## 🎯 Objetivo General:

Capacitar a los participantes en el uso avanzado de **Service Workers** y otras tecnologías web modernas para construir aplicaciones web progresivas (PWA) que ofrezcan una experiencia similar a la de aplicaciones nativas, incluso sin conexión.

## MÓDULO 1: Introducción a las Progressive Web Apps (PWA)

### Temas:

- ¿Qué es una PWA?
- Ventajas frente a aplicaciones nativas y web tradicional
- Requisitos técnicos para una PWA:
  - HTTPS
  - Manifest Web App
  - Service Worker
- Herramientas básicas y entorno de desarrollo

## MÓDULO 2: Fundamentos de Service Workers

### Temas:

- ¿Qué es un Service Worker?
- Ciclo de vida del Service Worker:
  - Registro
  - Instalación
  - Activación
- Comunicación entre Service Worker y cliente (postMessage)
- Buenas prácticas y consideraciones importantes:
  - Solo funciona bajo HTTPS (excepto localhost)
  - No tiene acceso al DOM

### Práctica:

- Registrar un Service Worker desde el cliente
- Mostrar mensajes en consola durante cada etapa del ciclo de vida

## MÓDULO 3: Uso del Cache Storage

### Temas:

- ¿Por qué usar `CacheStorage`?
- Métodos principales de la API Cache:
  - `caches.open()`
  - `cache.put()`, `cache.match()`, etc.
- Estrategias de caché manuales:
  - Cache-only
  - Network-only
  - Cache-first
  - Network-first
  - Stale-while-revalidate (manual)

### Práctica:

- Implementar una estrategia de caché para archivos estáticos (CSS, JS, imágenes)
- Interceptando solicitudes con `fetch` event
- Cachear contenido dinámico (ej.: API REST básica)

## MÓDULO 4: Sincronización en Segundo Plano (Background Sync)

### Temas:

- ¿Qué es Background Sync?
- Uso de la API `SyncManager`
- Tipos de sincronización:
  - One-off sync
  - Periodic sync (limitado por permisos del navegador)

### Práctica:

- Registrar una tarea de sincronización cuando el usuario no tiene conexión
- Manejar eventos `sync` en el Service Worker
- Ejemplo práctico: enviar datos de formulario cuando se recupere la conexión

## MÓDULO 5: Notificaciones Push

### Temas:

- Arquitectura Push vs Notifications
- Conceptos clave:
  - Push Server
  - VAPID keys
  - Suscripción a notificaciones
- APIs involucradas:
  - `PushManager`
  - `Notification`
  - `ServiceWorkerRegistration.showNotification()`

### Práctica:

- Solicitar permiso de notificación al usuario
- Registrar un suscriptor con `PushManager`
- Recibir y mostrar notificaciones push desde el Service Worker
- Enviar una notificación desde una API externa (opcional, demo con herramientas simples)

## MÓDULO 6: Estrategias Offline-First

### Temas:

- ¿Qué significa "Offline First"?
- Estrategias de manejo de datos offline:
  - Cachear respuestas de API
  - Almacenar peticiones fallidas y reintentarlas (usando IndexedDB)
  - Priorizar contenido local
- Integración con IndexedDB para persistencia de datos locales

### Práctica:

- Crear un sistema que muestre datos desde caché o IndexedDB si no hay red
- Usar `fetch` para actualizar datos cuando haya conexión

## MÓDULO 7: Introducción a Workbox (cuando sea conveniente)

### Temas:

- Limitaciones de hacer todo manualmente
- ¿Qué es Workbox?
- Principales módulos de Workbox:
  - `workbox-routing`
  - `workbox-strategies`
  - `workbox-cacheable-response`
  - `workbox-background-sync`
  - `workbox-expiration`
- Generación automática de SW con Workbox CLI

### Práctica:

- Convertir un Service Worker manual a uno usando Workbox
- Implementar estrategias avanzadas con menos código
- Agregar sincronización background y expiración de cachés fácilmente

## MÓDULO 8: Características Nativas y Capacidades del Navegador

### Temas:

- Acceso a dispositivos nativos mediante APIs web:
  - Geolocalización ( `Geolocation API` )
  - Cámara y micrófono ( `getUserMedia` )
  - Acelerómetro y sensores ( `DeviceMotion` , `DeviceOrientation` )
  - Acceso a contactos (con permisos)
- Detección de conectividad ( `navigator.onLine` )
- Fullscreen API
- Web Share API

### Práctica:

- Solicitar geolocalización del usuario y mostrar en mapa
- Tomar una foto desde el navegador y mostrarla
- Detectar cambios en la conexión a internet y mostrar estado

## MÓDULO 9: Proyecto Final – Construcción de una PWA completa

### Desafío:

Construir una PWA funcional con las siguientes características:

- Manifest configurado correctamente
- Service Worker registrado y funcional
- Caché de recursos estáticos y dinámicos
- Funcionamiento offline-first
- Soporte de notificaciones push
- Sincronización en segundo plano
- Uso de al menos dos características nativas (geolocalización, cámara, etc.)

## Recursos Adicionales

- Documentación oficial de MDN Web Docs:
  - Service Workers
  - Cache Storage
- Workbox Docs: https://developers.google.com/web/tools/workbox
- PWA Builder: https://www.pwabuilder.com
- Can I Use para compatibilidad: https://caniuse.com

---

# MÓDULO 1: Introducción a las Progressive Web Apps (PWA)

## Duración sugerida: 2 horas

## Objetivo del módulo:

Introducir a los participantes al concepto de **Progressive Web Apps (PWA)**, su importancia en el desarrollo web moderno y los componentes básicos que las conforman. Al finalizar este módulo, los asistentes deberán entender qué es una PWA, cuáles son sus ventajas, cómo se diferencia de una aplicación tradicional y qué requisitos técnicos requiere para funcionar correctamente.

## Temario detallado:

### 1. ¿Qué es una PWA? (Progressive Web App)

- Definición: Aplicaciones web que ofrecen experiencia similar a una app nativa.
- Características principales:
  - **Progresivas**: Funcionan en cualquier navegador.
  - **Reactivas**: Se adaptan a diferentes dispositivos.
  - **Conectables**: Funcionan sin conexión gracias al **Service Worker**.
  - **Instalables**: Se pueden agregar a la pantalla de inicio.
  - **Seguras**: Solo funcionan bajo HTTPS.
  - **Enlazables**: Se comparten mediante URLs normales.

> **Ejemplo práctico:** WhatsApp Web como PWA o Twitter Lite.

### 2. Ventajas frente a aplicaciones nativas y web tradicional

| Característica | Aplicación Web Tradicional | Aplicación Nativa | PWA |
|---|---|---|---|
| Acceso | Desde navegador | Instalación previa | Navegador + instalable |
| Requiere conexión | Sí | Depende | Puede funcionar offline |
| Actualización | Manual / página web | App Store / Play Store | Automática |
| Descubrimiento SEO | Sí | No siempre | Sí |
| Notificaciones push | No | Sí | Sí (con Service Worker) |

| Característica | Aplicación Web Tradicional | Aplicación Nativa | PWA |
| --- | --- | --- | --- |
| Instalación | No | Sí | Sí (sin tienda) |

> 📌 **Ventaja clave de las PWA:** No necesitas ir a una tienda de apps para instalarla.

