TOPIC:

Worker Service

BY:

Salas Diaz Guillermo

GROUP:

9noB

CLASS:

PWA

PROFESSOR:

Dr. Ray Brunett Parra Galavíz

Tijuana, Baja California, 14 de February del 2024

Introduction

Service workers are a powerful feature of modern web development, serving as proxy servers that sit between web applications, browsers, and networks. They enable developers to create web applications that can work offline, provide more responsive user experiences, and leverage advanced features like push notifications and background synchronization. In this essay, we will explore the characteristics, advantages, disadvantages, and applications of service workers in web development.

Índex

Table of ilustrations

1.What is a service worker?

A service worker is a sequence of commands executed by the browser in the background. This is a JavaScript file that continues to run even if the website is closed. These are the most notable characteristics of a service worker:

Service workers are a special type of web workers with the ability to intercept, modify, and respond to network requests using the Fetch API. Service workers can access the Cache API and client-side asynchronous data stores, such as IndexedDB, to store resources.

Service workers can speed up the PWA by caching resources locally and can also make your PWA more reliable by making it network independent.

The first time a user accesses your PWA, service worker is installed. The Service Worker then runs parallel to the application and can continue working even when the application is not running.

Service workers are responsible for intercepting, modifying, and responding to network requests. Alerts can be received when the application attempts to load a resource from the server or sends a request to obtain data from the server. When this happens, a service worker can decide to let the request go to the server or intercept it and return a response from the cache instead.

A service worker is a sequence of commands executed by the browser in the background. This is a JavaScript file that continues to run even if the website is closed.

At the technological center of the development of progressive web applications we have ServiceWorkers, a new JavaScript API that allows us to install a code script that acts as a proxy between the client, that is, our browser, and the server, where our application is located. page.

A ServiceWorker is capable of altering server responses, intercepting client requests, sending specific information to certain addresses, receiving push notifications, making updates even if the page is not open, and eventually other capabilities will be added to this section of the browser.

What makes it interesting is that through this technology and the ability to update and store files in the browser cache, we can offer a response to requests even without the internet.

To understand how a ServiceWorker works at a very high level, let's look at an example of how it would make our page appear without internet.

Traditionally, a browser requests information from the server through the network, that is, the Internet. For example, let's say that the browser, represented by this crocodile, sends a request through index.html, this Groot is the ServiceWorker, which for now does nothing and the request passes, goes to the internet, takes its time and comes back. The ServiceWorker has the ability to intercept these requests, for example, imagine that now our script saves static files from the server in the cache, we take the request for index.html, the SW notices that it already has that file locally, and prevents it from being make the request to the internet, sending the file, now you can see why in a progressive web application the internet is optional.

What do I need to use service workers?

- Service Worker Registration: You need to register a service worker script in your web application. This is typically done in your main JavaScript file using the navigator.serviceWorker.register() method.

- Service Worker Script: You need to create a JavaScript file that will serve as your service worker. This script will contain the logic for caching resources, handling network requests, and other tasks.

- Understanding of Promises and Fetch API: Service workers use promises and the Fetch API to handle asynchronous operations, so you need to have a basic understanding of these concepts.

- Testing and Debugging Tools: It's helpful to have tools for testing and debugging your service worker, such as Chrome DevTools or Firefox Developer Tools.

- Graceful Degradation: Since service workers are not supported in all browsers, it's important to implement a fallback strategy for browsers that do not support service workers, so your web application remains functional for all users.

- Browser support. In order to use the service worker we need a compatible browser. Today, the vast majority of browsers have already implemented it (such as Chrome or Firefox) or are in the process of achieving it (such as Safari or Edge).

- HTTPS. During the development phase, service workers can be used through “localhost”, as it is considered a secure server, even if it is not. However, it is not considered a secure way to enter the server using a server's IP, or loop IP. It is

important to note that in order to implement the service worker, we will need to configure HTTPS on the server.

