Solution 1 : sessionStorage

Le sessionStorage est une solution simple et efficace pour stocker l'access token côté client tout en évitant les risques de sécurité liés à l'utilisation de localStorage.

Mini POC

```
// 1 ) SessionStorage
sessionStorage.setItem('keycloakToken', token)
```

On pourra également récupérer le token avec sessionStorage.getItem et vérifier sa présence via un navigateur

Solution 2 : WebWorker

Mini POC

Dans le main.js

```
// 2 ) Web Worker
// Créer un nouveau worker
const keycloakWorker : Worker  = new Worker( scriptURL: 'WebWorker.js')
// Envoyer le jeton Keycloak au worker
keycloakWorker.postMessage( message: { type: 'SET_TOKEN', payload: token })
```

Créer un fichier WebWorker.js

```
let token : null  = null

// Écouter les messages du thread principal
new *
self.addEventListener( type: 'message',  listener: (event : MessageEvent<any> ) : void  => {
  const { type, payload } = event.data

  if (type === 'SET_TOKEN') {
    token = payload
    console.log('Le jeton Keycloak a été stocké dans le web worker.')
  } else if (type === 'GET_TOKEN') {
    // Envoyer le jeton au thread principal
    self.postMessage(token)
  }
})
```

Solution 3 : ServiceWorker

Mini POC

Dans le main.js :

```
// Enregistrer le Service Worker
if ('serviceWorker' in navigator) {
  navigator.serviceWorker.register( scriptURL: '/worker.js', options: {
    scope: './' })
}

// Stocker le token de Keycloak dans le cache du Service Worker
if ('serviceWorker' in navigator) {
  navigator.serviceWorker.ready.then(registration : ServiceWorkerRegistration => {
    registration.active.postMessage( message: {
      type: 'storeToken',
      token: token
    })
  })
}
```

Créer un fichier worker.js :

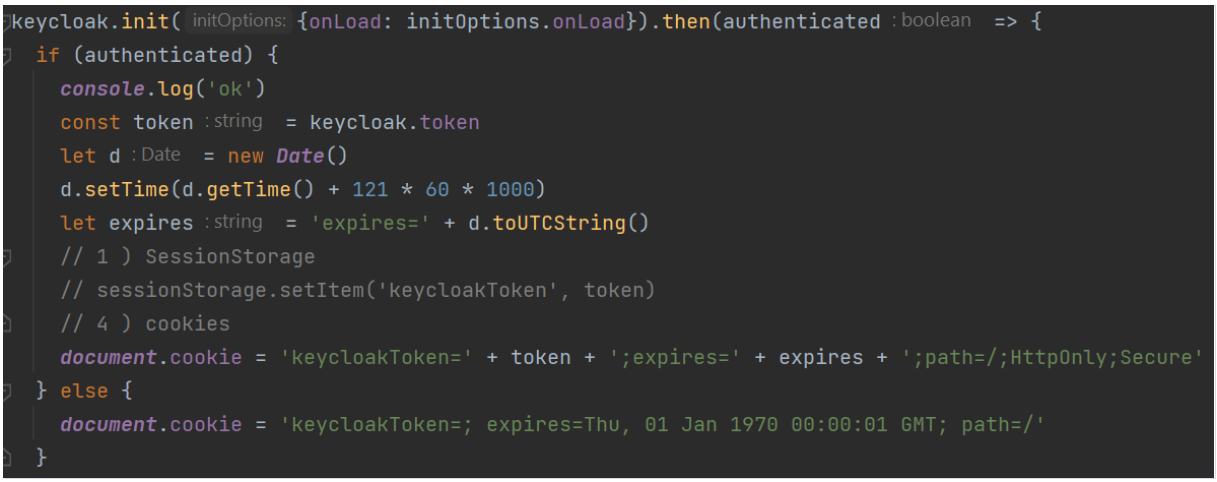
```
const cacheName : string = 'keycloak Token'

// Écouter l'événement "install" pour stocker les données dans le cache
new *
self.addEventListener( type: 'install', listener: event : Event => {
  event.waitUntil(
    caches.open(cacheName)
      .then(cache : Cache => cache.addAll( requests: [
        '/index.html',
        'src/main.js',
        'src/App.vue'
        // Ajouter ici les fichiers à stocker dans le cache
      ]))
  )
})
```

```
// Ecouter l'événement "fetch" pour récupérer les données depuis le cache
new *
self.addEventListener( type: 'fetch',  listener: event : Event  => {
  event.respondWith(
    caches.match(event.request)
      .then(response : Response | undefined  => {
        if (response) {
          return response
        }
        return fetch(event.request)
      })
  )
})
// Écouter le message envoyé depuis la page
new *
self.addEventListener( type: 'message',  listener: event : MessageEvent<any>  => {
  if (event.data && event.data.type === 'storeToken') {
    const token = event.data.token
    caches.open(cacheName).then(cache : Cache  => {
      cache.put( request: '/token', new Response(token))
    })
  }
})
```

Solution 4 : cookie de session

Les cookies de session sont des cookies qui expirent automatiquement à la fermeture du navigateur.

