Contenido

[1. Introducción a los Service Workers 2](#_Toc187830312)

[Requisitos 2](#_Toc187830313)

[2. Flujo básico de un Service Worker 2](#_Toc187830314)

[3. Ejemplo básico de Service Worker 2](#_Toc187830315)

[3.1. Paso 1: Estructura del proyecto 2](#_Toc187830316)

[3.2. Paso 2: Código del archivo index.html 2](#_Toc187830317)

[3.3. Paso 3: Código del archivo app.js 3](#_Toc187830318)

[3.3.1. Verificación de compatibilidad con Service Workers 3](#_Toc187830319)

[3.3.2. Registro del Service Worker 3](#_Toc187830320)

[3.3.3. Promesas para manejar el registro 4](#_Toc187830321)

[3.3.3.1. Éxito 4](#_Toc187830322)

[3.3.3.2. Error 4](#_Toc187830323)

[¿Qué es un Service Worker? 4](#_Toc187830324)

[Ventajas del uso de Service Workers 4](#_Toc187830325)

[¿Qué es una Promesa? 4](#_Toc187830326)

[Ejemplo de Uso de Promesas en Service Workers 5](#_Toc187830327)

[Resumen 6](#_Toc187830328)

[3.4. Paso 4: Código del archivo service-worker.js 6](#_Toc187830329)

[1. Definición del nombre del caché 6](#_Toc187830330)

[2. URLs para almacenar en el caché 7](#_Toc187830331)

[1. Escucha del evento install 8](#_Toc187830332)

[2. Registro de un mensaje en la consola 8](#_Toc187830333)

[3. Bloqueo del evento hasta completar el precaching 8](#_Toc187830334)

[4. Comportamiento del código 9](#_Toc187830335)

[Propósito del código 9](#_Toc187830336)

[Flujo típico del ciclo de vida del Service Worker 9](#_Toc187830337)

[4. Prueba de tu Service Worker 10](#_Toc187830338)

[5. Extensiones útiles 10](#_Toc187830339)

[Agregar mensajes en la consola 10](#_Toc187830340)

[6. Funcionalidades avanzadas 11](#_Toc187830341)

[1. Estrategias de caché 11](#_Toc187830342)

[2. Actualización automática de caché 11](#_Toc187830343)

[7. Conclusión 11](#_Toc187830344)

[8. Ejemplo de un service worker completo 12](#_Toc187830345)

[Service Worker: Ejemplo Completo 12](#_Toc187830346)

[¿Qué hace este Service Worker? 13](#_Toc187830347)

[Archivos adicionales recomendados: 14](#_Toc187830348)

[¿Cómo registrar el Service Worker? 14](#_Toc187830349)

Service Workers

## **Introducción a los Service Workers**

Un Service Worker es un script en JavaScript que se ejecuta en segundo plano, separado de la página web. Sirve para:

* Interceptar y manejar solicitudes de red.
* Almacenar recursos en caché.
* Habilitar funcionalidades offline.

### **Requisitos**

* Los Service Workers funcionan únicamente en servidores HTTPS (o localhost en desarrollo).
* Debes registrar el Service Worker en el navegador.

## **Flujo básico de un Service Worker**

1. Registrar el Service Worker en tu página web.
2. Instalar el Service Worker (almacena recursos en la caché).
3. Activar el Service Worker (limpia la caché antigua).
4. Interceptar solicitudes y servir recursos desde la caché o la red.

## **Ejemplo básico de Service Worker**

## **Paso 1: Estructura del proyecto**

Crea los siguientes archivos:

/mi-proyecto

|- index.html

|- styles.css

|- app.js

|- service-worker.js

## **Paso 2: Código del archivo index.html**

Este archivo registra el Service Worker y contiene tu aplicación básica.

<!DOCTYPE html>

<html lang="es">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Service Worker Demo</title>

<link rel="stylesheet" href="styles.css">

</head>

<body>

<h1>Hola, mundo</h1>

<p>Esta es una demo de Service Workers.</p>

<script src="app.js"></script>

</body>

</html>

## **Paso 3: Código del archivo app.js**

Registra el Service Worker en el navegador:

if ('serviceWorker' in navigator) {

navigator.serviceWorker.register('/service-worker.js')

.then(() => console.log('Service Worker registrado exitosamente.'))