Understanding the life cycle

The service worker lifecycle consists of several stages, from registration to activation and eventual termination. Here's an overview of the key stages:

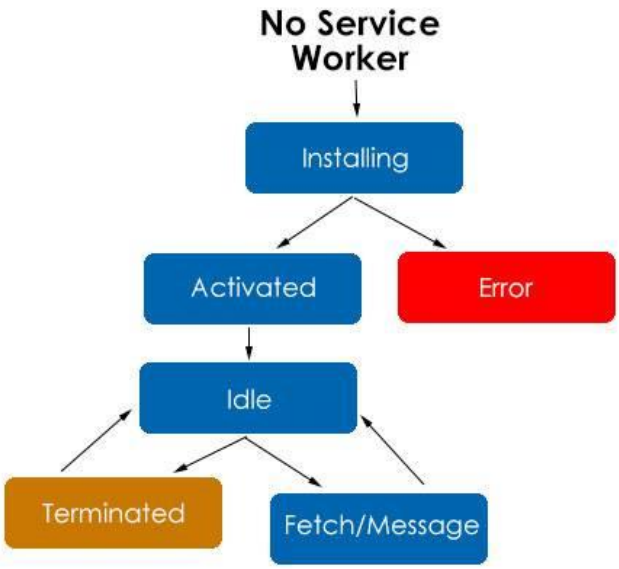


Ilustración 1. Understanding the life cycle

The life cycle of the service worker is totally independent of the website and its purpose is to achieve the best user experience, so it has different stages, as we see in the graph and we explain below in more detail focusing on the most important ones.

- Facility. In order to install a service worker we must register it in the JavaScript language of the page. Once registered, the browser begins the installation stage in the background. The installation is only performed once per service worker and to be successful the files must be properly cached.

- Activation. When we get the service worker installed correctly, it will be able to control the client and will go to the Activate state, allowing us to handle events.

- Terminated and Fetch. After activation, the service worker will control all possible pages, and two different states may appear: Terminated or memory saving mode and Fetch, which indicates that it is handling network requests.

- Fetch Events:When a controlled page makes a network request, the fetch event is fired in the service worker.You can intercept the request, serve a cached response, or fetch the resource from the network.

- Message Events:Service workers can communicate with pages using the postMessage API.

- When a message is sent from a page, the message event is fired in the service worker, allowing you to respond accordingly.

- Sync Events:Service workers can register for sync events, which are fired when the browser has network connectivity and can perform background synchronization tasks.

- This is useful for tasks like sending analytics data or updating content in the background.
- Idle:Once activated and controlling clients, the service worker enters the idle state, waiting for events to occur.

- During this state, the service worker can receive fetch, message, and sync events.

Once you know what happens when you register a ServiceWorker you can identify the states it may be in.

- Parsing - This is where the browser interprets the ServiceWorker code and validates that everything is correct, in addition, at this point our ServiceWorker could fail if we are not serving the page via https, because yes, this technology only works with a secure connection.

- Installing - In the installation process, this process can be delayed if within the ServiceWorker we use the waitUntil method of the installation event, we will talk about that later.

- Installed - When the ServiceWorker has finished installing, it goes to the installed or waiting state as it is also known, as we talked about previously.

- Activated - If a ServiceWorker did not exist or if we deliberately skipped the wait in the code, or if the user left the page overriding the previous ServiceWorker, congratulations, your new ServiceWorker passes the activated state and begins controlling the document and the network.

- Redundant - When a ServiceWorker is invalidated or the installation process fails, it goes into this state, in which we can assume, it is no longer valid.

How to add a service worker?

We can add a service worker in a simple way: just create the file that contains the script code and then register it using the register() of the browser API. We see it step by step:

- Create your service worker's JavaScript file. The first thing we will do is create the JavaScript file sw.js and we will place it in the root of the website. It is usually located at the same level as index.html. To begin, we can place the sw.js file blank, although it is interesting to know that there are different libraries and services to generate service worker code and make the task easier, such as the Workbox created by Google.

- Register the service worker. Once the sw.js file is created, we will register the service worker with the help of more JavaScript code. This process can be handled through a single call to a browser service worker API method. To do this we must make sure that the client supports said technology.

This is an example of the code to register the Service Worker:

```
if ('serviceWorker' in navigator) {
  window.addEventListener('load', function() {
    navigator.serviceWorker.register('/sw.js').then(function(registration) {
      // Si es exitoso
      console.log('SW registrado correctamente');
    }, function(err) {
      // Si falla
      console.log('SW fallo', err);
    });
  });
}
```

Ilustración 2. Example Service Worker

- View the registered service worker. To verify if we have correctly registered the service worker, we can check it through the developer tools, such as Chrome Developer Tools.

Registering a ServiceWorker is done somewhere in your JavaScript code, it can be as simple as opening a script tag in your HTML document and placing the registration code.

```
<script>
navigator.serviceWorker.register('/sw.js')
</script>
```

Ilustración 3. Register Service worker

The navigator object of the browser's global scope has a serviceWorker property which in turn contains a register method.

This method registers the new ServiceWorker, we only need to send the path where the JavaScript file that defines the ServiceWorker that we want to register will be found.

