

# PWA Création d'un Service worker

Y. Stroppa  
PSB M2

# Sommaire

## **Installation et démarrage de nodejs**

### **Préparation de l'application**

Register

Installation

Activate

Mode offline

# Réalisation sous NodeJS

## Description de NodeJS

## Fonctionnement d'un conteneur NodeJS

- Docker
- intérêts
- installation d'un conteneur nodejs sous votre machine

# Installation de nodeJS

## Organisation du répertoire et démarrage

```
const express = require('express');
const app = express();

// This serves static files from the specified directory
app.use(express.static(__dirname));

const server = app.listen(8081, () => {

  const host = server.address().address;
  const port = server.address().port;

  console.log('App listening at http://%s:%s', host, port);
});
```

fichiers à créer sous un répertoire  
server.js  
package.json

```
{
  "name": "psb-lab",
  "version": "2.0.0",
  "description": "",
  "main": "server.js",
  "directories": {
    "test": "test"
  },
  "scripts": {
    "test": "echo \\\"Error: no test specified\\\"
&& exit 1",
    "start": "node server.js"
  },
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "express": "^4.16.3"
  }
}
```

# Installation de nodeJS

**on rajoutera un index.html et un fichier vide dans js/main.js**

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta name="viewport" content="width=device-width, minimum-scale=1.0, initial-scale=1.0, user-scalable=yes">
    <meta charset="utf-8">
    <title>Promises Lab</title>
    <link rel="stylesheet" href="styles/main.css">
  </head>
  <body>
    <header>
      <h1>PSB Cours PWA Service worker</h1>
    </header>
    <main>
      <label for="country">Country Name:</label>
      <input id="country" type="text" placeholder="enter country name"><br><br>
      <button id="get-image-name">Get Image Name</button><br><br>
      <button id="fetch-flag-image">Fetch Flag Image</button><br><br>
      <div class="img-container" id="img-container">
        <!-- image added dynamically -->
      </div>
    </main>
    <footer>
      <a href="https://github.com/google-developer-training/pwa-training-labs">GitHub</a>
    </footer>
    <script src="js/main.js"></script>
  </body>
</html>
```

# Installation de nodeJS

on rajoutera sous /styles le fichier main.css

```
body {
  align-items: center;
  display: flex;
  flex-direction: column;
  font-family: 'Roboto', 'Helvetica', 'Arial', sans-serif;
  height: 100%;
  justify-content: space-between;
  margin: 0;
  min-height: 100vh;
  text-align: center;
}

header {
  background-color: #2196f3;
  box-shadow: 0 2px 2px 0 rgba(0, 0, 0, 0.14), 0 3px 1px -2px rgba(0, 0, 0, 0.2), 0 1px 5px 0 rgba(0, 0, 0, 0.12);
  color: white;
  margin-bottom: 10px;
  width: 100%;
}

main {
  width: 100%;
}

label {
  padding: 5px;
  text-align: left;
}

input {
  border: 1px black solid;
  border-radius: 3px;
  font-size: 14px;
  min-height: 20px;
}
```

# Installation de l'application

**Il faut mettre à jour et télécharger les packages spécifiques à Node.js à l'aide de la commande**

**#npm install**

**Téléchargement et installation dans le répertoire node\_modules des packages supplémentaires**

**Ensuite, on peut démarrer l'application à l'aide de la commande**

**#npm start**

**ou**

**#node server.js**

# Transformation de l'application en PWA

## Installation du service worker -- Vérification compatibilité

**Avant de commencer il faut vérifier que votre navigateur supporte bien le service worker**

**Pour ce faire : dans le fichier main.js ajoutez les commandes**

```
// vérifier compatibilité du service worker
if ('serviceWorker' in navigator) {
    console.log("Service Worker : Supported");
}
```

Pour vérifier, réactualisez le navigateur et regardez dans la console



# Service worker Register : LOAD

**Pour que le service worker puisse s'enregistrer il est nécessaire d'ajouter les instructions suivantes dans le fichier main.js: Promise**

```
// Verification navigateur supporte les services workers
if ('serviceWorker' in navigator) {
  // inscription du service workler
  window.addEventListener('load', () => {
    navigator.serviceWorker
      .register('../sw_cached_pages.js')
      .then (reg => console.log("Service worker : Register"))
      .catch(err=> console.log(`Service worker: Error ${err}`))
  })
}
```

Il faudra créer le fichier sw\_cached\_pages.js sur la racine de votre répertoire vide pour le moment

# Détail des instructions