.catch(error => console.error('Error al registrar el Service Worker:', error));

}

El código anterior implementa un mecanismo para registrar un Service Worker en un navegador compatible.

## **Verificación de compatibilidad con Service Workers**

if ('serviceWorker' in navigator) {

* El objeto global **navigator** representa el estado y la identidad del agente de usuario (el navegador).
* **serviceWorker** es una propiedad del objeto navigator que verifica si el navegador soporta **Service Workers**.
* Este if asegura que el código dentro solo se ejecute si el navegador tiene soporte para Service Workers.

## **Registro del Service Worker**

navigator.serviceWorker.register('service-worker.js')

* navigator.serviceWorker.register() es el método que registra un Service Worker.
* El argumento '**service-worker.js**' es la ruta al archivo del script del Service Worker que quieres registrar.
* El archivo service-worker.js contiene el código que define el comportamiento del Service Worker (caché, interceptación de solicitudes, etc.).

## **Promesas para manejar el registro**

## **Éxito**

.then(() => console.log('Service Worker registrado exitosamente.'));

* Si el registro del Service Worker tiene éxito, se ejecuta el método .then().
* Este método imprime el mensaje: 'Service Worker registrado exitosamente.'.

## **Error**

.catch(error => console.error('Error al registrar el Service Worker:', error));

* Si el registro falla, el método .catch() se ejecuta.
* Este método imprime un mensaje de error ('Error al registrar el Service Worker:') junto con la información del error (error).

### **¿Qué es un Service Worker?**

Un Service Worker es un script que actúa como un proxy entre tu aplicación web y la red. Sus principales características son:

1. Caché de recursos: Permite almacenar recursos (HTML, CSS, JS, imágenes, etc.) para habilitar funcionalidades offline.
2. Intercepción de solicitudes: Puede interceptar y manipular las solicitudes de red.
3. Notificaciones push: Permite manejar notificaciones enviadas desde el servidor.
4. Background sync: Sincroniza datos en segundo plano cuando hay conexión.

### **Ventajas del uso de Service Workers**

* Mejora la experiencia del usuario al permitir funcionalidades offline.
* Mejora el rendimiento al cargar recursos desde el caché en lugar de la red.
* Facilita el envío de notificaciones push para aumentar el engagement.

En el contexto de un Service Worker, una promesa es un mecanismo fundamental que se utiliza para manejar operaciones asíncronas. Los Service Workers operan como intermediarios entre la aplicación web y el servidor (o caché) y realizan tareas como interceptar solicitudes de red, almacenar recursos en caché y responder a solicitudes desde la caché.

### ¿Qué es una Promesa?

Una promesa es un objeto en JavaScript que representa un valor que puede estar disponible ahora, en el futuro o nunca. En el caso de los **Service Workers**, las promesas se utilizan para manejar tareas que no se completan instantáneamente, como:

1. Cargar recursos desde la red.
2. Guardar archivos en el caché.
3. Leer datos de la caché.

### Ejemplo de Uso de Promesas en Service Workers

Las promesas son fundamentales en los eventos más comunes de un Service Worker, como install, activate y fetch. Por ejemplo:

#### 1. **Evento** install

Este evento se dispara cuando se instala un nuevo Service Worker. Aquí, las promesas se usan para **precargar** recursos en el caché.

self.addEventListener('install', (event) => {

event.waitUntil(

caches.open('my-cache').then((cache) => {

return cache.addAll([

'/index.html',

'/styles.css',

'/script.js',

'/logo.png'

]);

})

);

});

* **event.waitUntil()** espera a que la promesa dentro de ella se resuelva antes de que el Service Worker pase al siguiente estado.

#### 2. **Evento** fetch

Este evento se dispara cada vez que la aplicación realiza una solicitud de red. Las promesas permiten interceptar y responder con recursos de la caché o desde la red.