Here is something very important, where you place your ServiceWorker impacts its scope, since a ServiceWorker will only process requests that are executed in the same scope in which it was registered.

This means that if you place your ServiceWorker in a scripts folder and register it like this:

```
<script>
navigator.serviceWorker.register('/scripts/sw.js')
</script>
```

Ilustración 4. Register Service worker navigator.

Only requests that start with /script/ will go through the ServiceWorker and the rest will not. So for 99% of the cases, you will want to place your ServiceWorker in the root of your project, that is, you should not place it inside subfolders, so that all requests can be processed by it.

A very common name, in fact, is sw.js, remember that using conventions is a good idea, take advantage of it and place your ServiceWorker in the root of your project, with that name.

The result of registering a ServiceWorker is a promise, which when resolved gives us relevant information about the ServiceWorker such as the state it is in. During the course we will be working with this information so don't forget to take it into account.

Events of a ServiceWorker

We typically define functionality for a ServiceWorker, binding functions to events that happen while the ServiceWorker is active.

```
self.addEventListener('activate', function(event) {});

self.addEventListener('install', function(event) {});
```

Ilustración 5. Event Service Worker

The self object in the browser always points to the global scope, no matter where it is running, in this case, the global context is that of the ServiceWorker, through it we can bind events, get clients and more.

```
self.addEventListener('push', function(event) {});

self.addEventListener('fetch', function(event) {});
```

Ilustración 6. Fetch and Push

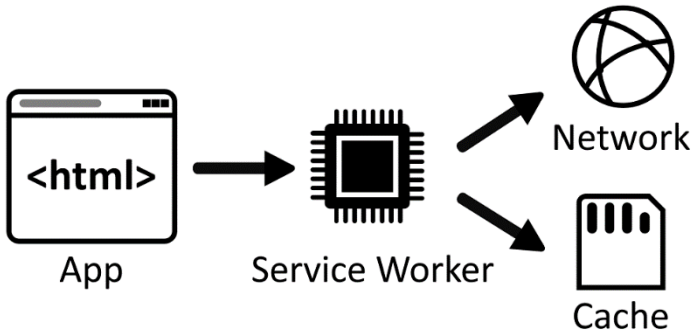
In addition to activation and installation, we also have the fetch and push events, which are triggered on each request to the network and when we receive a push notification respectively.

We also have a message event, which is very important because it is through this that we can communicate the DOM with our ServiceWorker.

```
self.addEventListener('message', function(event) {});
```

Ilustración 7. Event Message

As we mentioned before, some of the events that are part of the life cycle of a ServiceWorker are the installation and activation events, to which we can link code instructions as we see on the screen.



Ilustration 8. Service Worker

Service workers essentially act as proxy servers that sit between web applications, the browser, and the network (when available). They are intended, among other things, to enable the creation of effective offline experiences, intercept network requests and take appropriate action based on whether the network is available, and update assets residing on the server. They will also allow access to push notifications and background sync APIs.

A Service Worker is a type of JavaScript script that runs in the background of the web browser and is used to perform tasks that do not require direct interaction with the web page, such as cache management, push notifications, and background synchronization. They provide a way to manage the application's assets and requests, allowing for a more responsive and reliable user experience. Additionally, they can improve performance by caching resources and enabling offline functionality.

One of the key features of Service Workers is their ability to intercept and handle network requests. This allows developers to implement custom caching strategies, which can greatly improve the performance and reliability of web applications. Service Workers can also be used to handle push notifications, allowing web applications to send notifications to users even when the application is not open. This can be useful for delivering important updates and notifications to users in a timely manner.

Characteristics:

These are the most notable characteristics of a service worker:

- No DOM Access: Service workers cannot directly access the DOM of the pages they control. Instead, they communicate with pages using the postMessage interface, allowing for message-based communication.

- Network Request Control: As programmable network proxies, service workers can intercept and control how network requests from the pages they control are handled. This enables the implementation of custom caching strategies and offline capabilities.

- IndexedDB API: Service workers can maintain information using the IndexedDB API, which allows for the storage and retrieval of large amounts of structured data on the client side.

- Caching Systems: Service workers can implement different caching systems, such as cache-first, network-first, or stale-while-revalidate, to improve the performance and reliability of web applications.

- Background Execution: Service workers run in the background, regardless of whether the web page is open or not. This allows them to perform tasks like push notifications, background synchronization, and periodic updates.

- Event-Driven Programming: Service workers use a programming model based on events and promises. This allows them to respond to events such

as fetch, message, sync, and push, enabling them to perform actions based on these events asynchronously.