```
window.addEventListener('load', () => {  
  navigator.serviceWorker  
    .register('../sw.js')  
    .then (reg => console.log("Service worker : Register"))  
    .catch(err=> console.log(`Service worker: Error ${err}`))  
})
```

On invoque dans une promesse, l'objet `serviceWorker` en lui indiquant l'url du fichier du service worker et dans le cas où l'enregistrement se passe bien on affiche dans la console Register et dans le cas contraire Error et la nature de l'erreur. On fera attention à l'utilisation des côtes dans le deuxième cas qui n'est pas sans nous rappeler la syntaxe d'un shell (exécution de commande).

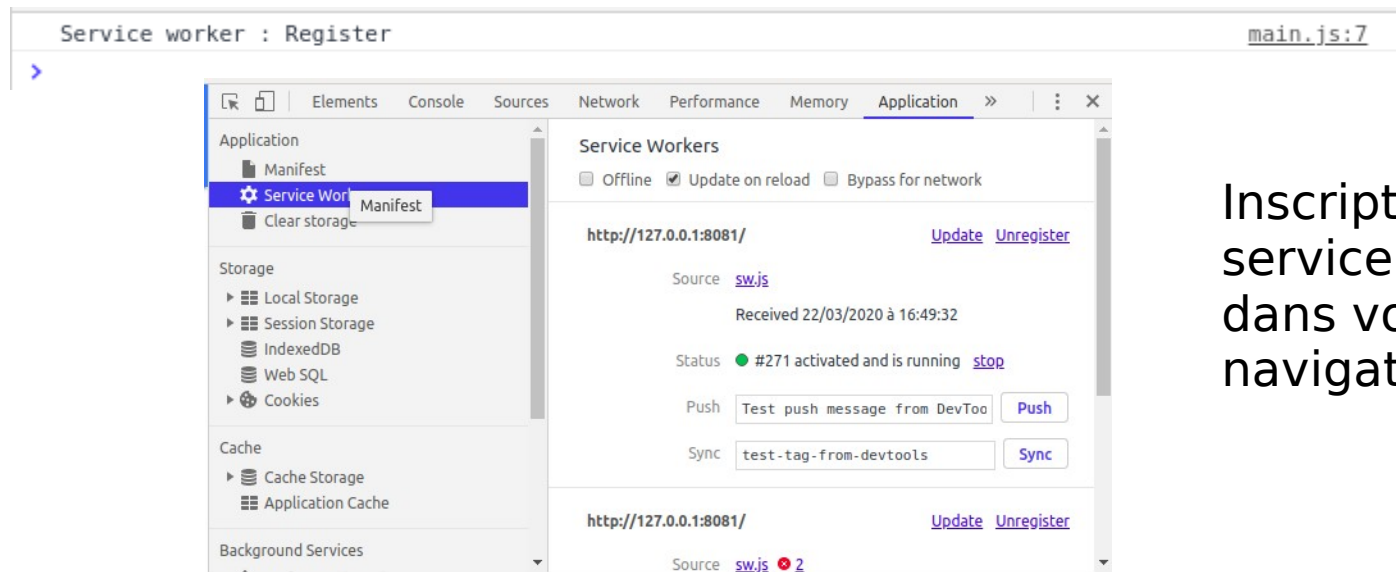
# Résultats

Erreur : pas de fichier sw.js

✖ A bad HTTP response code (404) was received when fetching the script.

Service worker: Error TypeError: Failed to register a ServiceWorker for scope ('http://127.0.0.1:8081/') with script ('http://127.0.0.1:8081/sw.js'): A bad HTTP response code (404) was received when fetching the script.

Tout se passe normalement : le fichier sw.js existe



Inscription du  
service worker  
dans votre  
navigateur

# Quelques précisions : cache du SW

L'interface Cache de l' API ServiceWorker représente le stockage pour les paires d'objet Request / Response object qui sont mises en cache dans le cadre du cycle de vie d'un ServiceWorker.

Vous avez la **responsabilité d'implémenter** la façon dont le script du ServiceWorker gère les mises à jour du Cache. Les éléments dans un Cache ne sont pas mis à jour à moins d'une requête explicite; ils **n'expirent** pas non plus à moins d'être supprimés. Utiliser `CacheStorage.open(cacheName)` pour ouvrir un objet Cache nommé spécifique et appeler ensuite toute méthode de l'objet Cache pour faire la maintenance du Cache.

Vous avez également la responsabilité de la purge périodique des entrées du cache. Chaque navigateur dispose d'une limite physique quant à la capacité de stockage qu'un service worker donné peut utiliser. Le navigateur fait de son mieux pour gérer l'espace disque, mais il peut supprimer le stockage du Cache pour une origine. Le navigateur supprimera généralement toutes les données d'une origine ou rien. Il faut s'assurer de versionner les caches par nom et d'utiliser seulement les caches à partir de la version du ServiceWorker avec lequel ils peuvent correctement fonctionner. Voir Supprimer des anciens caches pour plus d'informations.

# Quelques précisions : cache du SW

**Cache.match(request, options)**

Retourne une Promise qui se résout en une réponse associée à la première requête correspondante dans l'objet Cache.

**Cache.matchAll(request, options)**

Retourne une Promise qui se résout en un tableau de toutes les requêtes correspondantes dans l'objet Cache.

**Cache.add(request)**

Prend une URL, la récupère et ajoute l'objet réponse associé au cache donné. C'est une fonctionnalité équivalente à l'appel de fetch(), puis à l'utilisation de Cache.put() pour ajouter les résultats au cache.

**Cache.addAll(requests)**

Prend un tableau d'URLs, les récupère, et ajoute les objets réponses associés au cache donné.

**Cache.put(request, response)**

Prend à la fois une requête et sa réponse et l'ajoute au cache donné.

**Cache.delete(request, options)**

Trouve l'entrée Cache dont la clé est la requête, et le cas échéant, supprime l'entrée Cache et retourne une Promise qui se résout à true. Si aucune entrée Cache n'est trouvée, elle retourne false.

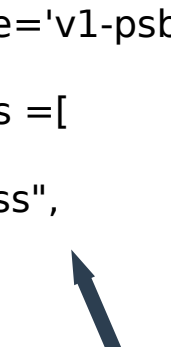
**Cache.keys(request, options)**

Retourne une Promise qui se résout en un tableau clés Cache.

# Service worker : install

On va dans cette étape s'occuper de l'installation du service worker ( au sein du navigateur client). La question à ce niveau est quels sont les fichiers que l'on souhaite copier dans le cache du navigateur pour une utilisation offline ... dans notre exemple très simple ce sont les fichiers index.html, styles/main.css, js/main.js. Maintenant dans le fichier sw.js il faut lui indiquer ces différentes informations.

Nom du cache