## 3. Componentes técnicos de una PWA

Una PWA no es solo un tipo de diseño o tecnología, sino una combinación de ciertos estándares y tecnologías web:

### a) HTTPS

- Es obligatorio: garantiza seguridad y privacidad
- Sin HTTPS, muchas APIs no funcionan (como geolocalización, notificaciones push)

### b) Web App Manifest

- Archivo JSON que define metadatos de la aplicación:
  - Nombre
  - Iconos
  - Color de tema
  - Pantalla inicial
  - Orientación predeterminada
- Permite que la aplicación se "instale" desde el navegador

```
// Ejemplo de manifest.json
{
  "name": "Mi Primera PWA",
  "short_name": "PWA Demo",
  "start_url": "/index.html",
  "display": "standalone",
  "background_color": "#ffffff",
  "theme_color": "#000000",
  "icons": [
    {
      "src": "icon-192x192.png",
      "sizes": "192x192",
      "type": "image/png"
    }
  ]
}
```

### c) Service Worker

- Es el motor detrás de la funcionalidad offline
- Actúa como proxy entre la aplicación y la red
- Permite caché inteligente, notificaciones push y sincronización en segundo plano

## 4. Entorno de desarrollo básico

### Herramientas recomendadas:

- Editor de código: **Visual Studio Code**
- Servidor local:
  - `Live Server` (extensión de VSCode)
  - `http-server` (npm)
  - Python con `python -m http.server`
- Herramientas de auditoría:
  - **Lighthouse** (en Chrome DevTools)
  - Web.dev/measure

> Importante: El proyecto debe estar bajo **HTTPS** para probar todas las funcionalidades (excepto en localhost).

## 5. Primeros pasos: Estructura básica de una PWA

### Paso 1: Crear archivo `manifest.json`

- Guardarlo en `/manifest.json`
- Agregar enlace en el `<head>` del HTML:

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

### Paso 2: Registrar un Service Worker (vacío por ahora)

- Crear archivo `sw.js` vacío
- Registrarlo desde el cliente:

```
if ('serviceWorker' in navigator) {
  navigator.serviceWorker.register('/sw.js')
    .then(reg => console.log('Service Worker registrado', reg))
    .catch(err => console.error('Error al registrar SW:', err));
}
```

### Paso 3: Probar con Lighthouse

- Abrir DevTools > Lighthouse
- Ejecutar auditoría PWA
- Verificar si faltan elementos por completar

## Actividad práctica

### Objetivo:

Crear una estructura mínima funcional de PWA con:

- Un `manifest.json` básico
- Registro de un Service Worker vacío
- Comprobación con Lighthouse

### Instrucciones:

1. Crea una carpeta llamada `mi-pwa`.
2. Dentro, crea:
    - `index.html`
    - `manifest.json`
    - `sw.js`
    - Un icono PNG de 192x192px
3. Registra el Service Worker.
4. Abre la página en Chrome y usa **Lighthouse** para comprobar si tu sitio cumple con los criterios básicos de PWA.

## Resumen del módulo

| Concepto | Descripción | ||-| | PWA | Aplicación web que ofrece experiencia nativa | | Ventajas | Offline, instalable, rápida, segura, indexable | | Componentes | HTTPS, Web App Manifest, Service Worker | | Herramientas | VSCode, servidor local, Lighthouse | | Práctica | Configurar estructura básica de PWA |

## Recursos recomendados

- MDN Web Docs – PWA
- Google Developers – PWA
- Web.dev – Fundamentos de PWA
- Lighthouse Audit

# MÓDULO 2: Fundamentos de los Service Workers

## ⏱ Duración sugerida: 3 horas

## 🎯 Objetivo del módulo:

Introducir a los participantes en el concepto, ciclo de vida y uso básico de los **Service Workers**, entendiendo cómo funcionan como pieza clave para construir aplicaciones web progresivas (PWA). Al finalizar este módulo, los asistentes deberán saber cómo registrar un Service Worker, entender su ciclo de vida e implementar una primera versión funcional.

## Temario detallado:

### 1. ¿Qué es un Service Worker?

- Es un script que se ejecuta en segundo plano del navegador.
- No tiene acceso al DOM.
- Actúa como intermediario entre la aplicación y la red.
- Permite funcionalidades avanzadas como:
    - Caché offline

- Notificaciones push
- Sincronización en segundo plano
- Interceptación de peticiones HTTP

> Piensa en él como un proxy cliente-side.

## 2. Requisitos técnicos

- Debe estar alojado en el mismo dominio o subdominio que la aplicación.
- Funciona únicamente bajo **HTTPS** (excepto en `localhost`).
- No puede acceder directamente al DOM ni usar `window`.

## 3. Ciclo de vida del Service Worker

Los Service Workers tienen un ciclo de vida diferente al de las páginas web. Se compone de tres fases principales:

### a) Registro

- El navegador descarga el archivo del Service Worker desde la URL especificada.
- Se inicia el proceso de instalación si no existe uno activo o hay cambios.

```
if ('serviceWorker' in navigator) {
  navigator.serviceWorker.register('/sw.js')
    .then(reg => console.log('SW registrado', reg))
    .catch(err => console.error('Error registrando SW:', err));
}
```

### b) Instalación (`install`)

- Ocurre solo cuando:
  - Es la primera vez
  - El archivo del Service Worker cambia
- Aquí puedes cachear recursos estáticos.

```
self.addEventListener('install', event => {
  console.log('Service Worker instalándose...');
  // Puedes cachear archivos aquí
});
```

### c) Activación (`activate`)

- Ocurre después de la instalación
- Aquí puedes limpiar caches antiguos o migrar IndexedDB

```
self.addEventListener('activate', event => {
  console.log('Service Worker activado');
  // Limpiar cachés viejos
});
```

> Importante: Para forzar una actualización del Service Worker, cambia su contenido o fuerza una recarga con `Ctrl + Shift + R`.

## 4. Comunicación entre página y Service Worker

El Service Worker no puede manipular el DOM, pero sí comunicarse con las ventanas abiertas usando `postMessage()`.

### Desde el cliente:

```
const swRegistration = await navigator.serviceWorker.ready;
swRegistration.active.postMessage({ msg: 'Hola Service Worker!' });
```

### En el Service Worker:

```
self.addEventListener('message', event => {
  console.log('Mensaje recibido:', event.data);
  event.source.postMessage({ reply: 'Hola desde SW!' });
});
```

> Útil para sincronizar estados, mostrar notificaciones o controlar eventos globales.

## 5. Buenas prácticas y consideraciones

- Siempre registra el Service Worker en el ámbito correcto (por ejemplo, raíz `/`).
- Usa `console.log()` dentro del Service Worker para debuggear en DevTools > Application > Service Workers.
- Evita hacer operaciones pesadas durante `install` o `activate`.
- Recuerda: el Service Worker se ejecuta en un hilo separado.

## 6. Práctica guiada: Registro y ciclo de vida

### Objetivo:

Crear un Service Worker básico que muestre mensajes en consola durante cada fase del ciclo de vida.