- Scope and Lifecycle: Service workers have a specific scope (the set of URLs for which they are responsible) and lifecycle (registration, installation, activation, idle, and termination), which determines when and how they are executed and interact with the web application .
- It cannot directly access the DOM, but instead communicates with the pages it controls using the PostMessage interface.

- As a programmable network proxy, it allows you to control how the page's network requests are handled.

- They are capable of maintaining information using the IndexedDB API.

- They can implement different caching systems.

- It runs in the background regardless of whether the web page is open or not.

- It can intercept and control network requests, allowing the creation of advanced caching strategies.

- Allows the implementation of functionalities such as push notifications and background synchronization.

- It uses a programming model based on events and promises.

Advantages:

- Improves application performance by enabling resource caching.
- Provides a faster and smoother user experience by loading content from cache instead of making network requests.
- It allows the creation of progressive web applications (PWA) that can work offline.
- Makes it easy to implement push notifications to keep users informed about important updates and events.
- Efficient Resource Caching: Service workers improve application performance by enabling efficient resource caching. They can cache static assets like HTML, CSS, JavaScript, and images, reducing the need for repeated network requests.
- Offline Functionality: Service workers enable the creation of progressive web applications (PWAs) that can work offline. By caching resources, PWAs can continue to function even when the user is offline, providing a seamless user experience.
- Reduced Network Usage: By loading content from cache instead of making network requests, service workers reduce the amount of data transferred over the network. This can lead to faster loading times and reduced bandwidth usage, particularly on mobile devices.
- Improved Reliability: Service workers can improve the reliability of web applications by serving cached content when the network is slow or

unavailable. This ensures that users can still access content even in challenging network conditions.

- Background Sync: Service workers allow for background synchronization, enabling web applications to sync data with a server even when the application is not actively in use. This can be useful for tasks like syncing user data or fetching updates in the background.

- Cross-platform Compatibility: Service workers are supported by most modern browsers, making them a cross-platform solution for enhancing web applications. This ensures a consistent user experience across different devices and browsers.

- Easy Implementation of Push Notifications: Service workers make it easy to implement push notifications in web applications. This allows developers to keep users informed about important updates and events, even when the application is not actively being used.

Disadvantages:

- It requires greater technical knowledge to implement, compared to other web technologies.

- It can introduce additional complexity to your application code due to its asynchronous and event-driven nature.

- It can consume device resources, especially on mobile devices with limited resources, if not managed correctly.

- Browser Support Limitations: While most modern browsers support service workers, older browsers and some mobile browsers do not. This can limit the reach of web applications that rely heavily on service workers for functionality.

- Security Concerns: Service workers have access to powerful features such as intercepting network requests and caching responses. If not implemented carefully, they can introduce security vulnerabilities, such as caching sensitive information or intercepting sensitive requests.

- Debugging Complexity: Debugging service workers can be challenging, as they run in a separate context from the main browser window. This can make it difficult to inspect and debug service worker code, especially when dealing with complex caching or synchronization logic.

- Cache Invalidation: Managing cached resources and ensuring they are updated when necessary can be challenging. Service workers must implement strategies for cache invalidation to ensure that users receive the latest content.

- Potential for Overuse: Service workers provide powerful capabilities, but if not used judiciously, they can lead to overuse and unnecessary consumption of device resources. Developers must carefully consider when and how to use service workers to avoid negatively impacting performance.

- Compatibility with Single Page Applications (SPAs): Service workers may not integrate seamlessly with certain single-page application (SPA) frameworks or architectures. Developers may need to make additional modifications to ensure compatibility and optimal performance.

- Increased Complexity in Development: Implementing service workers adds an additional layer of complexity to web development. Developers must have a solid understanding of service worker concepts and best practices, which can require additional time and effort to learn and implement.

Applications:

- Progressive Web Applications (PWA): Service Workers are essential for creating PWAs, as they allow the application to work offline and offer an experience similar to that of a native application.

- Advanced Cache Management: Service Workers can manage cache more efficiently than traditional browser-based strategies, enabling a better user experience by loading content faster.

- Push Notifications: Service Workers make it easy to implement push notifications in web applications, allowing developers to send notifications to users even when the application is not open in the browser.

- Background synchronization: Service Workers can perform tasks in the background, such as synchronizing data with a server, improving efficiency and user experience.

- Background Tasks: Service workers can be used to perform various background tasks, such as periodic data synchronization, updating content in the background, or pre-fetching resources that the user is likely to need. This can improve the overall efficiency and user experience of the web application.

