English

Repositorie made for my PWA technologie presentation, it’s an example with minimun js function so you can take it to have a code base for your PWA.

Service Worker PWA

In the context of a Progressive Web Application (PWA), a service worker is a JavaScript file that runs separately from the browser's main thread, acting as a programmable proxy between the web application, the browser, and the network. Here's an overview of what a service worker is, its purpose, and how to set it up:

What is a service worker?

- A service worker is a script that your browser runs in the background, independently of a web page, enabling functionalities that don't require a web page or user interaction.
- It can intercept and manage network requests, cache resources, and provide offline functionality.

Purpose of a service worker:

- Offline support: Service workers can cache resources, allowing PWAs to continue functioning even when the user is offline.
- Performance: They enable efficient caching strategies, reducing server load and providing faster response times.
- Push notifications: Service workers can receive push notifications, enabling engagement even when the application isn't open.
- Background synchronization: They can synchronize data with a server in the background, allowing seamless user experiences.

Setting up a service worker:

- Create a JavaScript file for your service worker, conventionally named service-worker.js or sw.js.
- Register the service worker in your main JavaScript file or HTML file using the navigator.serviceWorker.register() method.

- Implement event listeners in the service worker to handle different events such as installation, activation, fetch, push notifications, etc.
- Define caching strategies in the service worker to cache resources for offline access.
- Test your PWA with the service worker locally and deploy it to your hosting environment.

Preparation

Choose your deployment:

You can either deploy it via localhost, but you need to index your site on www.

You can use Plesk, but you will need to upload your files to Plesk each time.

name.webmanifest

Create a web manifest:

In the main index folder, create a file called oclock.webmanifest.

Next, we'll fill out the manifest with the bare minimum for now:

```
{
  "name": "O' Clock",
  "short_name": "Oclock",
  "start_url": "/?source=pwa",
  "display": "standalone",
  "background_color": "#fdfdfd",
  "theme_color": "#241a91",
  "orientation": "portrait-primary",
  "icons": [
    {
      "src": "/img/clock192.png",
      "type": "image/png",
      "sizes": "192x192"
    },
    {
      "src": "/img/clock512.png",
      "type": "image/png",
```

```
      "sizes": "512x512"
    }
  ],
  "screenshots": [
    {
      "src": "/img/screenshot1.png",
      "type": "image/png",
      "sizes": "720x405",
      "form_factor": "narrow"
    },
    {
      "src": "/img/screenshot2.png",
      "type": "image/jpg",
      "sizes": "1315x740",
      "form_factor": "wide"
    }
  ]
}
```

Don’t forget to link the manifest in your index.html/php file

<link rel="manifest" href="/name.webmanifest">

swRegister.js

addEventListener of 'load':

- This 'load' event is triggered when all elements of the web page have been loaded, including external resources such as images, scripts, and stylesheets.
- Within this event, an asynchronous function `registerServiceWorker` is defined to register the Service Worker.
- First, we check if the browser supports Service Workers by checking if the `serviceWorker` property is present in the `navigator` object.

- Then, we use `navigator.serviceWorker.register()` to register the Service Worker specified by the file 'sw.js'. We also provide a `scope` option that defines the scope of the Service Worker.
- Once the Service Worker is registered, we check its state using the properties `registration.installing`, `registration.waiting`, and `registration.active` to determine in which state it is. These states indicate whether the Service Worker is currently being installed, already installed, or active.
- Appropriate logging messages are displayed based on the state of the Service Worker.
- In case of an error during the registration of the Service Worker, an error message is logged to the console for potential debugging.

sw.js

Function Install:

- The 'install' event is triggered when the Service Worker is being installed in the browser.
- During this event, the Service Worker performs operations necessary to initialize its functionality.
- Within this event, the waitUntil method is used to instruct the browser not to consider the installation complete until all asynchronous operations are finished.
- The code block inside the function associated with the 'install' event is an asynchronous function (async () => {...}), meaning it can contain asynchronous operations such as opening caches and caching resources.
- In this example, the first operation performed is the call to self.skipWaiting(), which directs the Service Worker to immediately transition to the "active" state without waiting for existing tabs to close.
- Next, essential resources for the application are cached by calling the addResourcesToCache function, which takes a list of resources to cache as a parameter.

- All these operations are encapsulated within an asynchronous function to ensure they are properly executed before the installation is considered complete.
- In case of an error during installation, error messages are logged to the console for potential debugging.