```
const cacheName='v1-psb';

const cacheAssets =[
  "index.html",
  "/styles/main.css",
  "/js/main.js"
];
)
```

Liste des fichiers à stockés

```
self.addEventListener('install',(e) => {
  console.log('service worker: installed');
  e.waitUntil (
    caches
      .open(cacheName)
      .then(cache => {
        console.log('service worker: caching files');
        cache.addAll(cacheAssets);
      })
      .then(()=> self.skipWaiting())
  )
})
```

Invocation d'une promesse

# ExtendableEvent.waitUntil()

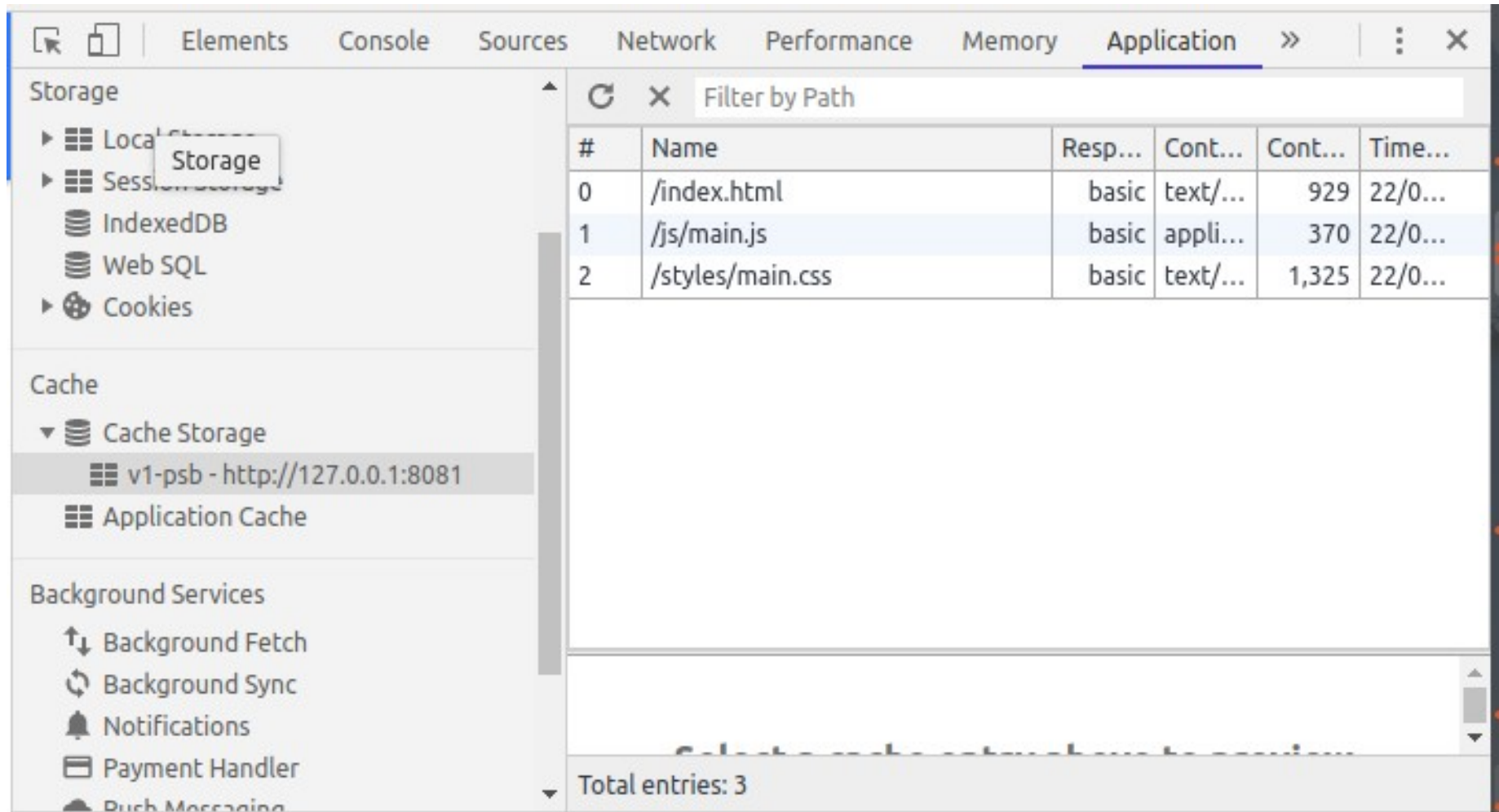
Les événements d'installation des services worker utilisent `waitUntil` pour tenir le service worker jusqu'à ce que la phase d'installation soit terminée. Si la promesse passée à `waitUntil` est rejetée, l'installation est considérée comme échouée et annulée. C'est principalement l'objectif recherché, de ne considérer comme installé que si l'ensemble des dépendances ont bien été copiées dans le cache.

**Syntaxe :**

**`event.waitUntil(promise)`**