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

event.respondWith(

caches.match(event.request).then((response) => {

return response || fetch(event.request);

})

);

});

* caches.match(event.request) busca el recurso en la caché.
* Si no se encuentra en la caché, fetch(event.request) hace la solicitud a la red.

#### 3. **Evento** activate

Este evento se utiliza para realizar tareas como limpiar versiones antiguas del caché. Aquí también se usan promesas.

self.addEventListener('activate', (event) => {

const cacheWhitelist = ['my-cache-v2'];

event.waitUntil(

caches.keys().then((cacheNames) => {

return Promise.all(

cacheNames.map((cacheName) => {

if (!cacheWhitelist.includes(cacheName)) {

return caches.delete(cacheName);

}

})

);

})

);

});

* caches.keys() devuelve una promesa con los nombres de los cachés existentes.
* Promise.all() espera a que todas las promesas de caches.delete(cacheName) se resuelvan antes de continuar.

### Resumen

En un Service Worker, las promesas:

* Aseguran que las operaciones asíncronas (como trabajar con la caché o la red) se manejen correctamente.
* Garantizan que el Service Worker espere a que las tareas críticas se completen antes de avanzar a un nuevo estado.
* Ofrecen una forma limpia y legible de encadenar y manejar tareas asíncronas.

## **Paso 4: Código del archivo service-worker.js**

Este es el código principal del Service Worker:

#### **1. Nombre de la caché y recursos a almacenar**

Define un nombre para la caché y una lista de archivos que deseas almacenar:

const CACHE\_NAME = 'mi-cache-v1';

const URLS\_TO\_CACHE = [

'/',

'/index.html',

'/styles.css',

'/app.js',

];

Este código define dos constantes relacionadas con la configuración de un Service Worker para implementar un sistema de caché en una aplicación web. Aquí está el desglose:

### **1. Definición del nombre del caché**

const CACHE\_NAME = 'mi-cache-v1';

* CACHE\_NAME es una constante que define el nombre único del caché.
* Este nombre se usa para identificar y gestionar un grupo específico de recursos en el almacenamiento del navegador.
* El sufijo -v1 (versión 1) indica la versión del caché. Es útil para implementar estrategias de actualización en el futuro (como mi-cache-v2).

### **2. URLs para almacenar en el caché**

const URLS\_TO\_CACHE = [

'/',

'/index.html',

'/styles.css',

'/app.js',

];

* URLS\_TO\_CACHE es una constante que define un array de rutas (URLs) que deben ser almacenadas en el caché.
* Estas URLs representan los recursos estáticos de la aplicación web que se necesitan para que funcione, incluso cuando no hay conexión a Internet.

#### Desglose de las URLs:

* '/': Representa la raíz de la aplicación (generalmente el archivo inicial index.html).
* '/index.html': Es el archivo HTML principal de la aplicación.
* '/styles.css': Es el archivo de estilos CSS que define el diseño de la aplicación.
* '/app.js': Es el archivo JavaScript principal que contiene la lógica de la aplicación.

#### **2. Evento** install

Este evento se activa cuando se instala el Service Worker. Aquí es donde se almacenan los recursos en caché:

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

console.log('Service Worker: Instalando...');

event.waitUntil(

caches.open(CACHE\_NAME).then(cache => {

console.log('Service Worker: Caché abierta');

return cache.addAll(URLS\_TO\_CACHE);

})

);

});

Este código registra un listener para el evento install del Service Worker. Durante el proceso de instalación, realiza varias tareas relacionadas con el almacenamiento en caché de recursos estáticos. Aquí tienes una explicación detallada de cada parte:

### **1. Escucha del evento** install

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

* self: Hace referencia al contexto del Service Worker (similar a this en otros contextos).
* addEventListener('install', ...): Registra un manejador para el evento install. Este evento se dispara cuando el navegador instala el Service Worker por primera vez o después de una actualización.

### **2. Registro de un mensaje en la consola**

console.log('Service Worker: Instalando...');

* Imprime un mensaje en la consola para indicar que el proceso de instalación del Service Worker ha comenzado.