Function Activate:

- The 'activate' event is triggered after the Service Worker has been successfully installed and is ready to be used.
- During activation, the Service Worker cleans up old caches to ensure that only the latest versions of resources are retained.
- The waitUntil method is used to instruct the browser not to complete the activation event until the promise returned by the asynchronous function is resolved or rejected. This ensures that all asynchronous operations, such as cache deletion, are completed before considering the activation complete.
- Within the asynchronous function, clients.claim() is used to immediately take control of client pages and avoid conflicts with old Service Worker instances.
- Next, the Service Worker retrieves the list of keys of existing caches with caches.keys(). It then iterates over these keys to check if they match the current cacheName. If there are caches with different names, they are deleted.
- Potential errors are handled and logged to the console for potential debugging.

Function addResourcesToCache:

- This addResourcesToCache function is responsible for adding resources to the Service Worker cache.
- It takes a list of resources as a parameter, which are the URLs of the resources to be cached.
- Inside the function, we start by opening the cache specified by the cacheName using the caches.open() method. This retrieves or creates a cache with the specified name.

- Then, all provided resources are added to the cache using the cache.addAll() method.
- This function is declared as asynchronous (async), meaning it returns a promise that will be resolved when all asynchronous operations inside the function are completed.
- A comment is added describing the function's role and specifying the type of parameter it takes, which aids in understanding its usage and functionality.

Function putInCache:

- This putInCache function is responsible for adding a request-response pair to the Service Worker cache.
- It takes two parameters: the request and its corresponding response.
- Inside the function, we begin by opening the cache specified by the cacheName using the caches.open() method.
- Then, the request-response pair is added to the cache using the cache.put() method.
- Since the function is declared as asynchronous (async), it returns a promise that will be resolved once all asynchronous operations inside the function are completed.
- A comment is included explaining the function's role, as well as the types of parameters it expects, which facilitates understanding of its usage and functionality.

Function cacheFirst:

- This cacheFirst function implements a cache-first caching strategy in the Service Worker.
- It takes an options object as a parameter, containing the request to be processed (options.request) and the fallback URL (options.fallbackUrl) to use in case of resource retrieval failure.
- Inside the function, we first check if the requested resource is available in the cache. If it is, we return the cached response. Otherwise, we attempt to retrieve the resource from the network and cache it before returning it.

- In case of an error or network unavailability, we return a fallback response, either retrieved from the cache or a generic response indicating a network error.

Function enableNavigationPreload:

- This enableNavigationPreload function is responsible for enabling navigation preload in the Service Worker if this feature is supported by the browser.
- It first checks if navigation preload is supported by verifying if self.registration.navigationPreload is defined.
- If navigation preload is supported, it is enabled by calling self.registration.navigationPreload.enable().

addEventListener Fetch:

- This 'fetch' event is triggered each time a resource is fetched by the application.
- Within this event, we utilize the cache-first caching strategy (cacheFirst) to handle requests.
- The cacheFirst function is called with parameters request, preloadResponsePromise, and fallbackUrl. This function is responsible for managing the resource retrieval logic by first checking the cache, then fetching from the network if necessary, with proper error handling and fallback response if needed.

Français

Ce repositorie a été fait pour ma présentation de la technologie PWA, c’est un exemple avec un minimun de fonction js donc vous pouvez l’utiliser pour une base de codage de votre PWA.

Service Worker PWA

Dans le contexte d'une application web progressive (PWA), un service worker est un fichier JavaScript qui s'exécute séparément du thread principal du navigateur, agissant comme un proxy programmable entre l'application web, le navigateur et le réseau.
Voici un aperçu de ce qu'est un service worker, de son but et de comment le configurer :

Qu'est-ce qu'un service worker ?

- Un service worker est un script que votre navigateur exécute en arrière-plan, séparément d'une page web, permettant des fonctionnalités qui ne nécessitent pas de page web ou d'interaction utilisateur.
- Il peut intercepter et gérer les requêtes réseau, mettre en cache des ressources et fournir une fonctionnalité hors ligne.

But d'un service worker :

- Support hors ligne : Les service workers peuvent mettre en cache des ressources, permettant aux PWA de continuer à fonctionner même lorsque l'utilisateur est hors ligne.
- Performance : Ils permettent des stratégies de mise en cache efficaces, réduisant la charge du serveur et fournissant des temps de réponse plus rapides.
- Notifications push : Les service workers peuvent recevoir des notifications push, permettant l'engagement même lorsque l'application n'est pas ouverte.
- Synchronisation en arrière-plan : Ils peuvent synchroniser des données avec un serveur en arrière-plan, permettant des expériences utilisateur transparentes.