```
addEventListener('install', event => {  
  const preCache = async () => {  
    const cache = await caches.open('static-v1');  
    return cache.addAll([  
      '/',  
      '/about/',  
      '/static/styles.css'  
    ]);  
  };  
  event.waitUntil(preCache());  
});
```

# Service worker : installed



The screenshot shows the Chrome DevTools Application tab. The left sidebar is expanded to the 'Storage' section, specifically the 'Cache' subsection. Under 'Cache Storage', the entry 'v1-psb - http://127.0.0.1:8081' is selected. The main panel displays a table of cached resources. The table has columns: #, Name, Resp..., Cont..., Cont..., and Time... The table contains three entries:

#	Name	Resp...	Cont...	Cont...	Time...
0	/index.html	basic	text/...	929	22/0...
1	/js/main.js	basic	appli...	370	22/0...
2	/styles/main.css	basic	text/...	1,325	22/0...

At the bottom of the panel, it says 'Total entries: 3'.

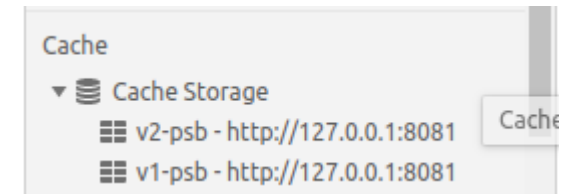


# Service worker - Activate

**Dans cette étape comme indiqué dans les pages précédentes, il faut s'occuper de la gestion des caches installés sur le navigateur. Si on change le nom du cache par v2.psb on se retrouve avec deux éléments dans le cache comme suit :**

```
// lors de l'activation du service worker nettoyage des caches
self.addEventListener('activate', (e) => {
  console.log('service worker: activated');
  e.waitUntil(
    caches.keys().then(cacheNames => {
      return Promise.all(
        cacheNames.map(cache => {
          if (cache !== cacheName) {
            // si on change le nom du cache permet de supprimer les anciens
            console.log('Service Worker : clearing old cache');
            return caches.delete(cache);
          }
        })
      );
    })
  );
});
```

Pour éviter



# Cache.keys

La méthode `keys()` de l'interface `Cache` retourne une `Promise` qui est résolue en un tableau de clé de `Cache`.

Les requêtes sont retournées dans le même ordre que l'ordre d'insertion.

Syntaxe :

```
cache.keys(request,{options}).then(function(response) {  
    //do something with your response array  
});
```

Exemple :

