ChatGPT

Service Workers in JavaScript: Theory, Operation, and Push Notification Examples

A **service worker** is a JavaScript script that runs in the browser's background (separate from the web page) and acts like a programmable network proxy between your web app and the network [1] [2]. It sits between the client and server (a "third layer" in your app's architecture [2]) and can intercept network requests, cache resources, and even receive push messages when your page is not open. Service workers enable key Progressive Web App (PWA) features: offline access, background data sync, and push notifications [1] [3]. Unlike normal scripts, a service worker has **no access to the DOM** and runs on its own thread [4]. It is **event-driven**, responding to `install`, `activate`, `fetch`, `push`, and other lifecycle events, and uses Promises for async actions [4] [5]. Crucially, service workers only run on **secure origins** (HTTPS or `localhost`) for security [6].

Why Service Workers Exist

Historically, web apps suffered poor reliability when the network was slow or offline. Traditional caching (like AppCache) was rigid and broke easily. Service workers were introduced to give web developers full control over network requests and caching, enabling truly offline-capable apps [3] [7]. With a service worker, you can implement an "offline-first" strategy: serve cached assets instantly and update them in the background when online [3]. This improves performance (users get fast, cached responses) and resilience (the app still works offline). For example, a service worker can intercept a page navigation and serve a cached HTML shell for a single-page app if the network is unavailable [3] [8].

Beyond offline support, service workers enable modern web features: they can show **push notifications** to re-engage users, and they can perform **background sync** of data when connectivity returns [1] [9]. They essentially make web apps behave more like native apps. (Indeed, Google's PWA guidelines call service workers "fundamental" for enabling app-like capabilities [1] [10].) Research shows 53% of users abandon a site that takes more than 3 seconds to load, so by caching assets and speeding up responses, service workers can significantly improve user retention [11] [8]. In summary, **service workers exist to give developers granular control** over caching and network behavior (replacing older unreliable methods) and to unlock background features (push, sync) for the web [3] [1].

How Service Workers Work (Lifecycle and Architecture)

When a web app first registers a service worker, the browser downloads and installs it. The basic lifecycle is: **download → install → activate** [5]. During the **install** event, the service worker can pre-cache static assets (HTML, JS, CSS, images) so that the app shell is available offline [5]. Once installed, the service worker waits until all existing pages using the old worker are closed, then it **activates** and takes control. After activation, the service worker intercepts network requests from pages under its scope, enabling custom caching strategies or offline responses [5] [1].

Service workers run in a special worker context: they **cannot access the DOM** and run on a separate thread [4]. They rely on asynchronous APIs (no blocking XHR or localStorage inside) [4]. Instead, they use the Cache API, IndexedDB, Fetch API, Push API, etc., all promise-based. For example, a service worker's `fetch` event handler might do:

```
// In sw.js (service worker)
self.addEventListener('fetch', event => {
  event.respondWith(
    caches.match(event.request)
      .then(cached => {
        return cached || fetch(event.request);
      })
  );
});
```

This code tries to serve a request from cache first, falling back to network if missing [12].

In practice, you register a service worker in your page's JavaScript like this (in a secure context):

```
// In main.js (on your page)
if ('serviceWorker' in navigator) {
  navigator.serviceWorker.register('/sw.js')
    .then(reg => console.log('Service Worker registered', reg.scope))
    .catch(err => console.error('SW registration failed', err));
}
```

This tells the browser to install your `/sw.js` script as a service worker for your site [5]. The `.register()` returns a promise that resolves to a `ServiceWorkerRegistration` object, through which you can later subscribe to push or query the service worker's status.

Capabilities and Use Cases

Because a service worker sits between your app and the network, it can **proxy requests** and serve cached or computed responses. Common use cases include:

- **Offline Caching:** By caching key assets during `install` and intercepting `fetch`, you can make a web app work offline. For example, a news reader PWA might cache news articles viewed while online, and then display them later without connectivity [3] [12]. This yields a much faster perceived load (from cache) and reliable experience.