Mini POC



```
keycloak.init( initOptions: {onLoad: initOptions.onLoad}).then(authenticated : boolean  => {
  if (authenticated) {
    console.log('ok')
    const token : string  = keycloak.token
    let d : Date  = new Date()
    d.setTime(d.getTime() + 121 * 60 * 1000)
    let expires : string  = 'expires=' + d.toUTCString()
    // 1 ) SessionStorage
    // sessionStorage.setItem('keycloakToken', token)
    // 4 ) cookies
    document.cookie = 'keycloakToken=' + token + ';expires=' + expires + ';path=/;HttpOnly;Secure'
  } else {
    document.cookie = 'keycloakToken=; expires=Thu, 01 Jan 1970 00:00:01 GMT; path=/'
  }
```

Si l'utilisateur est bien authentifié, on récupère la valeur du keycloak Token, qu'on va insérer dans les cookies de notre navigateur avec une durée de validité à définir, l'attribut HttpOnly permettra d'empêcher d'accéder aux cookies en Javascript : si malgré cette protection, un attaquant venait à injecter du Javascript, les cookies ne seront pas accessibles, ce qui limitera la portée de l'attaque. On ajoute également l'attribut Secure ce qui limitera l'utilisation du cookie à des canaux dits « sécurisés » (ou « sécurisé » est défini par le navigateur web).

Si l'utilisateur n'est plus authentifié, on modifie la valeur de « expires » à une date ultérieure à celle du jour ce qui permettra de supprimer du navigateur le cookie.

On définit également l'attribut SameSite sur Strict pour que le cookie soit envoyé seulement si le site correspond au site actuellement affiché dans la barre d'url ce qui empêchera d'y avoir accès depuis un autre domaine.

Tableau comparatif des niveaux de vulnérabilité contre les attaques XSS et CSRF :

Solution	XSS	CSRF
Service Worker	Faible	Faible
Web Worker	Faible	Faible
Cookies	Moyen	Moyen
Session Storage	Élevé	Faible

Explications :

Service Worker et Web Worker: Ces deux technologies sont utilisées pour exécuter du code Js en arrière-plan, elles n'ont pas accès au thread principal, donc leurs vulnérabilités est plus faibles.

Cookies : Les cookies stockent des données dans le navigateur, ce qui les rend vulnérables aux attaques XSS car les cookies sont souvent utilisés pour stocker des informations d'identification. Les attaques CSRF sont également possibles car les cookies sont automatiquement envoyés avec chaque requête http mais si les cookies sont correctement configurés avec les options « httpOnly », Secure et SameSite=strict cela réduit considérablement la vulnérabilité des cookies aux attaques XSS et CSRF.

Session Storage : Le stockage de session est similaire aux cookies car il stocke des données dans le navigateur mais sans pouvoir ajouter une mesure de protection simple pour réduire les risques. Un attaquant pourra injecter du code Js sur la page pour accèder aux données contenue dans le session storage, ce qui rend sa vulnérabilité contre les attaques XSS élevé, les attaques CSRF représentent un moins gros risque avec les sessions storages car elle ne sont pas envoyé à chaque requête http.

Solution	**Stockage de l'access token**	**Sécurité**	**Multi-onglets**
sessionStorage	Dans la mémoire du navigateur	Vulnérable aux attaques XSS	Non
WebWorker	Dans un thread distinct du navigateur	Plus sécurisé contre les attaques XSS et CSRF	Oui
ServiceWorker	Dans un thread distinct du navigateur	Plus sécurisé contre les attaques XSS et CSRF	Oui
Cookie de session	Dans un cookie stocké sur le navigateur	Sécurisé contre les attaques XSS et CSRF	Oui

Conclusion

Chacune des solutions offre ses propres avantages et inconvénients. Cependant la solution la plus adaptée serait d'utiliser un ServiceWorker ou un WebWorker pour stocker l'access token, car cela permettrait de garantir la conservation du token dans chaque onglet ouvert tout en évitant les risques de sécurité.

Code source du projet :

https://github.com/mohandsari/BookSpring