### Pasos:

1. Crea un proyecto básico con:
   - `index.html`
   - `sw.js`
2. Registra el Service Worker desde `index.html`
3. Agrega escuchadores para `install`, `activate` y `message`
4. Abre DevTools y verifica el registro y los logs
5. Realiza cambios en `sw.js` y forza una actualización

### Código sugerido:

**index.html**

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <title>Mi Primer Service Worker</title>
  <link rel="manifest" href="/manifest.json">
</head>
<body>
  <h1>Hola PWA</h1>
  <script src="app.js"></script>
</body>
</html>
```

**app.js**

```
if ('serviceWorker' in navigator) {
  navigator.serviceWorker.register('/sw.js')
    .then(reg => console.log('Service Worker registrado', reg))
    .catch(err => console.error('Error al registrar SW:', err));
}
```

**sw.js**

```
self.addEventListener('install', event => {
  console.log('[SW] Instalándose...');
});

self.addEventListener('activate', event => {
  console.log('[SW] Activado');
});

self.addEventListener('message', event => {
  console.log('[SW] Mensaje recibido:', event.data);
  event.ports[0].postMessage({ respuesta: 'Hola desde SW' });
});
```

# Resumen del módulo

| Concepto | Descripción | ||-| | Service Worker | Script que vive fuera del DOM y controla la app | | Ciclo de vida | `install`, `activate`, `fetch`, `message`, etc. | | Comunicación | Usar `postMessage()` entre cliente y SW | | Requisitos | HTTPS, mismo origen, sin acceso al DOM | | Práctica | Registrar SW y mostrar logs durante su ciclo de vida |

## Recursos recomendados

- MDN Web Docs – Service Worker
- Google Developers – Service Worker Introduction
- Chrome DevTools – Service Workers

# MÓDULO 3: Uso del Cache Storage

## ⏱ Duración sugerida: 4 horas

## 🎯 Objetivo del módulo:

Familiarizar a los participantes con el uso de `CacheStorage`, una API clave para almacenar recursos en caché y permitir funcionalidad offline-first. Al finalizar este módulo, los asistentes deberán saber cómo usar esta API para almacenar, recuperar y actualizar recursos desde un Service Worker.

## Temario detallado:

### 1. ¿Qué es `CacheStorage`?

- Es una API que permite almacenar solicitudes y respuestas HTTP.
- Funciona únicamente dentro del Service Worker.
- Permite cachear archivos estáticos (CSS, JS, imágenes) o dinámicos (APIs).

> 🧠 Similar al localStorage, pero orientado a peticiones HTTP.

### 2. Métodos principales de `CacheStorage`

#### a) `caches.open(nombreDelCache)`

Abre o crea un nuevo almacén de caché.

```
caches.open('mi-cache-v1')
```

#### b) `cache.put(request, response)`

Guarda una solicitud y su respuesta.

```
cache.put('/index.html', response);
```

#### c) `cache.match(request)`

Busca una solicitud en el caché y devuelve su respuesta si existe.

```
cache.match(event.request)
```

#### d) `cache.addAll(urls)`

Cachear varios recursos a la vez (útil durante instalación).

```
cache.addAll(['/index.html', '/style.css']);
```

#### e) `caches.delete(nombreDelCache)`

Elimina un cache específico.

```
caches.delete('mi-cache-v1');
```

### 3. Ciclo de vida del caché

### a) Cachear durante la instalación (`install`)

Ideal para recursos estáticos que no cambian frecuentemente.

```
self.addEventListener('install', event => {
  event.waitUntil(
    caches.open('mi-cache-v1')
      .then(cache => cache.addAll([
        '/',
        '/index.html',
        '/style.css',
        '/script.js'
      ]))
  );
});
```

### b) Interceptar peticiones (`fetch`)

Usamos el evento `fetch` para responder desde el caché o red.

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

## 4. Estrategias básicas de caché

### a) Cache-only

Solo usa lo que está en caché. No intenta ir a la red.

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

### b) Network-only

Ignora el caché y siempre va a la red.

```
event.respondWith(fetch(event.request));
```

### c) Cache-first

Intenta obtener recurso del caché; si no está, va a la red.

```
event.respondWith(
  caches.match(event.request)
    .then(response => response || fetch(event.request))
);
```

### d) Network-first

Primero intenta desde la red, si falla usa el caché.

```
event.respondWith(
  fetch(event.request).catch(() => caches.match(event.request))
);
```

### e) Stale-while-revalidate (manual)

Devuelve caché mientras actualiza desde la red (requiere clonar respuesta).

```
self.addEventListener('fetch', event => {
  const respuestaCache = caches.match(event.request);
  const respuestaRed = fetch(event.request).then(resp => {
    return caches.open('mi-cache-v1').then(cache => {
      cache.put(event.request, resp.clone());
      return resp;
    });
  });

  event.respondWith(Promise.race([respuestaRed, respuestaCache]));
});
```

## 5. Práctica guiada: Implementando estrategia `cache-first`

### Objetivo:

Crear un Service Worker que cargue recursos desde caché cuando estén disponibles, y vaya a la red solo si no están.

### Pasos:

1. En tu proyecto existente, agrega un archivo CSS y uno JS.
2. Modifica el Service Worker para:
    - Cachear esos archivos durante la instalación
    - Usar `cache-first` durante `fetch`
3. Prueba desconectarte de internet y recargar la página.

### Código sugerido:

**sw.js**

```
const CACHE_NAME = 'mi-cache-v1';

self.addEventListener('install', event => {
  event.waitUntil(
    caches.open(CACHE_NAME)
      .then(cache => {
        return cache.addAll([
          '/',
          '/index.html',
          '/style.css',
          '/script.js'
        ]);
      })
  );
});

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

## 6. Buenas prácticas y consideraciones

- Usa versiones diferentes de caché para evitar conflictos ( `v1` , `v2` , etc.)
- Limpia cachés antiguos durante el evento `activate` .
- Evita cachear demasiados archivos grandes (puede afectar rendimiento).
- Siempre prueba en modo offline desde DevTools.

### Ejemplo de limpieza de caché antiguo:

```
self.addEventListener('activate', event => {
  const cacheWhitelist = [CACHE_NAME];

  event.waitUntil(
    caches.keys().then(keys => {
      return Promise.all(keys.map(key => {
        if (!cacheWhitelist.includes(key)) {
          return caches.delete(key);
        }
      }));
    })
  );
});
```

## ✅ Resumen del módulo

| Concepto | Descripción | ||-| | `CacheStorage` | API para almacenar recursos HTTP | | `caches.open()` | Crea o abre un grupo de caché | | `cache.match()` | Busca una solicitud en caché | | Estrategias | Cache-first, Network-first, Stale-while-revalidate | | Práctica | Implementar un Service Worker que use caché inteligente | | Limpieza | Eliminar cachés antiguos en `activate` |

## Recursos recomendados

- MDN Web Docs – Cache
- Google Developers – Using CacheStorage
- Offline Cookbook (con ejemplos)

# MÓDULO 4: Sincronización en Segundo Plano (Background Sync)

## ⏱ Duración sugerida: 3 horas

## 🎯 Objetivo del módulo:

Introducir a los participantes en la **sincronización en segundo plano**, una funcionalidad clave para aplicaciones que deben operar sin conexión. Al finalizar este módulo, los asistentes deberán entender cómo usar la API `SyncManager` para programar tareas que se ejecuten cuando haya conexión, y serán capaces de implementar un ejemplo práctico como el envío de formularios o datos pendientes.

## Temario detallado:

### 1. ¿Qué es la sincronización en segundo plano?

- Es una funcionalidad que permite a las PWA **programar tareas que se ejecutan cuando hay conexión disponible**.
- Ideal para:
  - Enviar datos desde formularios
  - Actualizar información del servidor
  - Sincronizar cambios locales

> 📶 Útil para usuarios móviles con conexiones intermitentes.

### 2. APIs involucradas

#### a) Background Sync API

Permite registrar "tareas de sincronización" que el navegador ejecutará cuando sea posible.

#### b) Service Worker

Recibe el evento `sync` y realiza la acción programada (ej.: enviar datos).

#### c) IndexedDB *(opcional)*

Se usa comúnmente para almacenar datos locales mientras no hay conexión.

### 3. Tipos de sincronización

#### a) One-off sync

- Se programa una vez
- El navegador intenta ejecutarla cuando hay conexión

```
navigator.serviceWorker.ready.then(reg => {
  reg.sync.register('enviar-formulario');
});
```

### b) Periodic sync *(limitado)*

- Permite sincronizaciones periódicas
- Solo funciona bajo ciertas condiciones (permisos, PWA instalada, etc.)

> ⚠ Nota: Esta funcionalidad aún tiene soporte limitado y requiere permisos especiales.