### **3. Bloqueo del evento hasta completar el precaching**

event.waitUntil(

caches.open(CACHE\_NAME).then(cache => {

console.log('Service Worker: Caché abierta');

return cache.addAll(URLS\_TO\_CACHE);

})

);

#### event.waitUntil()

* Retrasa la finalización del evento install hasta que se complete la promesa pasada como argumento.
* Si la promesa falla, el Service Worker no se instalará.

#### caches.open(CACHE\_NAME)

* Abre o crea un caché con el nombre especificado en CACHE\_NAME.
* Retorna una promesa que resuelve con el objeto del caché.

#### .then(cache => { ... })

* Una vez que el caché está abierto, ejecuta una función donde:
  + Imprime 'Service Worker: Caché abierta' en la consola.
  + Llama a cache.addAll(URLS\_TO\_CACHE) para almacenar en el caché las URLs definidas en URLS\_TO\_CACHE.

#### cache.addAll(URLS\_TO\_CACHE)

* Recibe un array de URLs (por ejemplo, /index.html, /styles.css) y las almacena en el caché.
* Si alguno de los recursos falla al cargarse, se lanza un error y el proceso de instalación se detiene.

### **4. Comportamiento del código**

1. Evento install activado: Cuando se registra o actualiza un Service Worker, el navegador dispara el evento install.
2. Apertura del caché: Se crea o abre un caché con el nombre definido en CACHE\_NAME.
3. Precaching de recursos: Las URLs listadas en URLS\_TO\_CACHE se descargan y almacenan en el caché.
4. Espera hasta completar: La instalación del Service Worker no finaliza hasta que el proceso de precaching termine satisfactoriamente.

### **Propósito del código**

El objetivo es precargar recursos esenciales para la aplicación durante la instalación del Service Worker. Esto permite:

* Hacer que la aplicación funcione más rápido al cargar recursos desde el caché.
* Proporcionar una experiencia offline para los usuarios al tener recursos clave almacenados localmente.

### **Flujo típico del ciclo de vida del Service Worker**

1. install: Se cachean los recursos necesarios.
2. activate: Limpia cachés antiguos si es necesario.
3. fetch: Intercepta las solicitudes de red y responde con recursos desde el caché o la red según la estrategia.

#### **3. Evento** fetch

Intercepta solicitudes de red y responde con recursos almacenados en la caché o desde la red:

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

console.log('Service Worker: Fetch detectado para', event.request.url);

event.respondWith(

caches.match(event.request).then(response => {

// Si el recurso está en caché, lo devuelve

return response || fetch(event.request);

})

);

});

#### **4. Evento** activate

Elimina versiones antiguas de la caché cuando se activa un nuevo Service Worker:

self.addEventListener('activate', event => {

console.log('Service Worker: Activando...');

const cacheWhitelist = [CACHE\_NAME];

event.waitUntil(

caches.keys().then(cacheNames => {

return Promise.all(

cacheNames.map(cacheName => {

if (!cacheWhitelist.includes(cacheName)) {

console.log('Service Worker: Eliminando caché antigua', cacheName);

return caches.delete(cacheName);

}

})

);

})

);

});

## **Prueba de tu Service Worker**

1. Levanta un servidor local. Si usas VSCode, puedes instalar la extensión Live Server. También puedes usar http-server:

npx http-server

Abre http://localhost:8080.

1. Verifica el registro del Service Worker:
   * Abre las DevTools de tu navegador (F12).
   * Ve a la pestaña Application > Service Workers.
   * Deberías ver tu Service Worker activo.
2. Desconecta el Internet:
   * Refresca la página. Los recursos deberían cargarse desde la caché.