- **Content Caching and Performance:** Even with internet, service workers can implement advanced caching strategies (cache-first, network-first, stale-while-revalidate, etc.) for faster loads [13] [14]. For example, the "stale-while-revalidate" strategy returns a cached response immediately, then fetches a fresh copy to update the cache [15]. This ensures quick responses while keeping data up to date.

- **Background Sync:** Service workers can use the Background Sync API to defer actions until the network is available. For example, if a user posts a comment while offline, the service worker can queue that request and send it when connectivity returns [9]. (Support is currently limited to Chromium browsers, but it's a powerful way to make web apps robust offline.)

- **Push Notifications:** One of the most engaging features, push notifications let the server send timely messages to users even when the app isn't open. This uses the **Push API** in concert with service workers. Because service workers run in the background, they can receive a "push" event from a remote push service and display a notification via the Notification API [1] [16]. We will dive into this in detail below.

- **Other:** Service workers can also do things like background periodic sync or geofencing in the future, and can intercept form submissions or navigation to show custom offline pages. They effectively make the web "always on" within limits.

In short, service workers function like a programmable network proxy and event handler for web apps [17], enabling faster loads, offline use, and background features (notifications, sync) that were once exclusive to native apps.

Push Notifications via Service Workers

Push notifications are **server-initiated alerts** that appear on the user's device to notify about new events (messages, updates, reminders). On the web, push notifications work by registering a service worker that can receive *push events* even when the web page is closed. The flow is:

1. **Request permission:** The app asks the user's permission to show notifications using `Notification.requestPermission()`. The user must grant this [18]. It's best to do this in response to a user gesture (e.g. clicking "Enable notifications") after explaining why you need it.

2. **Register service worker:** The page registers a service worker (as shown above) and waits for it to be active.

3. **Subscribe to Push:** Using the Push API, the service worker creates a push subscription. In practice, from your page's JavaScript, you do:

```
navigator.serviceWorker.ready.then(reg => {
  const options = {
    userVisibleOnly: true,
    applicationServerKey: '<YOUR_PUBLIC_VAPID_KEY>'
  };
  return reg.pushManager.subscribe(options);
}).then(subscription => {
  console.log('Push subscription:', subscription);
  // Send subscription to your server for later use
});
```

Here, `applicationServerKey` is your app's public VAPID key (a credential pair that authenticates your server to the push service) [19]. The `subscribe()` returns a `PushSubscription` object containing an endpoint URL and cryptographic keys [20].

4. **Send subscription to server:** Your app sends this subscription info (via fetch/Ajax) to your backend server. The server will use it to send push messages to that user via a push service (like Firebase Cloud Messaging or the browser's built-in push service).

5. **Server sends push:** When there is new content or a message for the user, the server uses the push service (e.g. using a library like *web-push* in Node.js) to send a notification. Under the hood, the push service sends a message to the endpoint URL from the subscription. This triggers a `push` **event in the service worker**.

6. **Service worker handles push:** In the service worker file (`sw.js`), you listen for the `push` event and show a notification. For example:

```
// sw.js
self.addEventListener('push', event => {
  const data = event.data?.json() || {};
  const title = data.title || 'Hello!';
  const options = {
    body: data.body || 'You have a new message.',
    icon: data.icon || '/icon.png'
  };
  event.waitUntil(self.registration.showNotification(title, options));
});
```

   This code uses `event.data.json()` to get the payload sent from the server, then calls `showNotification()` to display it. The service worker can also handle `'notificationclick'` to respond when the user taps the notification (e.g. open a window).

This process is summarized by Google's web.dev guide: a client registers a service worker and calls `pushManager.subscribe()`, creating a unique endpoint URL for that user [21]. When the server posts to that endpoint, the service worker's `push` event fires, and it displays the notification [22]. Importantly, because service workers run in the background, **notifications can appear even if the web app is not open** [10]. This lets web apps re-engage users just like native apps do.

Note: Not all browsers support web push. As of 2025, Chrome, Firefox, and Edge support it, but Safari (especially on iOS) still lacks full Push API support [23]. Always check current browser compatibility before relying on push.