## 4. Ciclo de trabajo de Background Sync

1. **Cliente:** Detecta que no hay conexión y almacena datos localmente (ej.: IndexedDB)
2. **Cliente:** Programa una tarea de sincronización cuando hay datos pendientes
3. **Service Worker:** Recibe el evento `sync`
4. **Service Worker:** Ejecuta la lógica necesaria (ej.: enviar datos por fetch)
5. **Service Worker:** Confirma éxito o reintenta si falla

## 5. Implementación paso a paso

### Paso 1: Registrar el Service Worker

```
// app.js
if ('serviceWorker' in navigator && 'SyncManager' in window) {
  navigator.serviceWorker.register('/sw.js')
    .then(reg => console.log('SW registrado'))
    .catch(err => console.error('Error registrando SW:', err));
}
```

### Paso 2: Programar una sincronización

```
// app.js
function guardarYProgramar(datos) {
  // Guardar datos localmente (ej.: localStorage o IndexedDB)
  localStorage.setItem('formulario-pendiente', JSON.stringify(datos));

  // Programar sincronización
  navigator.serviceWorker.ready.then(reg => {
    reg.sync.register('enviar-formulario')
      .then(() => console.log('Sincronización programada'));
  });
}
```

### Paso 3: Manejar el evento `sync` en el Service Worker

```
// sw.js
self.addEventListener('sync', event => {
  if (event.tag === 'enviar-formulario') {
    event.waitUntil(
      (async () => {
        const datos = localStorage.getItem('formulario-pendiente');

        if (datos) {
          try {
            const response = await fetch('https://api.ejemplo.com/enviar', {
              method: 'POST',
              body: datos,
              headers: { 'Content-Type': 'application/json' }
            });

            if (response.ok) {
              localStorage.removeItem('formulario-pendiente');
              console.log('Datos enviados correctamente');
            } else {
              throw new Error('Fallo al enviar datos');
            }
          } catch (error) {
            console.error('No se pudo enviar:', error);
          }
        }
      })()
    );
  }
});
```

> ⚠ Nota: Este ejemplo usa `localStorage`, pero en producción se recomienda **IndexedDB** para mayor escalabilidad y seguridad.

## 6. Práctica guiada: Formulario offline-first

### Objetivo:

Crear un formulario que funcione incluso sin conexión, guarde los datos localmente y los envíe cuando haya conexión.

### Pasos:

1. Crear un formulario HTML simple.
2. Enviar datos mediante JavaScript.
3. Si no hay conexión, guardar datos localmente.
4. Programar sincronización con `SyncManager`.
5. En el Service Worker, interceptar evento `sync` y enviar datos.

### Código sugerido:

**index.html**

```
<form id="formulario">
  <input type="text" name="nombre" placeholder="Tu nombre" required>
  <button type="submit">Enviar</button>
</form>
<p id="estado"></p>
```

**app.js**

```
document.getElementById('formulario').addEventListener('submit', async function(e) {
  e.preventDefault();
  const nombre = this.nombre.value;
  const datos = { nombre };

  if (navigator.onLine) {
    try {
      const res = await fetch('https://api.ejemplo.com/enviar', {
        method: 'POST',
        body: JSON.stringify(datos),
        headers: { 'Content-Type': 'application/json' }
      });
      document.getElementById('estado').textContent = 'Enviado!';
    } catch (err) {
      document.getElementById('estado').textContent = 'Hubo un error.';
    }
  } else {
    localStorage.setItem('formulario-pendiente', JSON.stringify(datos));
    document.getElementById('estado').textContent = 'Guardado localmente...';

    navigator.serviceWorker.ready.then(reg => {
      reg.sync.register('enviar-formulario')
        .then(() => console.log('Sincronización programada'));
    });
  }
});
```

**sw.js**

```
self.addEventListener('sync', event => {
  if (event.tag === 'enviar-formulario') {
    event.waitUntil(
      (async () => {
        const datos = localStorage.getItem('formulario-pendiente');
        if (!datos) return;

        try {
          const res = await fetch('https://api.ejemplo.com/enviar', {
            method: 'POST',
            body: datos,
            headers: { 'Content-Type': 'application/json' }
          });

          if (res.ok) {
            localStorage.removeItem('formulario-pendiente');
            console.log('Datos sincronizados');
          }
        } catch (err) {
          console.error('Fallo en sincronización:', err);
        }
      })()
    );
  }
});
```

## 🧠 Resumen del módulo

| Concepto | Descripción |
|---|---|
| Background Sync | Permite programar acciones que se ejecutan con conexión |
| SyncManager | API que registra tareas de sincronización |
| Evento `sync` | Interceptado en el Service Worker |
| Estrategia | Guardar datos localmente y reintentar cuando haya conexión |
| Práctica | Formulario que funciona offline y se sincroniza después |

## Recursos recomendados

- MDN Web Docs – SyncManager

- Google Developers – Background Sync
- Jake Archibald – Background Sync
- Web.dev – Background Sync

# MÓDULO 5: Notificaciones Push

## ⏱ Duración sugerida: 4 horas

## 🎯 Objetivo del módulo:

Introducir a los participantes en las **notificaciones push** para Progressive Web Apps. Al finalizar este módulo, los asistentes deberán entender cómo funciona el flujo de notificaciones push, cómo solicitar permisos al usuario, cómo suscribirse al servicio de notificaciones y cómo recibir y mostrar una notificación desde el Service Worker.

> ⚠ **Nota:** Para este módulo se asume que ya tienes un backend funcional (por ejemplo, una API REST en ASP.NET Core). Se indicará qué endpoints crear o consumir sin entrar en detalles profundos de implementación.

## Temario detallado:

### 1. ¿Qué son las notificaciones push?

- Son mensajes que se muestran al usuario incluso cuando la aplicación no está abierta.
- Funcionan gracias a:
  - **Service Worker**
  - **Push API**
  - **Notification API**
  - **Servidor Push (VAPID)**

> 🔔 Similar a las notificaciones nativas de apps móviles, pero en el navegador.

### 2. Arquitectura general de Push Notifications

| Componente | Descripción |
|---|---|
| Navegador | Gestiona la suscripción del usuario |
| Service Worker | Recibe y muestra las notificaciones |
| Push Server | Envia mensaje a través de un servicio como Firebase Cloud Messaging |
| Backend App | Controla cuándo y qué enviar |

### 3. Requisitos técnicos

- HTTPS (obligatorio)
- PWA instalable
- Soporte del navegador (Chrome, Edge, Firefox, etc.)
- Claves VAPID (Voluntary Application Server Identification)

### 4. Flujo de trabajo de notificaciones push

1. **Solicitar permiso al usuario**
2. **Registrar el Service Worker**
3. **Obtener una suscripción PushManager**
4. **Enviar la suscripción al backend**
5. **Backend almacena la suscripción**
6. **Backend envía notificación usando claves VAPID**
7. **Navegador recibe notificación y el Service Worker la muestra**

### 5. Solicitar permiso de notificación

```
Notification.requestPermission().then(result => {
  if (result === 'granted') {
    console.log('Permiso concedido');
  } else {
    console.log('Permiso denegado o no otorgado');
  }
});
```

### 6. Suscribirse al servicio Push

```
navigator.serviceWorker.ready.then(reg => {
  reg.pushManager.subscribe({
    userVisibleOnly: true,
    applicationServerKey: urlBase64ToUint8Array('TU_PUBLIC_VAPID_KEY')
  }).then(subscription => {
    console.log('Usuario suscrito:', subscription);

    // Enviar suscripción al backend
    fetch('/api/subscribe', {
      method: 'POST',
      body: JSON.stringify(subscription),
      headers: {
        'Content-Type': 'application/json'
      }
    });
  });
});

// Función auxiliar para convertir VAPID Key
function urlBase64ToUint8Array(base64String) {
  const padding = '='.repeat((4 - base64String.length % 4) % 4);
  const base64 = (base64String + padding)
    .replace(/-/g, '+')
    .replace(/_/g, '/');

  const rawData = window.atob(base64);
  const outputArray = new Uint8Array(rawData.length);

  for (let i = 0; i < rawData.length; ++i) {
    outputArray[i] = rawData.charCodeAt(i);
  }

  return outputArray;
}
```