```
caches.open('v1').then(function(cache) {  
    cache.keys().then(function(response) {  
        response.forEach(function(element, index, array) {  
            cache.delete(element);  
        });  
    });  
});
```

# Vétification de mode offline

**Est-ce que dans cet état l'application peut fonctionner en mode offline. Pour ce faire, il vous suffit de vous remettre sur la console dans application/service worker et de passer en mode offline. Rechargez la page ??**

**Que se passe t- il ?**

# Résultat



## Aucun accès à Internet

Voici quelques conseils :

- Vérifiez les câbles réseau, le modem et le routeur.
- Reconnectez-vous au réseau Wi-Fi

ERR\_INTERNET\_DISCONNECTED

The screenshot shows a web browser window with the address bar displaying `127.0.0.1:8081`. The page content displays the error message "Aucun accès à Internet" (No internet access) along with a dinosaur icon and a list of troubleshooting tips. The DevTools application panel is open, showing the "Service Workers" section. It lists a service worker for `http://127.0.0.1:8081/` with source `sw.js` and 3 errors. The status is "#281 activated and is running". The console at the bottom shows the error `ERR_INTERNET_DISCONNECTED`.

????

**Il nous manque quoi ....**

**==> il faut compléter avec l'événement fetch  
notre service worker pour lui indiquer comment  
doit fonctionner la page ....**

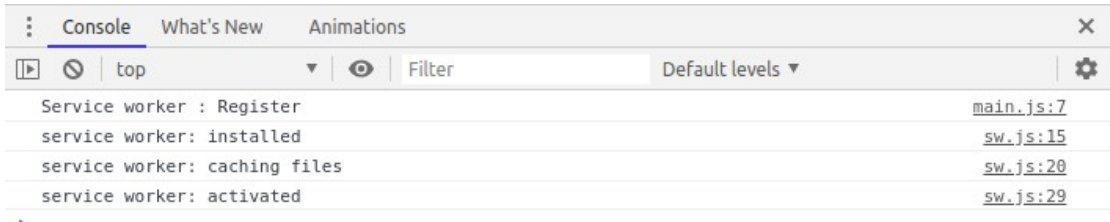
# Service worker -- Fetch

**On doit compléter et définir comment doit fonctionner l'ensemble ...**

```
// call Fetch event
self.addEventListener('fetch',e => {
  console.log('Service Worker: fetching');
  e.respondWith (
    // on va repondre avec les elements du cache
    fetch(e.request).catch(() => caches.match(e.request))
  )
})
```

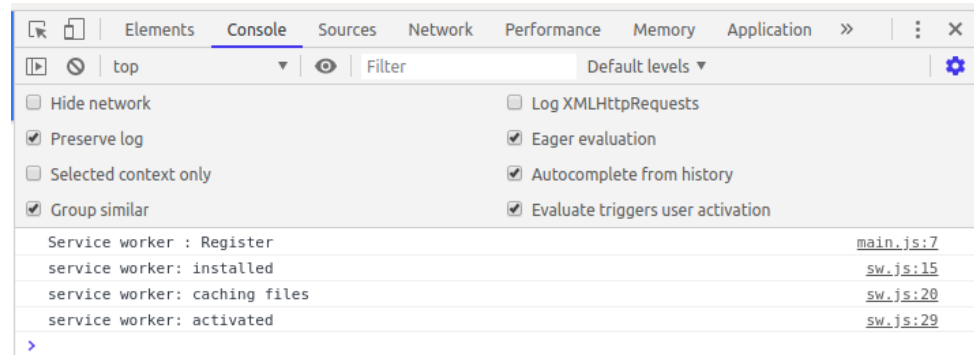
A chaque requête du navigateur dans l'application on va chercher dans le cache.

# Résultats



Service worker : Register [main.js:7](#)  
service worker: installed [sw.js:15](#)  
service worker: caching files [sw.js:20](#)  
service worker: activated [sw.js:29](#)

On voit pas la partie fetching



Hide network ☐ Log XMLHttpRequests ☐  
☒ Preserve log ☒ Eager evaluation  
☐ Selected context only ☒ Autocomplete from history  
☒ Group similar ☒ Evaluate triggers user activation



Service worker : Register [VM6 main.js:7](#)  
service worker: installed [sw.js:15](#)  
service worker: caching files [sw.js:20](#)  
service worker: activated [sw.js:29](#)  
**3** Service Worker: fetching [sw.js:48](#)  
Navigated to <http://127.0.0.1:8081/>  
Service worker : Register [main.js:7](#)

# Résultat Offline

The screenshot displays a web browser window with the address bar showing `127.0.0.1:8081`. The page title is "PSB Cours PWA Service worker". The main content area features a form with the label "Country Name:" and an input field containing "enter country name". Below the input are two blue buttons: "Get Image Name" and "Fetch Flag Image".

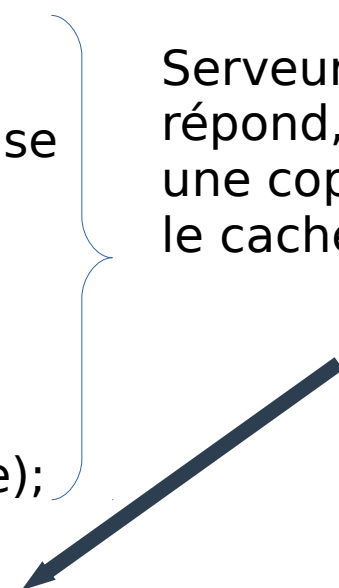
The browser's DevTools application is open, showing the "Application" panel. The left sidebar lists various components: Application (Manifest, Service Workers, Clear storage), Storage (Local Storage, Session Storage, IndexedDB, Web SQL, Cookies), Cache (Cache Storage, Application Cache), Background Services (Background Fetch, Background Sync, Notifications, Payment Handler, Push Messaging), and Frames (top).

The right pane of the "Application" panel shows the "Service Workers" section. It includes checkboxes for "Offline" (checked), "Update on reload" (checked), and "Bypass for network" (unchecked). Below this, a service worker is listed for `http://127.0.0.1:8081/` with links for "Update" and "Unregister". The "Source" is `sw.js` (8 versions), and it was "Received 22/03/2020 à 18:10:46". The status is "#296 activated and is running" with a "stop" link. The "Clients" list shows `http://127.0.0.1:8081/` with a "focus" link. There are "Push" and "Sync" buttons with associated input fields: "Test push message from DevTools" and "test-tag-from-devtools".



# Pour compléter le fetch