- Offline Forms: Service workers can be used to store form data locally and submit it to the server when the network is available. This ensures that users can interact with forms even when offline, improving the usability of web applications in low-connectivity environments.

- Geo-location Services: Service workers can be used to implement geo-location services, allowing web applications to track the user's location in the background and provide location-based services or content.

- Load Balancing and Failover: Service workers can be used to implement intelligent load balancing and failover strategies, allowing web applications to switch between different servers or network paths based on availability and performance metrics.

- Real-time Collaboration: Service workers can facilitate real-time collaboration in web applications by managing shared data and synchronizing changes between users in the background.

- Resource Prefetching: Service workers can prefetch resources that are likely to be needed by the user, such as images, CSS, or JavaScript files, before they are actually requested. This can improve the perceived performance of the web application by reducing latency.

- Adaptive Content Delivery: Service workers can be used to deliver adaptive content based on the user's device capabilities, network conditions, or preferences. This can help optimize the user experience and reduce unnecessary data usage.

- Dynamic Content Updates: Service workers can be used to implement dynamic content updates, allowing web applications to update content in real-time without requiring the user to manually refresh the page.

Other use case ideas

Service workers are also intended to be used for such things as:

1. Background data synchronization.

2. Responding to resource requests from other origins.

3. Receiving centralized updates to expensive-to-calculate data such as geolocation or gyroscope, so multiple pages can make use of one set of data.

4. Client-side compiling and dependency management of CoffeeScript, less, CJS/AMD modules, etc. for development purposes.

5. Hooks for background services.

6. Custom templating based on certain URL patterns.

7. Performance enhancements, for example pre-fetching resources that the user is likely to need in the near future, such as the next few pictures in a photo album.

8. Background Data Synchronization: Service workers can be used to synchronize data in the background, ensuring that the user's data is always up-to-date even when the application is not actively in use.

9. Cross-Origin Resource Requests: Service workers can respond to resource requests from other origins, allowing for cross-origin resource sharing (CORS) and enabling more flexible and dynamic web applications.

10. Centralized Updates for Expensive Data: Service workers can receive centralized updates for expensive-to-calculate data, such as geolocation or gyroscope information. This allows multiple pages to make use of the same set of data, reducing duplication of effort and improving efficiency.

11. Offline Data Processing: Service workers can perform data processing tasks offline, such as parsing and analyzing large datasets, without requiring continuous network connectivity. This can be useful for applications that need to perform complex calculations or analysis on user data.

12. Enhanced Security Measures: Service workers can be used to implement enhanced security measures, such as verifying requests and responses for authenticity and preventing malicious attacks.

13. Improved Performance: By offloading resource-intensive tasks to service workers, web applications can achieve improved performance and responsiveness, leading to a better overall user experience.

14. Customized Content Delivery: Service workers can be used to deliver customized content based on user preferences, location, or other factors, enhancing the personalization of web applications.

15. Content Pre-fetching: Service workers can pre-fetch content that is likely to be needed by the user, based on their browsing history or behavior, improving the speed and efficiency of content delivery.

16. Background Notifications: Service workers can be used to handle background notifications, such as alerting users to new messages or updates, even when the application is not actively in use.

API mocking.

In the future, service workers will be able to do a number of other useful things for the web platform that will bring it closer towards native app viability. Interestingly, other specifications can and will start to make use of the service worker context, for example:

1. Background synchronization: Start up a service worker even when no users are at the site, so caches can be updated, etc.
2. Reacting to push messages: Start up a service worker to send users a message to tell them new content is available.
3. Reacting to a particular time & date.
4. Entering a geo-fence.

Conclution

Service workers are a valuable tool for developers looking to create high-performing and reliable web applications. Their ability to work offline, manage caching effectively, and handle background tasks makes them indispensable for modern web development. As web technologies continue to evolve, service workers will remain a key technology for creating dynamic and engaging web experiences.

Bibliografía

docs, m. w. (2022 de Diciembre de 05). *Developer Mozilla*. Recuperado el 02 de February de 2024, de Org: https://developer.mozilla.org/es/docs/Web/API/Service_Worker_API

Facilito, C. (2023 de 07 de 08). *Código Facilito*. Obtenido de https://codigofacilito.com/articulos/que-es-sw-pwa

MSEEdgeTeam, M. H. (03 de 04 de 2023). *Microsoft*. Obtenido de Learn: https://learn.microsoft.com/es-es/microsoft-edge/progressive-web-apps-chromium/how-to/service-workers