### 7. Mostrar una notificación en el Service Worker

```
self.addEventListener('push', event => {
  const data = event.data.json();
  const options = {
    body: data.body,
    icon: 'icon-192x192.png',
    badge: 'badge.png',
    vibrate: [100, 50, 100],
    data: {
      url: data.url || '/'
    }
  };

  event.waitUntil(
    self.registration.showNotification(data.title, options)
  );
});

self.addEventListener('notificationclick', event => {
  event.notification.close();

  event.waitUntil(
    clients.openWindow(event.notification.data.url)
  );
});
```

## 8. Endpoints del backend (ASP.NET Core Web API)

### a) `/api/subscribe` – POST

- **Objetivo:** Guardar la suscripción del cliente
- **Request Body Ejemplo:**

```
{
  "endpoint": "https://fcm.googleapis.com/fcm/send/...",
  "keys": {
    "p256dh": "BNOrw8T7tDRKUJy9zXjNYu...",
    "auth": "d9sDI..."
  }
}
```

- **Acción recomendada:** Guardar esta suscripción en base de datos

### b) `/api/push` – POST

- **Objetivo:** Enviar una notificación push a todos los usuarios suscritos
- **Lógica interna:** Usar una librería como `WebPush` en .NET para enviar el mensaje con las claves VAPID
- **Ejemplo de payload:**

```
{
  "title": "¡Hola!",
  "body": "Esta es una notificación push",
  "url": "/dashboard"
}
```

## 9. Práctica guiada: Sistema de notificaciones push básico

### Objetivo:

Crear un sistema que permita:

- Registrar al usuario para recibir notificaciones
- Enviar una notificación desde el backend
- Mostrarla en el navegador incluso si la página no está abierta

### Pasos:

1. Crear botón para pedir permiso de notificación
2. Registrar suscripción y enviarla al endpoint `/api/subscribe`
3. Enviar una notificación desde el backend al endpoint `/api/push`
4. Interceptarlo en el Service Worker y mostrarlo con `showNotification()`

### 10. Buenas prácticas y consideraciones

- Siempre verificar el estado del permiso antes de pedirlo.
- Manejar errores de suscripción (ej.: usuario niega permiso).
- Usar claves VAPID seguras y almacenarlas en variables de entorno.
- Probar notificaciones en distintos navegadores y dispositivos.
- No saturar al usuario con notificaciones innecesarias.

## Resumen del módulo

| Concepto | Descripción | ||-| | Push Notifications | Mensajes que llegan aunque la app no esté abierta | | Push API | Registra suscripciones de usuario | | Notification API | Muestra mensajes en pantalla | | Service Worker | Intercepta notificaciones y las muestra | | Backend | Envía mensajes usando VAPID | | Endpoints | `/api/subscribe`, `/api/push` |

## Recursos recomendados

- MDN Web Docs – Push API
- Google Developers – Push Notifications
- WebPush C# Library
- VAPID Generator Tool

# MÓDULO 6: Estrategias Offline-First

## Duración sugerida: 4 horas

## Objetivo del módulo:

Introducir a los participantes en las **estrategias offline-first**, enfocadas en construir aplicaciones que funcionen perfectamente sin conexión. Al finalizar este módulo, los asistentes deberán entender cómo diseñar una aplicación para priorizar datos locales, usar IndexedDB como base de datos cliente y combinarla con estrategias de caché y sincronización.

## Temario detallado:

### 1. ¿Qué significa "Offline First"?

- Prioriza el uso de datos locales almacenados en el navegador.
- Si no hay conexión, se muestra contenido desde caché o base de datos local.
- Si hay conexión, actualiza datos y sincroniza cambios.

> El usuario debe poder usar la app aunque esté desconectado.

### 2. Ventajas del enfoque offline-first

| Ventaja | Descripción |
|---|---|
| Rendimiento | Menos dependencia de red mejora velocidad |
| Experiencia | La aplicación sigue siendo usable sin internet |
| Fiabilidad | Funciona incluso en zonas con conexión intermitente |
| Sincronización | Cambios se almacenan localmente y se envían cuando hay red |

### 3. Componentes clave de una arquitectura offline-first

| Componente | Propósito |
|---|---|
| `Service Worker` | Intercepta peticiones y sirve desde caché |
| `CacheStorage` | Almacena recursos estáticos (CSS, JS, imágenes) |

| Componente | Propósito |
| --- | --- |
| `IndexedDB` | Base de datos cliente para datos estructurados |
| `Background Sync` | Sincroniza cambios cuando hay conexión |
| `LocalStorage / SessionStorage` | Almacenamiento simple (no recomendado para grandes volúmenes) |

## 4. Uso de IndexedDB – Conceptos básicos

### a) ¿Qué es IndexedDB?

- Es una **base de datos NoSQL** integrada en el navegador.
- Permite almacenar objetos estructurados.
- Ideal para guardar datos complejos cuando no hay conexión.

### b) Flujo básico de uso

1. Abrir/conectar a la base de datos
2. Crear un objeto de almacenamiento ( `objectStore` )
3. Insertar, actualizar, eliminar o leer registros
4. Cerrar conexión cuando termines

### c) Ejemplo básico de conexión e inserción

```
const request = indexedDB.open("MiAppDB", 1);

request.onupgradeneeded = function(event) {
  const db = event.target.result;
  if (!db.objectStoreNames.contains('datos')) {
    db.createObjectStore('datos', { keyPath: 'id' });
  }
};

request.onsuccess = function(event) {
  const db = event.target.result;

  const tx = db.transaction('datos', 'readwrite');
  const store = tx.objectStore('datos');

  store.put({
    id: 'usuario_001',
    nombre: 'Juan',
    timestamp: Date.now()
  });

  tx.oncomplete = () => db.close();
};
```

## 5. Estrategias comunes de manejo de datos offline

### a) Cache + Network (Stale-while-revalidate)

- Devuelve datos desde caché mientras refresca desde red.
- Ideal para listas dinámicas.

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

  event.waitUntil(
    fetch(event.request).then(response => {
      return caches.open('dinamico-v1').then(cache => cache.put(event.request, response));
    })
  );
});
```

### b) Datos locales primero

- Usar IndexedDB para mostrar información local si no hay conexión.