Example: Plain JavaScript App with Push Notifications

Below is a minimal example of a web page (`index.html`) with an accompanying `main.js` and `sw.js` to enable push notifications. This is greatly simplified and omits server-side details (assume you have a backend to send push messages with the subscription info):

```
<!-- index.html -->
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Push Notification Demo</title>
</head>
<body>
  <h1>Push Notification Demo</h1>
  <button id="enablePush">Enable Push Notifications</button>
  <script src="main.js"></script>
</body>
</html>
```

```
// main.js
const enableBtn = document.getElementById('enablePush');
enableBtn.addEventListener('click', () => {
  if ('serviceWorker' in navigator && 'PushManager' in window) {
    // Register the service worker
    navigator.serviceWorker.register('sw.js')
      .then(registration => {
        console.log('Service Worker registered:', registration);

        // Request notification permission
        return Notification.requestPermission()
          .then(permission => {
            if (permission !== 'granted') {
              throw new Error('Permission not granted for Notification');
            }
            // Subscribe to push
            const subscribeOptions = {
              userVisibleOnly: true,
              applicationServerKey: '<YOUR_PUBLIC_VAPID_KEY>'
            };
            return registration.pushManager.subscribe(subscribeOptions);
          });
      })
      .then(pushSubscription => {
        console.log('Received PushSubscription:', pushSubscription);
```

```
        // TODO: Send pushSubscription to application server
      })
      .catch(error => {
        console.error('Error during service worker registration or push
setup:', error);
      });
  } else {
    console.warn('Push messaging is not supported in this browser');
  }
});
```

```
// sw.js (Service Worker)
self.addEventListener('push', event => {
  console.log('Push event received:', event);
  const data = event.data ? event.data.json() : {};
  const title = data.title || 'Default Title';
  const options = {
    body: data.body || 'Default body text',
    icon: data.icon || '/icon.png'
  };
  event.waitUntil(
    self.registration.showNotification(title, options)
  );
});

// Optional: handle click on notification
self.addEventListener('notificationclick', event => {
  event.notification.close();
  event.waitUntil(
    clients.openWindow('https://example.com')  // Open a page
  );
});
```

Explanation: In this example, clicking the “Enable Push Notifications” button starts the process. We register `sw.js` as our service worker (so it can run in background) [24]. Once registered, we ask the user for notification permission (`Notification.requestPermission()`). If granted, we call `registration.pushManager.subscribe()` with `{userVisibleOnly: true}` and our public VAPID key [20]. This returns a `PushSubscription` object (logged in console) that includes an endpoint URL and keys. In a real app, you would send that subscription object to your server (e.g. via `fetch`), so the server can later send push messages.

The `sw.js` file listens for `push` events. When a push message arrives (triggered by the server), the service worker calls `showNotification()` to display it to the user [22]. We use `event.data.json()` to get the notification payload (which the server includes when sending), then show a notification with a title and body. The code mimics the example flow given on web.dev [22].

This minimal app demonstrates the key parts: **register SW + subscribe to push in the page script**, and **handle `push` in the SW to show notifications**. In practice, you would also implement the server side (e.g. using the web-push library) to send notifications to the subscription endpoint [25].

Example: React App with Push Notifications

The process in a React application is conceptually the same as above. You register a service worker and call the Push API. In a React app (e.g. created with Create React App), you might do this in a component or custom hook. For example:

```
// Example React component (PushButton.js)
import { useEffect } from 'react';

function PushButton() {
  useEffect(() => {
    // Register service worker on component mount
    if ('serviceWorker' in navigator && 'PushManager' in window) {
      navigator.serviceWorker.register('/sw.js')
        .then(registration => {
          console.log('SW registered:', registration);
        })
        .catch(err => {
          console.error('SW registration failed:', err);
        });
    }
  }, []);

  const subscribeUser = () => {
    Notification.requestPermission().then(permission => {
      if (permission !== 'granted') {
        alert('Permission not granted for notifications');
        return;
      }
      navigator.serviceWorker.ready.then(registration => {
        registration.pushManager.subscribe({
          userVisibleOnly: true,
          applicationServerKey: '<YOUR_PUBLIC_VAPID_KEY>'
        }).then(subscription => {
          console.log('Push subscribed:', subscription);
          // TODO: send subscription to server
        });
      });
    });
  };

  return (
```

```
      <button onClick={subscribeUser}>
        Enable Push Notifications
      </button>
    );
}

export default PushButton;
```

In this React example, we register the service worker when the component mounts (you could also do it in your main index.js). When the user clicks the button, we request notification permission. If granted, we wait for `navigator.serviceWorker.ready` (ensuring the SW is active) and then call `pushManager.subscribe()` with the same options [20]. The returned subscription is logged (you would send it to your server).

