) {
  console.log('Usuario guardado:', response);
});
```

3. **RxDB**:

   - RxDB es una base de datos basada en **Reactive Programming**. Permite sincronización de datos en tiempo real, almacenando localmente y replicando los datos cuando hay conectividad.
   - Es ideal para aplicaciones que requieren una gran interacción en tiempo real entre clientes y servidores.

4. **GunDB**:

   - GunDB es una base de datos descentralizada, lo que significa que se puede compartir entre varios dispositivos y usuarios sin necesidad de un servidor centralizado.
   - Es particularmente útil para aplicaciones colaborativas o descentralizadas donde múltiples usuarios necesitan interactuar con los mismos datos.

   Ejemplo básico de GunDB:

```
const gun = Gun();

// Guardar datos
gun.get('usuario').put({ nombre: 'Juan', edad: 30 });

// Recuperar datos
gun.get('usuario').on(function(data) {
  console.log('Datos del usuario:', data);
});
```

Cada una de estas bases de datos tiene su propio conjunto de características, y la elección dependerá de los requisitos de tu aplicación PWA. **IndexedDB** es ideal para el almacenamiento de grandes cantidades de datos offline, mientras que bases de datos como **PouchDB** y **RxDB** son perfectas para aplicaciones que requieren sincronización de datos en tiempo real.

5. Creando una PWA

5.1 Creando un proyecto PWA en ReactJs

Para crear una **PWA en React** de manera sencilla, puedes usar la herramienta `create-react-app`, que ya incluye soporte para **PWA** con un **Service Worker** preconfigurado. Aquí están los pasos para crear un proyecto React con capacidades PWA:

1. Abre una terminal y ejecuta el siguiente comando:

```
npx create-react-app mi-pwa --template cra-template-pwa
```

   Este comando crea un proyecto React con la configuración necesaria para una **Progressive Web App (PWA)**.

2. El archivo `manifest.json` y el `Service Worker` estarán listos para usar. El archivo `manifest.json` contiene la configuración de la aplicación, como los iconos, colores, y otros metadatos importantes para la PWA.

3. Para comenzar el servidor de desarrollo, ejecuta:

```
npm start
```

Este comando inicia tu aplicación y la sirve localmente en `http://localhost:3000`, lista para ser desarrollada con capacidades de PWA.

5.2 Registrando un Service Worker

En una PWA, el **Service Worker** es el componente central que habilita el trabajo sin conexión y la gestión de caché. Por defecto, `create-react-app` incluye un archivo `serviceWorkerRegistration.js` para manejar el registro del **Service Worker**.

1. En el archivo `src/index.js`, habilita el registro del Service Worker cambiando la línea correspondiente:

```
import * as serviceWorkerRegistration from './serviceWorkerRegistration';

serviceWorkerRegistration.register();  // Activar el Service Worker
```

Al hacer esto, el Service Worker se registrará automáticamente cuando el usuario cargue la aplicación.

5.3 Personalizando un Service Worker

Puedes personalizar el comportamiento de tu **Service Worker** para ajustar las estrategias de caché o agregar nuevas funcionalidades como notificaciones push o sincronización en segundo plano. Aquí te mostramos cómo hacer modificaciones básicas en el Service Worker:

1. En el archivo `src/service-worker.js` (o `serviceWorkerRegistration.js`), puedes personalizar el manejo de eventos de red. Por ejemplo, para agregar una estrategia de **Cache-First**:

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

Este código asegura que el Service Worker primero busque en el caché antes de hacer una solicitud de red.

5.4 Navegando a través de una PWA

Una vez que la PWA está instalada en el dispositivo, se puede navegar por la aplicación de manera fluida, como si fuera una aplicación nativa. Los siguientes aspectos son importantes al navegar a través de una PWA:

1. **Experiencia sin conexión**: Gracias a los **Service Workers**, una PWA puede funcionar sin conexión, sirviendo recursos almacenados en caché cuando no hay acceso a la red.
2. **Navegación fluida**: Al estar instalada, la PWA se ejecuta a pantalla completa sin barra de navegador, mejorando la experiencia del usuario.

5.5 Implementando las distintas estrategias de almacenamiento en caché de Service Worker

Existen varias estrategias para el almacenamiento en caché que puedes implementar en tu **Service Worker** para optimizar el rendimiento de la PWA:

1. **Cache-First**: Busca primero en caché, y si no encuentra el recurso, hace la solicitud de red.

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

2. **Network-First**: Intenta obtener el recurso desde la red primero, y si falla, lo busca en el caché.

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

3. **Stale-While-Revalidate**: Devuelve el recurso del caché mientras actualiza en segundo plano desde la red.