```
function obtenerDatos() {
  if (navigator.onLine) {
    fetch('/api/datos')
      .then(res => res.json())
      .then(datos => {
        guardarEnIndexedDB(datos);
        mostrarDatos(datos);
      });
  } else {
    obtenerDesdeIndexedDB().then(datos => mostrarDatos(datos));
  }
}
```

### c) Cola de operaciones pendientes

- Guardar cambios locales en IndexedDB
- Enviarlos al servidor cuando haya conexión usando Background Sync

## 6. Práctica guiada: App de tareas offline-first

### Objetivo:

Crear una aplicación que permita agregar, ver y sincronizar tareas incluso sin conexión.

### Pasos:

1. Mostrar lista de tareas desde IndexedDB
2. Si hay conexión, también cargar desde API REST
3. Agregar nuevas tareas y guardarlas en IndexedDB
4. Programar sincronización con Background Sync
5. En Service Worker, enviar tareas pendientes al backend

### Código sugerido:

**app.js**

```
document.getElementById('form-tarea').addEventListener('submit', function(e) {
  e.preventDefault();
  const tarea = this.tarea.value;

  const nuevaTarea = {
    id: 'tarea_' + Date.now(),
    texto: tarea,
    estado: 'pendiente'
  };

  if (navigator.onLine) {
    fetch('/api/tareas', {
      method: 'POST',
      body: JSON.stringify(nuevaTarea),
      headers: { 'Content-Type': 'application/json' }
    }).then(() => mostrarTarea(nuevaTarea));
  } else {
    guardarTareaLocal(nuevaTarea);
    navigator.serviceWorker.ready.then(reg => reg.sync.register('sync-tareas'));
    mostrarTarea(nuevaTarea);
  }
});
```

**sw.js**

```
self.addEventListener('sync', event => {
  if (event.tag === 'sync-tareas') {
    event.waitUntil(
      (async () => {
        const tareas = await obtenerTareasPendientes();

        for (const tarea of tareas) {
          try {
            await fetch('/api/tareas', {
              method: 'POST',
              body: JSON.stringify(tarea),
              headers: { 'Content-Type': 'application/json' }
            });
            await eliminarTareaLocal(tarea.id);
          } catch (err) {
            console.error('No se pudo sincronizar:', tarea.id);
          }
        }
      })()
    );
  }
});
```

### 7. Buenas prácticas

- Usa `navigator.onLine` para detectar cambios de estado de red.
- Evita depender únicamente de `localStorage` para grandes cantidades de datos.
- Limpia datos locales una vez sincronizados.
- Maneja errores de sincronización y reintenta si es posible.
- Prioriza datos locales si no hay conexión.

## Resumen del módulo

| Concepto | Descripción | ||-| | Offline-first | Arquitectura centrada en datos locales | | IndexedDB | Base de datos cliente para almacenamiento estructurado | | CacheStorage | Almacena recursos estáticos | | Background Sync | Envía datos cuando hay conexión | | Práctica | Tareas guardadas localmente y sincronizadas después |

## Recursos recomendados

- MDN Web Docs – IndexedDB

- Google Developers – Offline UX
- Jake Archibald – Offline Cookbook
- Workbox Background Sync

# MÓDULO 7: Introducción a Workbox

## ⏱ Duración sugerida: 3 horas

## 🎯 Objetivo del módulo:

Introducir a los participantes en **Workbox**, una biblioteca desarrollada por Google que facilita la creación y gestión de Service Workers. Al finalizar este módulo, los asistentes deberán entender cómo usar Workbox para simplificar tareas como el manejo de caché, sincronización en segundo plano y notificaciones push, además de migrar código manual a estrategias basadas en Workbox.

## Temario detallado:

### 1. ¿Qué es Workbox?

- Es una colección de bibliotecas y herramientas para construir Progressive Web Apps (PWA).
- Facilita el uso de Service Workers con estrategias predefinidas.
- Permite generar automáticamente un Service Worker usando Workbox CLI.

> 🧠 Piénsa en él como "jQuery para Service Workers" – pero moderno y potente.

### 2. Ventajas de usar Workbox

| Ventaja | Descripción |
|---|---|
| Reducción de código | Estrategias listas para usar |
| Mantenimiento fácil | Mejor organización del Service Worker |
| Buenas prácticas integradas | Soporta caché versionado, expiración, etc. |
| Herramientas CLI | Genera SWs automáticamente |

### 3. Principales módulos de Workbox

#### a) `workbox-routing`

- Maneja las rutas interceptadas por el Service Worker
- Reemplaza el uso manual de `fetch` events

```
import { registerRoute } from 'workbox-routing';
```

#### b) `workbox-strategies`

- Ofrece estrategias predefinidas:
  - `CacheFirst`, `NetworkFirst`, `StaleWhileRevalidate`, etc.

```
import { CacheFirst } from 'workbox-strategies';
```

#### c) `workbox-cacheable-response`

- Filtra qué respuestas se deben cachear según criterios (ej.: status HTTP)

```
import { CacheableResponsePlugin } from 'workbox-cacheable-response';
```

#### d) `workbox-background-sync`

- Simplifica la implementación de Background Sync

```
import { BackgroundSyncPlugin } from 'workbox-background-sync';
```

### e) `workbox-expiration`

- Controla la expiración automática de elementos en caché

```
import { ExpirationPlugin } from 'workbox-expiration';
```

## 4. Instalación y configuración básica

### a) Opción 1: Usar CDN (recomendado para desarrollo rápido)

Agrega esto en tu archivo `sw.js`:

```
importScripts('https://storage.googleapis.com/workbox-cdn/releases/6.5.4/workbox-sw.js');

if (workbox) {
  console.log('Yay! Workbox is loaded 🎉');
} else {
  console.log('Boo! Workbox didn\'t load 😬');
}
```

> ⚠ Asegúrate de tener conexión a internet al probar esta opción.

### b) Opción 2: Descargar e importar localmente

- Descarga desde https://github.com/GoogleChrome/workbox
- Coloca los archivos en `/workbox/`
- Importa así:

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

## 5. Ejemplos prácticos con Workbox

### a) Cachear recursos estáticos con `CacheFirst`

```
import { registerRoute } from 'workbox-routing';
import { CacheFirst } from 'workbox-strategies';
import { CacheableResponsePlugin } from 'workbox-cacheable-response';

registerRoute(
  ({ request }) => request.destination === 'script' || request.destination === 'style',
  new CacheFirst({
    cacheName: 'recursos-estaticos',
    plugins: [
      new CacheableResponsePlugin({ statuses: [0, 200] }),
      new ExpirationPlugin({ maxEntries: 50, maxAgeSeconds: 30 * 24 * 60 * 60 }) // 30 días
    ]
  })
);
```

### b) Cachear imágenes con `StaleWhileRevalidate`

```
registerRoute(
  ({ request }) => request.destination === 'image',
  new StaleWhileRevalidate({
    cacheName: 'imagenes',
    plugins: [
      new ExpirationPlugin({ maxEntries: 60, maxAgeSeconds: 30 * 24 * 60 * 60 })
    ]
  })
);
```

### c) Sincronización background con `BackgroundSyncPlugin`

```
const bgSyncPlugin = new BackgroundSyncPlugin('cola-formularios', {
  maxRetentionTime: 24 * 60 // 24 horas
});

registerRoute(
  new RegExp('/api/formulario'),
  new NetworkOnly({
    plugins: [bgSyncPlugin]
  }),
  'POST'
);
```

### 6. Generación automática de Service Worker con Workbox CLI

#### a) Instalar Workbox CLI

```
npm install --save-dev workbox-cli
```

#### b) Configurar `workbox-config.js`

```
module.exports = {
  "globDirectory": "./",
  "globPatterns": ["**/*.{html,js,css,png,jpg,ico,json}"],
  "swDest": "sw.js"
};
```

#### c) Generar Service Worker