```
// call Fetch event
self.addEventListener('fetch',e => {
  console.log('Service Worker: fetching');
  e.respondWith (
    fetch(e.request)
    .then (res => {
      // faire une copie/clone de la reponse
      const resClone=res.clone();
      caches
      .open(cacheName)
      .then(cache => {
        // add response to cache
        cache.put(e.request, resClone);
      });
      return res;
    }).catch(err => caches.match(e.request).then(res => res))
  );
});
```



Serveur qui répond, on fait une copie dans le cache

Serveur qui ne répond pas on va chercher dans le cache répond

# Extension et expérimentation

**Pour tester sur votre infrastructure avec des postes différents du poste de développement, il est nécessaire de passer en mode HTTPS.**

## Générer les certificats

```
openssl genrsa -out key.pem  
openssl req -new -key key.pem -out csr.pem  
openssl x509 -req -days 9999 -in csr.pem -signkey key.pem -out cert.pem
```

A installer sur le répertoire racine de votre site

# Installation de nodejs en mode https

```
const https = require('https');
const fs = require('fs');
const express = require('express');
const app = express();

// This serves static files from the specified directory
app.use(express.static(__dirname));

https.createServer({
  key: fs.readFileSync('key.pem'),
  cert: fs.readFileSync('cert.pem')
}, app).listen(8081);
```

Pour y accéder à partir d'un autre poste ; vous pouvez taper les commandes

<https://192.168.1.58:8081/index.html>

# Problème de la validation des certificats auto-signés

Le problème est que les navigateurs pour les certificats auto-signés ne valide pas le service worker. Donc impossible de charger le cache et de tester notre service en mode offline.

Pour ce faire, il est nécessaire de démarrer le navigateur avec des paramètres spécifiques :

**#google-chrome --user-data-dir=/tmp/foo --ignore-certificate-errors --unsafely-treat-insecure-origin-as-secure=<https://192.168.1.58:8081>**

Une fois lancé, on peut vérifier que cette fois-ci les éléments service worker et le cache s'active correctement et lorsqu'on arrête le serveur et on peut quand même charger les pages à partir du cache.