```
self.addEventListener('fetch', (event) => {
  event.respondWith(
    caches.open('my-cache').then((cache) => {
      return cache.match(event.request).then((response) => {
        let fetchPromise = fetch(event.request).then((networkResponse) => {
          cache.put(event.request, networkResponse.clone());
          return networkResponse;
        });
        return response || fetchPromise;
      });
    })
  );
});
```

Estas estrategias te permiten balancear el uso de caché y red según los requisitos de la aplicación, mejorando el rendimiento y la confiabilidad.

5.6 Accediendo a periféricos del Sistema Operativo

Las PWA pueden acceder a ciertos periféricos del dispositivo mediante APIs específicas del navegador. Algunas de las APIs más comunes incluyen:

- **Geolocalización**: Puedes acceder a la ubicación del dispositivo usando la API de geolocalización del navegador.

```
navigator.geolocation.getCurrentPosition((position) => {
  console.log(position.coords.latitude, position.coords.longitude);
});
```

- **Cámara**: Puedes acceder a la cámara del dispositivo usando la API de `MediaDevices`.

```
navigator.mediaDevices.getUserMedia({ video: true })
  .then((stream) => {
    let video = document.querySelector('video');
    video.srcObject = stream;
  })
  .catch((err) => console.error('Error al acceder a la cámara:', err));
```

- **Notificaciones Push**: Mediante **Service Workers**, puedes enviar notificaciones push a los usuarios, incluso cuando la PWA no está activa.

5.7 Despliegue de una PWA

El despliegue de una **PWA** implica asegurarse de que la aplicación esté servida a través de **HTTPS**, ya que los **Service Workers** solo funcionan bajo conexiones seguras. Aquí están los pasos básicos para desplegar tu PWA en producción:

1. **Construir la aplicación para producción**:

```
npm run build
```

   Esto generará una versión optimizada de la aplicación en la carpeta `build`.

2. **Subir la aplicación a un servidor HTTPS**: Puedes desplegar la aplicación en cualquier servidor que soporte HTTPS, como **Netlify**, **Vercel**, **Firebase Hosting**, o tu propio servidor.

3. **Verificar el comportamiento de la PWA**: Después de desplegar la aplicación, verifica que el **Service Worker** esté funcionando y que la PWA se pueda instalar correctamente.

   Puedes usar herramientas como **Lighthouse** (integrada en Chrome DevTools) para auditar y mejorar la calidad de la PWA, asegurándose de que cumple con los requisitos de rendimiento, accesibilidad y mejores prácticas.

Con estos pasos, habrás desplegado una **PWA** completamente funcional, capaz de trabajar sin conexión, mejorar el rendimiento, y ofrecer una experiencia similar a las aplicaciones nativas.

6. Explorando el Estado de una PWA con Lighthouse

Lighthouse es una herramienta de auditoría automatizada que evalúa el rendimiento, accesibilidad, mejores prácticas, SEO y capacidad de una aplicación web para ser una **PWA**. Utilizando **Lighthouse**, puedes obtener un informe detallado sobre el estado de tu **Progressive Web App**, permitiendo identificar áreas de mejora.

6.1 Instalando la extensión Lighthouse en el navegador

Lighthouse está integrado directamente en **Chrome DevTools**, por lo que no es necesario instalar una extensión aparte. Sin embargo, también puedes instalar la extensión de **Lighthouse** desde la **Chrome Web Store** si prefieres usarla de forma independiente.

1. Para usar Lighthouse desde Chrome DevTools:

   - Abre **Chrome** y navega a la página de tu PWA.
   - Haz clic derecho en la página y selecciona **Inspeccionar** para abrir las **DevTools**.
   - Ve a la pestaña **Lighthouse**.

2. Para instalar la extensión:

- Abre la **Chrome Web Store**.
- Busca **Lighthouse** y selecciona la opción de extensión de **Google**.
- Haz clic en **Añadir a Chrome**.

Una vez instalada, verás un icono de Lighthouse en la barra de extensiones del navegador.

6.2 Algunos aspectos relevantes de Lighthouse (Rápido, instalable, optimizado para PWA)

Lighthouse analiza múltiples aspectos de la PWA para determinar si cumple con los requisitos necesarios para proporcionar una **experiencia de usuario de alta calidad**. Los puntos más importantes que evalúa Lighthouse para las PWA son:

- **Rápido**: Mide el tiempo de carga de la página. Una PWA debe cargar rápido, incluso en redes móviles lentas. Lighthouse evalúa factores como el **First Contentful Paint** (FCP), el **Largest Contentful Paint** (LCP), y la **interactividad**.

- **Instalable**: Verifica si la aplicación es instalable. La PWA debe tener un manifiesto y un **Service Worker** correctamente configurado, y debe estar servida a través de HTTPS para garantizar la seguridad.

- **Optimizada para PWA**: Lighthouse revisa si la PWA sigue las mejores prácticas para Progressive Web Apps, como el uso eficiente de caché, la capacidad de funcionar sin conexión, y la optimización de los recursos cargados.

6.3 Ejecutando Lighthouse (Análisis de carga de página)

Para ejecutar una auditoría con Lighthouse en **Chrome DevTools**, sigue estos pasos:

1. **Abre la pestaña de Lighthouse** en **DevTools**:

   - Haz clic en **F12** o haz clic derecho en la página y selecciona **Inspeccionar**.
   - Navega a la pestaña **Lighthouse**.

2. **Selecciona las categorías que deseas auditar**:

- Lighthouse permite elegir diferentes categorías para auditar, incluyendo **Performance**, **Progressive Web App**, **Best Practices**, **SEO**, y **Accessibility**. Selecciona **Progressive Web App** para verificar específicamente las características de PWA.

3. **Genera el informe**:

    - Haz clic en el botón **Generate report** para comenzar la auditoría. Lighthouse ejecutará una serie de pruebas en la página y generará un informe detallado.

4. **Revisión del informe**:

    - El informe de Lighthouse mostrará una puntuación general para cada categoría, junto con recomendaciones específicas para mejorar el rendimiento o cumplir con los estándares de PWA.

6.4 Revisando el informe de Lighthouse

El informe generado por **Lighthouse** incluye varias secciones clave que ayudan a comprender el estado actual de la PWA:

1. **Puntuación general**: Cada categoría (Performance, SEO, PWA, etc.) tiene una puntuación del 1 al 100, que refleja la calidad en esa área. Una buena PWA debe alcanzar al menos una puntuación de **90** en la categoría **PWA**.