```
npx workbox generateSW
```

> Esto crea un Service Worker totalmente funcional con estrategias predeterminadas.

### 7. Práctica guiada: Convertir un Service Worker manual a Workbox

#### Objetivo:

Tomar un Service Worker creado manualmente en módulos anteriores y reemplazarlo por uno basado en Workbox.

#### Pasos:

1. Agregar Workbox via CDN o descarga local
2. Registrar rutas para cachear CSS, JS e imágenes
3. Implementar estrategia `StaleWhileRevalidate` para contenido dinámico
4. Probar funcionalidad offline

### 8. Buenas prácticas

- Usa Workbox CLI para automatizar generación de SWs en producción
- Siempre usa nombres de caché significativos y versionados
- Limpia cachés antiguos con `activate` event o `ExpirationPlugin`
- Combina varias estrategias según tipo de recurso
- Usa Workbox DevTools para debuggear en Chrome

## Resumen del módulo

| Concepto | Descripción | ||-| | Workbox | Biblioteca para facilitar el uso de Service Workers | | Módulos clave | `routing`, `strategies`, `background-sync`, `expiration` | | Estrategias | `CacheFirst`, `NetworkFirst`, `StaleWhileRevalidate` | | CLI | Genera SWs automáticamente | | Práctica | Convertir Service Worker manual a uno basado en Workbox |

## Recursos recomendados

- Workbox Docs – Google Developers
- Workbox GitHub Repo
- Workbox CLI Guide
- Workbox Recipes

# MÓDULO 8: Características nativas, localización, cámara y más

## ⏱️ Duración sugerida: 4 horas

## 🎯 Objetivo del módulo:

Introducir a los participantes en el uso de **APIs web modernas** que permiten acceder a características del dispositivo físico como geolocalización, cámara, sensores y más. Al finalizar este módulo, los asistentes deberán conocer las principales APIs disponibles en los navegadores modernos para construir aplicaciones web progresivas con funcionalidad nativa.

## Temario detallado:

### 1. ¿Qué son las características nativas en una PWA?

- Son funcionalidades del dispositivo físico (móvil o escritorio) a las que se puede acceder desde el navegador.
- Incluyen:
  - Geolocalización
  - Cámara y micrófono
  - Sensores de movimiento
  - Acceso a contactos
  - Acelerómetro
  - Fullscreen API
  - Web Share API

> 📱 Estas APIs convierten una aplicación web en una experiencia similar a una app nativa.

### 2. Geolocalización – `Geolocation API`

#### a) ¿Para qué sirve?

Permite obtener la ubicación actual del usuario (latitud y longitud).

#### b) Ejemplo básico

```
if ("geolocation" in navigator) {
  navigator.geolocation.getCurrentPosition(
    position => {
      console.log("Latitud:", position.coords.latitude);
      console.log("Longitud:", position.coords.longitude);
    },
    error => {
      console.error("Error al obtener ubicación:", error.message);
    }
  );
} else {
  console.log("Geolocalización no soportada");
}
```

#### c) Opciones adicionales

```
navigator.geolocation.getCurrentPosition(
  successCallback,
  errorCallback,
  {
    enableHighAccuracy: true, // Mayor precisión
    timeout: 5000,            // Máximo tiempo de espera
    maximumAge: 0             // No usar datos antiguos
  }
);
```

#### d) Práctica:

Mostrar ubicación del usuario en un mapa usando Google Maps o Leaflet.js

### 3. Acceso a cámara y micrófono – `getUserMedia()`

### a) ¿Para qué sirve?

Permite acceder a cámaras y micrófonos del dispositivo.

### b) Ejemplo básico

```
const video = document.getElementById('video');

navigator.mediaDevices.getUserMedia({ video: true, audio: true })
  .then(stream => {
    video.srcObject = stream;
  })
  .catch(err => {
    console.error("No se pudo acceder a la cámara/micrófono", err);
  });
```

### c) Mostrar imagen capturada

```
const canvas = document.getElementById('canvas');
const context = canvas.getContext('2d');

document.getElementById('tomarFoto').addEventListener('click', () => {
  context.drawImage(video, 0, 0, canvas.width, canvas.height);
});
```

### d) Práctica:

Crear una mini aplicación que tome una foto y la muestre en pantalla

## 4. Sensores de dispositivo – `DeviceMotion` y `DeviceOrientation`

### a) `DeviceMotionEvent`

Detecta aceleración del dispositivo (movimiento)

```
window.addEventListener('devicemotion', event => {
  const acceleration = event.acceleration;
  console.log(`Aceleración X: ${acceleration.x}`);
  console.log(`Aceleración Y: ${acceleration.y}`);
  console.log(`Aceleración Z: ${acceleration.z}`);
});
```

### b) `DeviceOrientationEvent`

Detecta orientación del dispositivo (giroscopio)

```
window.addEventListener('deviceorientation', event => {
  console.log(`Alpha: ${event.alpha}`); // Rotación horizontal
  console.log(`Beta: ${event.beta}`);   // Inclinación frontal
  console.log(`Gamma: ${event.gamma}`); // Inclinación lateral
});
```

> ⚠ Requiere HTTPS o permisos explícitos en algunos dispositivos.

## 5. Detección de conectividad – `navigator.onLine`

### a) ¿Para qué sirve?

Detecta si el dispositivo tiene conexión a internet.

### b) Uso básico

```
console.log("Estás online?", navigator.onLine);

window.addEventListener('online', () => {
  alert("Conexión restablecida");
});

window.addEventListener('offline', () => {
  alert("Sin conexión");
});
```

### c) Práctica:

Mostrar un mensaje visual cuando el usuario esté sin conexión

## 6. Acceso a contactos – `ContactsManager` (experimental)

### a) ¿Para qué sirve?

Permite seleccionar contactos del dispositivo (solo en Chrome móvil).

### b) Ejemplo básico

```
if ('contacts' in navigator && 'ContactsManager' in window) {
  const props = { multiple: true };
  const contacts = await navigator.contacts.select(['name', 'tel'], props);
  console.log(contacts);
}
```

> ⚠ Requiere HTTPS, PWA instalada y permiso explícito del usuario.

## 7. Fullscreen API

### a) ¿Para qué sirve?

Permite mostrar contenido en modo pantalla completa.

### b) Ejemplo básico

```
const elem = document.getElementById("miVideo");

document.getElementById("pantallaCompleta").addEventListener("click", () => {
  if (elem.requestFullscreen) {
    elem.requestFullscreen();
  }
});
```

### c) Salir de pantalla completa

```
document.exitFullscreen();
```

## 8. Web Share API

### a) ¿Para qué sirve?

Permite compartir contenido directamente desde el navegador.

### b) Ejemplo básico

```
if (navigator.share) {
  navigator.share({
    title: 'Mi sitio',
    text: 'Mira esta página increíble',
    url: 'https://midominio.com'
  }).then(() => console.log('Compartido'))
    .catch(err => console.error('Error al compartir:', err));
}
```

> ⚠ Solo funciona en contextos seguros (HTTPS) y en dispositivos móviles principalmente.

### 9. Práctica guiada: App de registro de usuarios offline-first con acceso a cámara y geolocalización

**Objetivo:**

Crear una aplicación que permita:

- Registrar usuarios con nombre, foto y ubicación
- Funcione offline
- Guarde datos localmente y sincronice cuando haya conexión

**Pasos:**

1. Solicitar permiso de cámara y geolocalización
2. Tomar una foto y guardarla en IndexedDB
3. Obtener ubicación y guardarla también
4. Usar Background Sync para enviar datos cuando haya red

### 10. Buenas prácticas

- Siempre pedir permisos explícitamente y manejar rechazos
- Verificar soporte del navegador antes de usar cada API
- Probar en dispositivos móviles reales
- Usar HTTPS siempre que sea posible
- Priorizar accesibilidad y privacidad del usuario

## ✅ Resumen del módulo

| Concepto | Descripción | ||-| | Geolocation API | Obtener ubicación del usuario | | getUserMedia | Acceder a cámara y micrófono | | DeviceMotion / Orientation | Detectar movimiento y orientación | | ContactsManager | Seleccionar contactos (experimental) | | Fullscreen API | Mostrar contenido en pantalla completa | | Web Share API | Compartir contenido desde el navegador | | navigator.onLine | Detectar estado de conexión | | Práctica | Aplicación de registro con fotos y ubicación |

## Recursos recomendados

- MDN Web Docs – Device APIs
- Google Developers – Device Access
- Web.dev – Device Access
- Can I Use – Feature Detection

# MÓDULO 9: Proyecto Final – Construcción de una PWA completa

## ⏱ Duración sugerida: 6 horas (puede dividirse en varias sesiones)

## 🎯 Objetivo del módulo:

Consolidar todos los conocimientos adquiridos durante la capacitación mediante la **construcción de una Progressive Web App funcional y completamente operativa offline**, integrando todas las funcionalidades aprendidas:

- Registro y uso de Service Worker
- Caché con `CacheStorage` e integración con Workbox
- Sincronización en segundo plano
- Notificaciones push
- Estrategias offline-first
- Acceso a características nativas del dispositivo

Al finalizar este módulo, los participantes deberán tener un proyecto funcional que demuestre su capacidad para construir PWAs completas.

## Temario detallado:

### 1. Definición del proyecto final

### Aplicación propuesta:

**"Registro de visitas técnicas"**

Una aplicación web que permite a técnicos o vendedores:

- Registrar clientes visitados
- Tomar fotos del lugar
- Obtener ubicación GPS
- Guardar datos localmente si no hay conexión
- Enviar datos al backend cuando haya red
- Recibir notificaciones push programadas

## 2. Requisitos técnicos del proyecto

| Requisito | Descripción |
|---|---|
| HTTPS | Obligatorio para todas las funcionalidades |
| Manifest Web App | Para instalación desde navegador |
| Service Worker | Para caché, background sync y notificaciones |
| IndexedDB | Almacenamiento local de datos offline |
| Background Sync | Sincronizar datos cuando haya conexión |
| Push Notifications | Mostrar alertas desde backend |
| Geolocalización | Registrar coordenadas de visita |
| getUserMedia | Capturar foto del cliente/lugar |
| Web Share API (opcional) | Compartir reportes |

## 3. Estructura del proyecto