## **Extensiones útiles**

### **Agregar mensajes en la consola**

Para depurar más fácilmente:

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

console.log('Service Worker: Fetch detectado para', event.request.url);

event.respondWith(

caches.match(event.request).then(response => {

console.log(response ? 'Cargando desde caché:' : 'Cargando desde la red:', event.request.url);

return response || fetch(event.request);

})

);

});

## **Funcionalidades avanzadas**

### **1. Estrategias de caché**

Puedes implementar diferentes estrategias de caché según tus necesidades:

* Cache First (Usar primero la caché):

event.respondWith(

caches.match(event.request).then(response => {

return response || fetch(event.request);

})

);

* Network First (Intentar primero la red):

event.respondWith(

fetch(event.request).catch(() => caches.match(event.request))

);

### **2. Actualización automática de caché**

Para actualizar archivos en caché:

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

event.waitUntil(

caches.open(CACHE\_NAME).then(cache => {

return cache.addAll(URLS\_TO\_CACHE).then(() => self.skipWaiting());

})

);

});

## **Conclusión**

Este tutorial te enseña cómo implementar un Service Worker básico para almacenar en caché los recursos de tu aplicación. Los Service Workers son una herramienta poderosa para crear aplicaciones web rápidas, confiables y funcionales incluso sin conexión a Internet.

## **Ejemplo de un service worker completo**

**Service Worker** que realiza tareas comunes como instalación, activación y manejo de solicitudes de red, incluyendo almacenamiento en caché:

### ****Service Worker: Ejemplo Completo****

// Nombre del caché

const CACHE\_NAME = 'my-cache-v1';

// Recursos para precargar

const PRECACHE\_RESOURCES = [

'/',

'/index.html',

'/styles.css',

'/script.js',

'/logo.png'

];

// Evento de instalación

self.addEventListener('install', (event) => {

console.log('[Service Worker] Instalando...');

event.waitUntil(

caches.open(CACHE\_NAME).then((cache) => {

console.log('[Service Worker] Precargando recursos en la caché...');

return cache.addAll(PRECACHE\_RESOURCES);

})

);

});

// Evento de activación

self.addEventListener('activate', (event) => {

console.log('[Service Worker] Activando y limpiando cachés antiguas...');

event.waitUntil(

caches.keys().then((cacheNames) => {

return Promise.all(

cacheNames.map((cacheName) => {

if (cacheName !== CACHE\_NAME) {

console.log(`[Service Worker] Eliminando caché antigua: ${cacheName}`);

return caches.delete(cacheName);

}

})

);

})

);

});

// Evento de fetch (intercepta solicitudes de red)

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

console.log(`[Service Worker] Interceptando solicitud para: ${event.request.url}`);

event.respondWith(

caches.match(event.request).then((response) => {

if (response) {

console.log(`[Service Worker] Sirviendo desde la caché: ${event.request.url}`);

return response;

}

console.log(`[Service Worker] Realizando solicitud a la red: ${event.request.url}`);

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

// Opcional: Guardar en caché la nueva respuesta

return caches.open(CACHE\_NAME).then((cache) => {

cache.put(event.request, networkResponse.clone());

return networkResponse;

});

});

}).catch((error) => {

console.error(`[Service Worker] Error al manejar la solicitud: ${error}`);

// Opcional: Mostrar una página de fallback

if (event.request.mode === 'navigate') {

return caches.match('/offline.html');

}

})

);

});

### ¿Qué hace este Service Worker?

1. **Precarga recursos al instalarse**:
   * Guarda recursos esenciales en la caché para que estén disponibles sin conexión.
2. **Limpia cachés antiguas al activarse**:
   * Borra versiones anteriores del caché que ya no son necesarias.
3. **Intercepta solicitudes de red**:
   * Busca primero en la caché si el recurso solicitado ya está guardado.
   * Si no está en la caché, lo obtiene de la red, lo guarda en la caché y luego lo devuelve.
4. **Manejo de errores**:
   * Si ocurre un error (como estar sin conexión), intenta proporcionar una página de fallback (opcional).

### Archivos adicionales recomendados:

* **/offline.html**: Página HTML básica para mostrar cuando no hay conexión a Internet.

### ¿Cómo registrar el Service Worker?

En el archivo principal de tu aplicación (por ejemplo, script.js):

if ('serviceWorker' in navigator) {

navigator.serviceWorker.register('/service-worker.js')

.then((registration) => {

console.log('[App] Service Worker registrado con éxito:', registration);

})

.catch((error) => {

console.error('[App] Error al registrar el Service Worker:', error);

});

}