2. **Detalles de la auditoría PWA**: Aquí, Lighthouse analiza aspectos específicos de una PWA, incluyendo:

    - **Responsive**: ¿Es la aplicación adaptable a diferentes tamaños de pantalla?
    - **Fast loading**: ¿Carga la PWA rápidamente, incluso en redes lentas?
    - **Works offline**: ¿Funciona la PWA sin conexión a Internet?
    - **Installable**: ¿Tiene un manifiesto válido y un Service Worker funcionando para permitir la instalación?

3. **Sugerencias de mejora**: Lighthouse proporciona recomendaciones prácticas para mejorar el rendimiento, como:

- **Reducir el tamaño de las imágenes**.
- **Minificar archivos CSS y JavaScript**.
- **Optimizar el uso de caché y precaching en el Service Worker**.

4. **Problemas críticos**: Si Lighthouse detecta que tu PWA no es instalable o no funciona sin conexión, se mostrará un mensaje con los pasos necesarios para corregir esos problemas.

5. **Análisis de recursos cargados**: Lighthouse también muestra una lista de todos los recursos cargados durante la auditoría (archivos CSS, imágenes, scripts, etc.) y señala cuáles son los más pesados o tardan más en cargarse. Esto te ayudará a identificar los elementos que pueden estar ralentizando tu PWA.

Con este análisis, puedes mejorar iterativamente el rendimiento y las funcionalidades de tu PWA para ofrecer una experiencia de usuario óptima.

Material de Referencia

Libros:

- **"Learning Progressive Web Apps"** de John M. Wargo
  Este libro proporciona una introducción completa a las PWA, cubriendo todos los conceptos clave como **Service Workers**, **manifiestos**, **precaching** y las mejores prácticas para su desarrollo. Perfecto para desarrolladores que buscan construir aplicaciones rápidas y confiables.

- **"Progressive Web Apps"** de Jason Grigsby
  Un excelente recurso para aprender a optimizar las aplicaciones web modernas para trabajar offline, mejorar la velocidad y ofrecer una experiencia de usuario consistente en diferentes dispositivos. El libro se enfoca en las tecnologías que hacen posibles las PWA, como el **Service Worker** y el **caché**.

- **"High Performance Browser Networking"** de Ilya Grigorik

Aunque no está exclusivamente enfocado en PWA, este libro ofrece una sólida base en el rendimiento de la red y cómo optimizar la carga de recursos en aplicaciones web, algo esencial para cualquier PWA.

Enlaces a Recursos Online:

- **Progressive Web Apps - MDN Web Docs**: Una excelente guía proporcionada por MDN que explica desde los conceptos básicos de las **PWA** hasta la implementación de características avanzadas como el **almacenamiento en caché** y el uso de **Service Workers**.

- **Google Developers - PWA Documentation**: La documentación oficial de Google sobre **Progressive Web Apps**, que incluye guías prácticas, tutoriales y ejemplos de cómo crear y optimizar PWA utilizando **Lighthouse**, **Service Workers** y **Web App Manifests**.

- **Lighthouse - Google Developers**: Documentación detallada sobre cómo utilizar **Lighthouse** para auditar aplicaciones web, mejorar el rendimiento, y garantizar que cumplan con los estándares de PWA.

- **Workbox - Google Developers**: Una biblioteca desarrollada por Google que facilita la creación de **Service Workers** y la implementación de estrategias de caché avanzadas, optimizando el comportamiento offline de las aplicaciones.

Videos Recomendados:

- **What Are Progressive Web Apps? - Fireship**: Un video introductorio rápido sobre qué son las PWA, cómo funcionan y por qué son importantes en el ecosistema de desarrollo web moderno. Ideal para obtener una visión general clara y concisa.

- **Service Workers in Depth - Academind**: Un video que explica en profundidad cómo funcionan los **Service Workers**, cubriendo aspectos como el ciclo de vida, la gestión del caché y cómo manejar eventos de red en una PWA.

Otros Recursos Útiles:

- **Awesome PWA - GitHub**: Una lista de recursos sobre PWA que incluye herramientas, tutoriales, bibliotecas, y ejemplos de aplicaciones web progresivas.

- **Workbox GitHub Repository**: Repositorio oficial de **Workbox**, una biblioteca que facilita la creación de Service Workers y la implementación de estrategias avanzadas de caché para **PWAs**.

CORFO
BecasCapital Humano

dalid
aspasia
LA FORMACIÓN DE TU FUTURO

MÓDULO 6

DESARROLLO DE APLICACIONES WEB PROGRESIVAS (PWA)