```
mi-pwa/
│
├── index.html
├── manifest.json
├── sw.js
├── app.js
├── style.css
├── images/
│   ├── icon-192x192.png
│   └── badge.png
├── workbox-config.js (opcional)
└── README.md
```

## 4. Paso a paso del desarrollo

### Paso 1: Configuración inicial

1. Crear archivo `manifest.json` básico con:
    - Nombre de la aplicación
    - Iconos
    - `start_url`
    - `display: standalone`
2. Registrar el Service Worker en `app.js`:

```
if ('serviceWorker' in navigator) {
  navigator.serviceWorker.register('/sw.js')
    .then(reg => console.log('SW registrado', reg))
    .catch(err => console.error('Error registrando SW:', err));
}
```

3. Añadir enlace al manifest en `<head>` de `index.html`:

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

### Paso 2: Implementar caché con Workbox

1. Usar CDN para importar Workbox en `sw.js`:

```
importScripts('https://storage.googleapis.com/workbox-cdn/releases/6.5.4/workbox-sw.js');
```

2. Configurar estrategias básicas:

```
const { registerRoute } = workbox.routing;
const { StaleWhileRevalidate } = workbox.strategies;
const { CacheableResponsePlugin } = workbox.cacheableResponse;
const { ExpirationPlugin } = workbox.expiration;

// Cachear recursos estáticos
registerRoute(
  ({ request }) => request.destination === 'script' || request.destination === 'style',
  new StaleWhileRevalidate({
    cacheName: 'recursos-estaticos'
  })
);

// Cachear imágenes
registerRoute(
  ({ request }) => request.destination === 'image',
  new StaleWhileRevalidate({
    cacheName: 'imagenes',
    plugins: [
      new ExpirationPlugin({ maxEntries: 60, maxAgeSeconds: 30 * 24 * 60 * 60 })
    ]
  })
);
```

### Paso 3: Acceso a cámara y geolocalización

1. Solicitar permiso de cámara:

```
document.getElementById('tomarFoto').addEventListener('click', async () => {
  const stream = await navigator.mediaDevices.getUserMedia({ video: true });
  const video = document.getElementById('video');
  video.srcObject = stream;
});
```

2. Tomar foto y mostrarla:

```
document.getElementById('capturar').addEventListener('click', () => {
  const canvas = document.getElementById('canvas');
  const video = document.getElementById('video');
  const context = canvas.getContext('2d');
  context.drawImage(video, 0, 0, canvas.width, canvas.height);
});
```

3. Obtener ubicación del usuario:

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

### Paso 4: Guardar datos localmente con IndexedDB

1. Crear base de datos:

```
const indexedDB =
  window.indexedDB ||
  window.mozIndexedDB ||
  window.webkitIndexedDB ||
  window.msIndexedDB;

const request = indexedDB.open("VisitasTecnicas", 1);

request.onupgradeneeded = function(event) {
  const db = event.target.result;
  if (!db.objectStoreNames.contains('visitas')) {
    db.createObjectStore('visitas', { keyPath: 'id' });
  }
};
```

2. Guardar visita localmente:

```
function guardarVisita(visita) {
  const db = ...;
  const tx = db.transaction('visitas', 'readwrite');
  const store = tx.objectStore('visitas');
  store.put(visita);
}
```

### Paso 5: Sincronización en segundo plano

1. Programar sincronización cuando no hay conexión:

```
if (navigator.onLine) {
  enviarVisitaAlServidor(visita);
} else {
  guardarVisitaLocal(visita);
  navigator.serviceWorker.ready.then(reg => {
    reg.sync.register('sync-visitas');
  });
}
```

2. Manejar evento `sync` en `sw.js`:

```
self.addEventListener('sync', event => {
  if (event.tag === 'sync-visitas') {
    event.waitUntil(sincronizarVisitas());
  }
});
```

### Paso 6: Notificaciones push

1. Registrar suscripción:

```
navigator.serviceWorker.ready.then(reg => {
  reg.pushManager.subscribe({
    userVisibleOnly: true,
    applicationServerKey: urlBase64ToUint8Array(VAPID_PUBLIC_KEY)
  }).then(subscription => {
    fetch('/api/subscribe', {
      method: 'POST',
      body: JSON.stringify(subscription),
      headers: { 'Content-Type': 'application/json' }
    });
  });
});
```

2. Mostrar notificación en `sw.js`:

```
self.addEventListener('push', event => {
  const data = event.data.json();
  self.registration.showNotification(data.title, {
    body: data.body,
    icon: '/icon-192x192.png',
    badge: '/badge.png'
  });
});
```

### 5. Práctica guiada: Desarrolla tu propia PWA

**Actividad final:**

Construye una aplicación web progresiva siguiendo estos pasos:

1. Define el propósito de tu aplicación (ej.: lista de tareas, diario personal, agenda de contactos, etc.)
2. Integra todas las funcionalidades aprendidas:
   - Service Worker
   - Caché con Workbox
   - IndexedDB
   - Background Sync
   - Geolocalización o cámara
   - Notificaciones push
3. Realiza auditoría con Lighthouse y asegúrate de pasar todas las métricas PWA
4. Sube tu proyecto a GitHub y comparte el link

### 6. Buenas prácticas finales

| Práctica | Descripción | ||-| | Auditorías con Lighthouse | Verifica calidad y cumplimiento PWA | | Testing offline | Usa DevTools para simular desconexión | | Seguridad | Siempre bajo HTTPS | | Documentación | Incluye instrucciones de uso y dependencias | | Mejora continua | Agrega nuevas funcionalidades según necesidad |

## Resumen del módulo

| Concepto | Descripción | ||-| | PWA Completa | Integración de todas las tecnologías aprendidas | | Workbox | Simplifica el manejo de Service Worker | | IndexedDB | Almacena datos estructurados offline | | Background Sync | Envía datos cuando hay conexión | | Notificaciones Push | Interactúa con usuarios incluso fuera de la app | | Características nativas | Cámara, geolocalización, sensores, etc. | | Práctica final | Proyecto funcional listo para producción |

## Recursos recomendados

- PWA Builder
- Lighthouse Audit
- Workbox Recipes
- Google Developers – PWA Checklist