Activité - Moteur de recherche de jeux vidéo - Étape 5

Notre application est maintenant opérationnelle. Il ne nous reste plus qu'à en faire une « Progressive Web App ».

Mettre à jour le fichier `manifest.json`

Le fichier `manifest.json` est un composant clé pour les Progressive Web Apps (PWA). Il s'agit d'un fichier JSON qui contient des informations structurées sur l'application web. Ce fichier permet aux développeurs de spécifier comment l'application apparaît et se comporte sur les appareils des utilisateurs.

Caractéristiques Principales

- **name**: Le nom complet de l'application.

- **short$_{name}$**: Un nom plus court pour l'application, utilisé dans les contextes où l'espace est limité.
- **start$_{url}$**: L'URL de démarrage de l'application lorsqu'elle est lancée depuis l'écran d'accueil.
- **display**: Définit le mode d'affichage de l'application (exemple: plein écran, fenêtre standard).
- **background$_{color}$**: La couleur de fond utilisée lors du chargement de l'application.
- **description**: Une description concise de ce que fait l'application.
- **icons**: Des images de différentes tailles pour représenter visuellement l'application.

Ce fichier est essentiel pour convertir une application web classique en une PWA, offrant une expérience utilisateur plus proche de celle d'une application native. Il est habituellement inclus dans l'élément `` du document HTML.

. La bonne nouvelle est que `create-react-app` l'a déjà créé pour nous, et l'a même déjà référencé dans le fichier `index.html`. Vous pouvez trouver cela dans le dossier `/public` de votre application.

Modifier les entrées `name` et `short$_{name}$` pour qu'elles reflètent celles-ci:

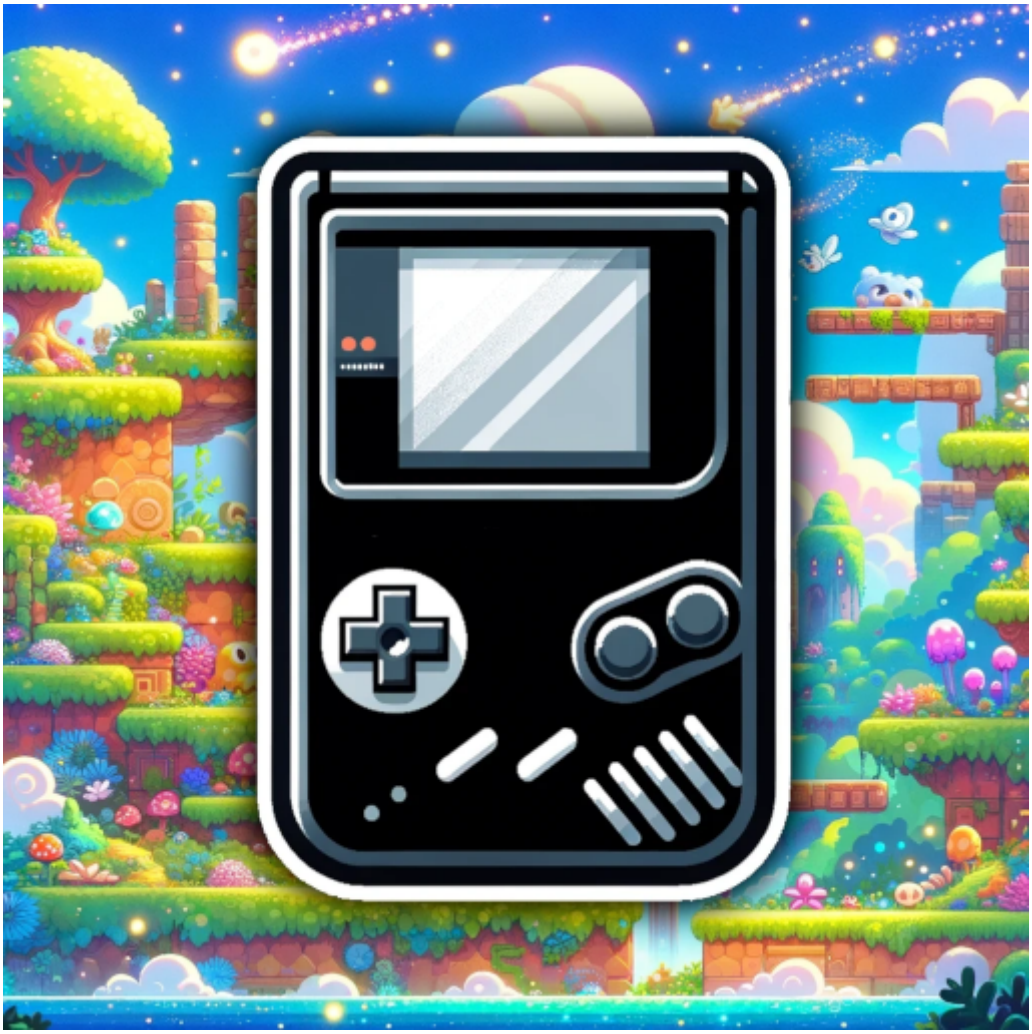
```
{
"short_name": "Jeux",
"name": "Annuaire des jeux",
"icons": [
{
"src": "favicon.ico",
"sizes": "64x64 32x32 24x24 16x16",
"type": "image/x-icon"
},
{
"src": "logo192.png",
"type": "image/png",
"sizes": "192x192"
},
{
"src": "logo512.png",
"type": "image/png",
"sizes": "512x512"
}
],
"start_url": ".",
"display": "standalone",
```

```
"theme_color": "#000000",
"background_color": "#ffffff"
}
```