Configuration d'un service worker :

- Créez un fichier JavaScript pour votre service worker, conventionnellement nommé service-worker.js ou sw.js.
- Enregistrez le service worker dans votre fichier JavaScript principal ou fichier HTML en utilisant la méthode navigator.serviceWorker.register().
- Implémentez des écouteurs d'événements dans le service worker pour gérer différents événements tels que l'installation, l'activation, la récupération, les notifications push, etc.

- Définissez des stratégies de mise en cache dans le service worker pour mettre en cache des ressources pour un accès hors ligne.
- Testez votre PWA avec le service worker localement et déployez-la sur votre environnement d'hébergement.

Préparation

Choisissez votre déploiment:

Vous pouvez soit le faire par le localhost mais il faut mettre votre site à l’index de www

Vous pouvez utiliser plesk mais il faudra télécharger à chaque fois vos fichier sur plesk

name.webmanifest

Créer un web manifest:

Dans l’index dossier principal vous créez un fichier que l’on va appelé oclock.webmanifest

Ensuite on va remplir le manifest pour le moment du minimun syndical:

```
{
  "name": "O' Clock",
  "short_name": "Oclock",
  "start_url": "/?source=pwa",
  "display": "standalone",
  "background_color": "#fdfdfd",
  "theme_color": "#241a91",
  "orientation": "portrait-primary",
  "icons": [
    {
      "src": "/img/clock192.png",
```

```
      "type": "image/png",
      "sizes": "192x192"
    },
    {
      "src": "/img/clock512.png",
      "type": "image/png",
      "sizes": "512x512"
    }
  ],
  "screenshots": [
    {
      "src": "/img/screenshot1.png",
      "type": "image/png",
      "sizes": "720x405",
      "form_factor": "narrow"
    },
    {
      "src": "/img/screenshot2.png",
      "type": "image/jpg",
      "sizes": "1315x740",
      "form_factor": "wide"
    }
  ]
}
```

Ne pas oublier de lier le manifest à votre index.html/php

<link rel="manifest" href="/name.webmanifest">

swRegister.js

addEventListener de ‘load’:

- Cet événement `'load'` est déclenché lorsque tous les éléments de la page web ont été chargés, y compris les ressources externes telles que les images, les scripts et les feuilles de style.

- À l'intérieur de cet événement, une fonction asynchrone `registerServiceWorker` est définie pour enregistrer le Service Worker.
- Tout d'abord, on vérifie si le navigateur prend en charge les Service Workers en vérifiant si la propriété `serviceWorker` est présente dans l'objet `navigator`.
- Ensuite, on utilise `navigator.serviceWorker.register()` pour enregistrer le Service Worker spécifié par le fichier `'sw.js'`. On lui fournit également une option `scope` qui définit la portée du Service Worker.
- Une fois le Service Worker enregistré, on vérifie son état à l'aide de la propriété `registration.installing`, `registration.waiting` et `registration.active` pour déterminer dans quel état il se trouve. Ces états indiquent si le Service Worker est en cours d'installation, déjà installé ou actif.
- Les messages de journalisation appropriés sont affichés en fonction de l'état du Service Worker.
- En cas d'erreur lors de l'enregistrement du Service Worker, un message d'erreur est enregistré dans la console pour le débogage éventuel.

sw.js

Function Install:

- L'événement `'install'` est déclenché lors de l'installation du Service Worker dans le navigateur.
- Pendant cet événement, le Service Worker effectue les opérations nécessaires à l'initialisation de son fonctionnement.
- Dans cet événement, on utilise la méthode `waitUntil` pour indiquer au navigateur de ne pas considérer l'installation comme terminée tant que toutes les opérations asynchrones ne sont pas terminées.
- Le bloc de code à l'intérieur de la fonction associée à l'événement `'install'` est une fonction asynchrone (`async () =>`

`{...}`), ce qui signifie qu'il peut contenir des opérations asynchrones comme l'ouverture de caches et la mise en cache de ressources.

- Dans cet exemple, la première opération effectuée est l'appel à `self.skipWaiting()`, qui indique au Service Worker de passer immédiatement à l'état "actif" sans attendre que les onglets existants soient fermés.
- Ensuite, des ressources essentielles pour l'application sont mises en cache en appelant la fonction `addResourcesToCache`, qui prend en paramètre une liste de ressources à mettre en cache.
- Toutes ces opérations sont encapsulées dans une fonction asynchrone pour garantir qu'elles sont correctement exécutées avant que l'installation ne soit considérée comme terminée.
- En cas d'erreur lors de l'installation, des messages d'erreur sont enregistrés dans la console pour le débogage éventuel.