# Pour finir avec le push-notification

[pwa-training-labs/push-notification-lab/app](https://pwa-training-labs/push-notification-lab/app)

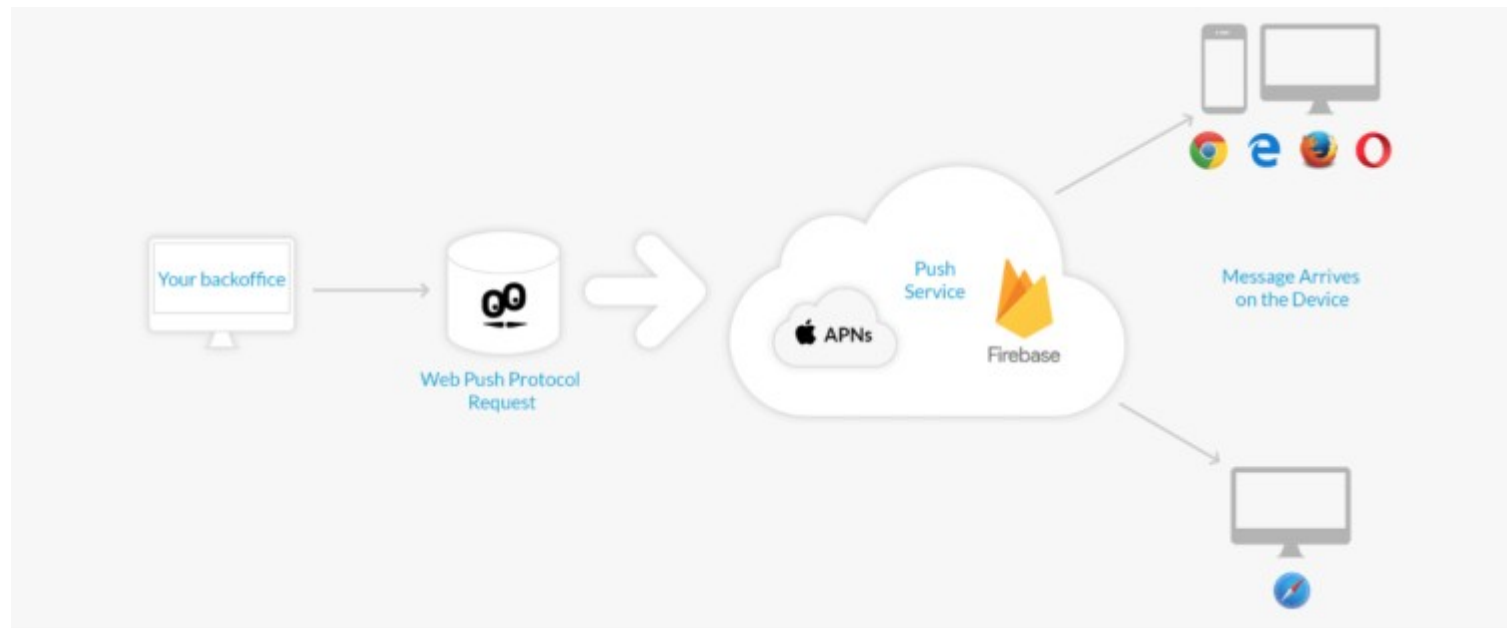
**L'API push notification permet de développer des mécanismes de réception et d'envoi de notifications à l'aide du SW.**

**Notification.requestPermission**

# Pour finir avec le push-notification

[pwa-training-labs/push-notification-lab/app](https://pwa-training-labs/push-notification-lab/app)

**On peut s'appuyer sur Firebase pour les notifications**



# **API Fetch : mise en oeuvre**

**</pwa-training-labs/fetch-api-lab/app>**

**Comment utiliser l'API Fetch pour demander des ressources**

**Comment construire des entêtes spécifiques en fonction des méthodes HTTP**

**Comment lire et construire des headers personnalisés**

**Usage et limitation CORS**

# API Fetch : mise en oeuvre

</pwa-training-labs/fetch-api-lab/app>

**L'API Fetch permet d'accéder à des ressources distantes. La méthode utilisée par défaut est un mode GET mais l'API peut demander également dans les autres méthodes HTTP. Permet également de gérer les problèmes multi-domaines CORS. L'API Fetch utilise également des promesses pour effectuer ses différentes demandes. Dans l'exercice proposé par Google plusieurs types de ressources vont être demandées aux serveurs :**

Ressources json

Ressources Image

Ressources texte

Ressources de header



# API Fetch : mise en oeuvre

## /pwa-training-labs/fetch-api-lab/app


### Fetch API Lab

Fetch JSONFetch imageFetch textHEAD request

Name:

Message:

POST request



Fetch is a game usually played with a dog. An object, such as a stick or ball, is thrown a moderate distance away from the animal, and it is the animal's objective to grab and retrieve it. Many times, the owner of the animal will say "Fetch" to the animal before or after throwing the object. In rare instances, cats, especially younger cats, have been known to engage in fetch behavior.

# API Fetch : mise en oeuvre

</pwa-training-labs/fetch-api-lab/app>

## Demande de ressources JSON : Pour ce faire

```
function validateResponse(response) {  
  if (!response.ok) {  
    throw Error(response.statusText);  
  }  
  return response;  
}
```