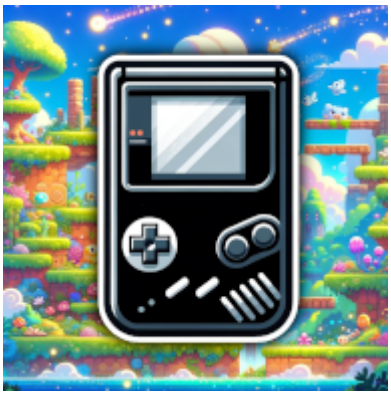
Le champ "display": "standalone" indique que l'application doit s'exécuter dans un mode autonome. Cela signifie que la PWA se lancera et fonctionnera comme une application native. En mode autonome, la PWA s'ouvre dans sa propre fenêtre, sans barre d'adresse ni autres éléments de l'interface utilisateur du navigateur. Cela permet à la PWA de ressembler à une application mobile classique.

Vous pouvez, si vous le souhaitez, modifier les fichiers icônes. Il suffit de remplacer ceux qui sont, par défaut, dans le répertoire `/public`. Cela est important pour une application finalisée que vous allez déployer. Pour cette activité, vous pouvez utiliser des images suivantes:

- {



  }
- {



}

- {https://formacitron.github.io/shopslist/favicon.ico}

Le service worker

Un service worker est un script qui fonctionne en arrière-plan de votre navigateur pour aider les sites web à fonctionner hors ligne et à charger plus vite. Il agit comme un intermédiaire entre le site web et l'internet, permettant de sauvegarder certaines informations pour que vous puissiez y accéder même sans connexion. C'est très utile pour rendre les sites web plus rapides et accessibles à tout moment.

C'est le service worker qui va faire en sorte que notre application soit installable sur un appareil, et qui lui permettra de démarrer quand il n'y a pas de connexion Internet.

Création d'un service worker

Les service workers présentent une certaine complexité et nécessitent un certain temps pour être maîtrisés. Heureusement, Google a développé une bibliothèque appelée Workbox qui simplifie grandement leur utilisation en masquant les aspects les plus complexes.

L'outil Create React App (CRA) intègre tout ce qu'il faut pour mettre en place un service worker facilement. En ajoutant un fichier nommé /src/service-worker.js, vous modifiez le processus de construction de l'application (via la commande npm run build), facilitant ainsi l'intégration des service workers.

On peut ensuite ajouter un code générique permettant la mise en cache des ressources de base de l'application:

```
import {precacheAndRoute} from 'workbox-precaching';
```

```
/* eslint-disable-next-line no-restricted-globals */
precacheAndRoute(self.__WB_MANIFEST);
```

ATTENTION: Le commentaire permet de désactiver des règles d'ESLINT. Il ne faut pas le supprimer sous peine de provoquer un plantage au moment de la construction (build).

La configuration Workbox montrée ici utilise la stratégie de mise en cache appelée "Precaching". Cette stratégie consiste à mettre en cache à l'avance les ressources statiques lors de l'installation du service worker. Cela signifie que dès que le service worker est installé, il met en cache les fichiers définis dans self.__WBMANIFEST, qui est généralement un ensemble de fichiers statiques générés automatiquement pendant la phase de build de votre application (comme les fichiers HTML, CSS, JavaScript, images, etc.).