Function Activate:

- L'événement `'activate'` est déclenché après que le Service Worker a été installé avec succès et est prêt à être utilisé.
- Pendant l'activation, le Service Worker nettoie les anciens caches afin de garantir que seules les versions les plus récentes des ressources sont conservées.
- La méthode `waitUntil` est utilisée pour indiquer au navigateur de ne pas terminer l'événement d'activation tant que la promesse retournée par la fonction asynchrone n'est pas résolue ou rejetée. Cela garantit que toutes les opérations asynchrones, telles que la suppression des caches, sont terminées avant de considérer l'activation comme terminée.
- Dans la fonction asynchrone, `clients.claim()` est utilisé pour prendre immédiatement le contrôle des pages clientes et éviter les conflits avec les anciennes instances du Service Worker.

- Ensuite, le Service Worker récupère la liste des clés des caches existants avec `caches.keys()`. Il itère ensuite sur ces clés pour vérifier si elles correspondent au `cacheName` actuel. S'il y a des caches avec des noms différents, ils sont supprimés.
- Les erreurs potentielles sont gérées et enregistrées dans la console pour le débogage éventuel.

Function addResourcesToCache:

- Cette fonction `addResourcesToCache` est responsable de l'ajout des ressources au cache du Service Worker.
- Elle prend en paramètre une liste de ressources, qui sont les URL des ressources à mettre en cache.
- À l'intérieur de la fonction, on commence par ouvrir le cache spécifié par le nom `cacheName` en utilisant la méthode `caches.open()`. Cela permet de récupérer ou de créer un cache avec le nom spécifié.
- Ensuite, toutes les ressources fournies sont ajoutées au cache en utilisant la méthode `cache.addAll()`.
- Cette fonction est déclarée comme asynchrone (`async`), ce qui signifie qu'elle renvoie une promesse qui sera résolue lorsque toutes les opérations asynchrones à l'intérieur de la fonction sont terminées.
- J'ai également ajouté un commentaire décrivant le rôle de la fonction et précisant le type de paramètre qu'elle prend, ce qui facilite la compréhension de son utilisation et de son fonctionnement.

Function putInCache:

- Cette fonction `putInCache` est chargée d'ajouter une paire de requête-réponse au cache du Service Worker.
- Elle prend deux paramètres : la requête (`request`) et sa réponse correspondante (`response`).
- À l'intérieur de la fonction, on commence par ouvrir le cache spécifié par le nom `cacheName` en utilisant la méthode `caches.open()`.

- Ensuite, la paire de requête-réponse est ajoutée au cache à l'aide de la méthode `cache.put()`.
- Comme la fonction est déclarée comme asynchrone (`async`), elle renvoie une promesse qui sera résolue une fois que toutes les opérations asynchrones à l'intérieur de la fonction sont terminées.
- J'ai inclus un commentaire expliquant le rôle de la fonction, ainsi que les types de paramètres qu'elle attend, ce qui facilite la compréhension de son utilisation et de son fonctionnement.

Function cacheFirst:

- Cette fonction `cacheFirst` implémente une stratégie de mise en cache prioritaire dans le Service Worker.
- Elle prend en paramètre un objet `options` contenant la requête à traiter (`options.request`) et l'URL de secours (`options.fallbackUrl`) à utiliser en cas d'échec de récupération de la ressource.
- À l'intérieur de la fonction, on vérifie d'abord si la ressource demandée est disponible dans le cache. Si c'est le cas, on renvoie la réponse mise en cache. Sinon, on tente de récupérer la ressource depuis le réseau et on la met en cache avant de la renvoyer.
- En cas d'erreur ou d'indisponibilité du réseau, on renvoie une réponse de secours, soit récupérée depuis le cache, soit une réponse générique indiquant une erreur réseau.

Function enabledNavigationPreload:

- Cette fonction `enableNavigationPreload` est responsable d'activer la précharge de navigation dans le Service Worker, si cette fonctionnalité est prise en charge par le navigateur.
- Elle vérifie d'abord si la précharge de navigation est prise en charge en vérifiant si `self.registration.navigationPreload` est défini.
- Si la précharge de navigation est prise en charge, elle est activée en appelant `self.registration.navigationPreload.enable()`.

AddEventListener Fetch:

- Cet événement `'fetch'` est déclenché chaque fois qu'une ressource est récupérée par l'application.
- Dans cet événement, on utilise la stratégie de mise en cache prioritaire (`cacheFirst`) pour traiter les requêtes.
- La fonction `cacheFirst` est appelée avec les paramètres `request`, `preloadResponsePromise` et `fallbackUrl`. Cette fonction est chargée de gérer la logique de récupération de ressources en vérifiant d'abord dans le cache, puis en récupérant depuis le réseau si nécessaire, avec une gestion appropriée des erreurs et une réponse de secours en cas de besoin.