```
function readResponseAsJSON(response) {  
  return response.json();  
}
```

```
// Fetch JSON -----  
function fetchJSON() {  
  fetch('examples/animals.json')  
    .then(validateResponse)  
    .then(readResponseAsJSON)  
    .then(logResult)  
    .catch(logError);  
}
```

On retrouve la promesse avec un ensemble de callbacks chaînés

# API Fetch : mise en oeuvre

</pwa-training-labs/fetch-api-lab/app>

## Demande de ressources BLOB :

### Pour ce faire

```
function showImage(responseAsBlob) {  
  const container = document.getElementById('img-container');  
  const imgElem = document.createElement('img');  
  container.appendChild(imgElem);  
  const imgUrl = URL.createObjectURL(responseAsBlob);  
  imgElem.src = imgUrl;  
}  
function readResponseAsBlob(response) {  
  return response.blob();  
}
```

On retrouve la promise avec un ensemble de callbacks chaînés

```
function fetchImage() {  
  fetch('examples/fetching.jpg')  
    .then(validateResponse)  
    .then(readResponseAsBlob)  
    .then(showImage)  
    .catch(logError);  
}
```

# API Fetch : mise en oeuvre

</pwa-training-labs/fetch-api-lab/app>

## Demande de ressources **TEXTE**:

### Pour ce faire

On retrouve la promise avec un ensemble de callbacks chaînés

```
function showText(responseAsText) {  
  const message = document.getElementById('message');  
  message.textContent = responseAsText;  
}  
function readResponseAsText(response){  
  return response.text();  
}  
  
function fetchText() {  
  // TODO  
  fetch('examples/words.txt')  
    .then(validateResponse)  
    .then(readResponseAsText)  
    .then(showText)  
    .catch(logError);  
}
```

# API Fetch : mise en oeuvre

</pwa-training-labs/fetch-api-lab/app>

## Envoi d'une demande en mode POST sur un autre serveur CORS

On retrouve la promise avec un ensemble de callbacks chaînés

```
/* NOTE: Never send unencrypted user credentials in
production! */
function postRequest() {
  fetch('http://localhost:5000/', {
    method: 'POST',
    body: 'name=david&message=hello'
  })
  .then(validateResponse)
  .then(readResponseAsText)
  .then(showText)
  .catch(logError);
}
```

# API Fetch : mise en oeuvre

</pwa-training-labs/fetch-api-lab/app>

```
const express = require('express');
const app = express();
const bodyParser = require('body-parser')
const multer = require('multer');
const port = 5000;

app.use((req, res, next) => {
  res.setHeader('Content-Type', 'text/plain')
  // enable CORS
  res.set('Access-Control-Allow-Origin', '*');
  res.set('Access-Control-Allow-Methods', 'POST, GET, OPTIONS');
  res.set('Access-Control-Allow-Headers', 'X-CUSTOM, Content-Type');
  next();
})

const upload = multer();
const formParser = upload.fields([]);
const jsonParser = bodyParser.json();
const textParser = bodyParser.text();
```

Autorisations  
nécessaires pour le  
CORS

# API Fetch : mise en oeuvre

</pwa-training-labs/fetch-api-lab/app>

```
app.post('/', [formParser, jsonParser, textParser], (req, res) => {
  res.write(JSON.stringify(req.headers, null, 2))
  res.write('\n\n')
  const contentType = req.get('content-type');
  if (contentType.includes('text/plain')) {
    res.write(req.body)
  }
  if (contentType.includes('application/json') ||
    contentType.includes('multipart/form-data')) {
    res.write(JSON.stringify(req.body, null, 2))
  }
  res.end()
});
```

```
const server = app.listen(port, () => {
  const host = server.address().address;
  const port = server.address().port;
  console.log('App listening at http://%s:%s', host, port);
});
```

# Conteneurisation

**Pour démarrer on va configurer une image docker contenant votre application et Node.js. Il vous suffit de vous appuyer sur la description du Dockerfile pour générer votre image.**

**La construction du Dockerfile va vous permettre de construire une image à partir d'une image de Node.js et d'ajouter votre applications par copie de fichiers.**



# Conteneurisation

```
# -----  
# CNRS 2018  
# -----  
#FROM node:10.12.0-alpine  
FROM node:alpine  
RUN apk add --no-cache curl  
# Create app directory  
WORKDIR /usr/src/app  
COPY package*.json ./  
  
RUN npm install  
  
# Bundle app source  
COPY . .  
ENV DISPLAY :0  
EXPOSE 3000  
CMD [ "npm", "start" ]
```

Commandes :

```
#docker build -t imagepsb .
```

```
#docker run -p 3000:3000 -d imagepsb
```