The `/sw.js` file (placed in the public folder so it's served at the root) is identical to the plain JS case: it listens for `push` and calls `showNotification()` [22].

If you use Create React App's PWA template, it provides a basic service worker setup for caching. You can customize it by adding push logic as shown above. (For example, change `serviceWorker.unregister()` to `serviceWorker.register()` in `index.js` to opt into SW, then add your `push` handler [26].) Libraries like Firebase Cloud Messaging (FCM) are also popular for web push in React; they handle subscription and messaging for you. But the core idea is always: **register a SW, subscribe to push, and handle `push` events in the SW** [21] [22].

Summary

Service workers are a powerful, low-level browser technology that run in the background to give web apps enhanced capabilities. They exist to solve longstanding issues with web reliability and capability: enabling offline experiences, fine-grained caching for performance, and background features like push notifications and sync [1] [3]. In essence, a service worker is a programmable proxy server at the client side, controlled by your JavaScript code, that can intercept network requests and respond however you program it [17].

In practice, you use service workers to cache assets (so your app can load instantly and even work offline), and to show notifications pushed from your server. For push notifications specifically, the workflow is: register the SW, get permission, create a push subscription (via `PushManager.subscribe()` [20]), send it to your server, and then let the SW show notifications when a message arrives [22]. This allows your web app to re-engage users with timely alerts even when the page isn't open [10] [16].

Below is a high-level list of steps for adding push to a web app:

- **Register the service worker** in your page's JavaScript (if supported) using `navigator.serviceWorker.register('sw.js')` [24].
- **Request notification permission** with `Notification.requestPermission()`. Proceed only if granted.

- **Subscribe to push**: `navigator.serviceWorker.ready.then(reg => reg.pushManager.subscribe({userVisibleOnly: true, applicationServerKey: YOUR_KEY}))` [20].
- **Send subscription to server**, so you can send push messages later.
- **On the server**, use a push library to send messages to the subscription endpoint URL.
- **Handle `push` in sw.js `**: use` self.addEventListener('push', ...) `to receive messages and call` showNotification()` [22].

Service workers have quickly become a foundational web technology. They **don't replace** web pages but augment them with background capabilities. By mastering service workers, you can make your web apps fast, reliable offline, and capable of engaging users through notifications. For a deeper dive, see the MDN Service Worker API guide and Google's service workers documentation [1] [10].

Sources: Authoritative documentation and guides on service workers and web push (MDN, Google's web.dev, Chromium team content, etc.) [1] [3] [16] [10] [22] [20] (code and concepts are also drawn from these sources).

---

[1] [4] [6] [7] Service Worker API - Web APIs | MDN
https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API

[2] [9] [11] [15] [16] [23] Nimble – Story: Here's Why Service Workers Have Superpowers
https://nimblestudio.com/story/heres-why-service-workers-have-superpowers

[3] [5] [17] Using Service Workers - Web APIs | MDN
https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API/Using_Service_Workers

[8] [12] [13] [14] What Are Service Workers and How They Help Improve Performance - KeyCDN
https://www.keycdn.com/blog/service-workers

[10] [18] [21] [22] [24] [25] Use push notifications to engage and re-engage users | Articles | web.dev
https://web.dev/articles/use-push-notifications-to-engage-users

[19] [20] PushManager: subscribe() method - Web APIs | MDN
https://developer.mozilla.org/en-US/docs/Web/API/PushManager/subscribe

[26] Making a Progressive Web App | Create React App
https://create-react-app.dev/docs/making-a-progressive-web-app/