Service Workers and caching strategies

🛠️ Service Workers and Caching Strategies in JavaScript

Service Workers are a powerful feature of modern browsers that enable **background tasks**, **offline experiences**, and **advanced caching strategies**. They're the backbone of **Progressive Web Apps (PWAs)**.

🔧 What Is a Service Worker?

A **Service Worker** is a **script that runs in the background**, separate from the main browser thread. It:

- **Intercepts network requests**
- Can **cache files** and **serve responses from cache**
- Enables **offline access**
- Can receive **push notifications** and perform **background sync**

It follows a **lifecycle**, is **event-driven**, and **runs only over HTTPS** (except on localhost).

📚 Service Worker Lifecycle

1. **Registration**

   Register the service worker in your app:

   ```
   if ('serviceWorker' in navigator) {
     navigator.serviceWorker.register('/sw.js')
       .then(reg => console.log('Registered:', reg.scope))
       .catch(err => console.error('Failed:', err));
   }
   ```

2. **Installation**

Triggered when the SW is first installed.

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

3. **Activation**

   Cleans up old caches or updates.

```
self.addEventListener('activate', event => {
  event.waitUntil(
    caches.keys().then(keys =>
      Promise.all(
        keys.filter(key => key !== 'v1').map(key => caches.delete(key))
      )
    )
  );
});
```

4. **Fetch**

   Intercepts network requests.

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

📦 Caching Strategies

1. Cache First (Offline First)

- Try cache → fallback to network

```
caches.match(req).then(res => res || fetch(req))
```

✅ Good for static assets

⚠️ May serve outdated content

---

2. Network First

- Try network → fallback to cache

```
fetch(req)
  .then(res => {
    caches.put(req, res.clone());
    return res;
  })
  .catch(() => caches.match(req));
```

✅ Great for dynamic content

⚠️ Slower due to network dependency

---

3. Stale-While-Revalidate

- Serve cache immediately, update in background

```
event.respondWith(
  caches.open('v1').then(cache =>
    cache.match(event.request).then(cachedRes => {
      const fetchPromise = fetch(event.request).then(networkRes => {
        cache.put(event.request, networkRes.clone());
        return networkRes;
      });
      return cachedRes || fetchPromise;
    })
  )
);
```

✅ Combines speed and freshness

⚠️ Slightly more complex

4. Network Only

- Always use network

  ✅ For critical live data (e.g., payments)

5. Cache Only

- Only use cache

  ✅ For offline-only apps

  ⚠️ Risk of missing content

🧪 Other Service Worker Features

- **Background Sync** – Defer actions until connectivity is back
- **Push Notifications** – Handle and display notifications
- **Clients API** – Communicate with pages from the service worker

🔒 Security Note

Service Workers require:

- **HTTPS** (except `localhost`)
- Explicit **user permission** for notifications or background sync

🧼 Cleaning Up Cache (Versioning)

Use versioning to manage cache:

```
const CACHE_NAME = 'my-cache-v2';

self.addEventListener('activate', event => {
  event.waitUntil(
    caches.keys().then(keys =>
      Promise.all(
        keys.filter(key => key !== CACHE_NAME)
            .map(key => caches.delete(key))
      )
    )
  );
});
```

✅ Summary

Feature	Description
Runs in background	Handles requests, caching, and more
Enables offline	Cache assets and data
Lifecycle phases	Install → Activate → Fetch
Key APIs	`caches`, `fetch`, `postMessage`, `clients`
Common strategies	Cache First, Network First, SWR, etc.
Required protocol	HTTPS only (except localhost)

Want a real-world PWA example or a demo app using Service Workers? I can help build one from scratch.