La stratégie par défaut adoptée ici est donc la suivante :

Mise en cache à l'avance (Precaching) : Les ressources sont mises en cache dès que le service worker est installé. Cela garantit que toutes les ressources nécessaires sont disponibles hors ligne après la première visite de l'utilisateur.

Réponse du cache en premier (Cache First) : Lorsqu'une ressource est demandée (par exemple, lors du chargement d'une page ou d'une ressource statique comme une image ou un script), le service worker vérifie d'abord le cache. Si la ressource est trouvée dans le cache, elle est servie directement depuis là, ce qui permet un chargement rapide et réduit la dépendance au réseau.

Cette configuration est idéale pour les applications qui ont un ensemble de ressources statiques connues à l'avance et qui souhaitent offrir une expérience utilisateur rapide et fiable, en particulier dans les scénarios hors ligne ou sur des réseaux lents.

Enregistrement du service worker

Une fois le service worker créé, il faut faire en sorte que le navigateur Internet de vos utilisateurs le trouve. Vous pour réaliser cela dans le fichier `/src/index.js`, grâce au code suivant en bas:

```
if ('serviceWorker' in navigator) {
window.addEventListener('load', () => {
navigator.serviceWorker.register('/service-worker.js')
.then(registration => {
console.log('Service Worker registered: ', registration);
})
```

```
.catch(registrationError => {
console.log('Service Worker registration failed: ', registrationError);
});
});
}
```

Votre application est maintenant une Progressive Web App !

Tester votre application

Pour pouvoir tester votre service worker, vous avez besoin de « construire » votre application. Pour cela, il faut arrêter `npm start`, s'il est actif, puis lancer la commande: `npm run build`. Une fois l'action terminée, vous devriez pouvoir lancer `serve -s build`, puis consulter le résultat dans votre navigateur. En utilisant les outils pour développeur, vous devriez constater qu'un service worker est actif (dans l'onglet `Application` de Chrome ou de Firefox). À partir de ses outils, vous pouvez simuler l'absence de réseau, et constater que le rechargement de la page fonctionne toujours, et que les favoris restent consultables. Seul l'appel à l'API ne fonctionne pas. Vous pouvez modifier cette partie de l'application en ajoutant un message élégant.

> Il est possible que votre service worker soit rejeté pour des raisons de sécurité car une Progressive Web App doit être servie via une connexion sécurisée (https). Dans certaines situations, localhost est considéré comme sécurisé, mais la configuration de votre poste peut varier.

Rendre l'application installable

Une Progressive Web App qui répond aux critères nécessaires est installable (sous Android, et PC). Nous avons mis en place les éléments indispensables (manifest, service worker, icônes) pour que la nôtre soit éligible. Il nous reste à personnaliser le prompt d'installation.

Ajouter un prompt d'installation

Je vous propose de placer le code nécessaire dans notre composant `App`. Si nous ajoutons une partie d'interface graphique au-dessus du routeur, celle-ci sera alors présente sur toutes les pages.

Comme cette partie devra être cachée tant que l'installation n'est pas disponible (le navigateur décidera du bon moment), il nous faut prévoir une manière de la faire apparaitre au bon moment. Pour cela, nous allons utiliser un state, ainsi qu'un rendu conditionnel.

Commençons par ajouter un state dans notre composant `App`:

```
const [install, setInstall] = useState(false);
```

Puis transformons notre JSX pour ajouter un prompt d'installation. Nous prévoyons d'appeler une méthode `handleInstall` si l'utilisateur décide de cliquer sur le bouton `Installer`:

```
return (
<BookmarksContext.Provider value={{ bookmarks, setBookmarks }}>
{install && (
<div className="bg-gray-300 shadow-gray-700 p-4 flex items-center">
<div className='flex-grow text-center'>Voulez-vous installer
l'application sur votre appareil ?</div>
<button className="px-4 py-2 rounded text-white bg-teal-600" onClick=
{handleInstall}>Installer</button>
</div>
)}
<RouterProvider router={router}></RouterProvider>
</BookmarksContext.Provider>
);
```

Notre prompt ne sera affiché que si le state `install` devient `true`.

Attendre l'événement `beforeinstallprompt`

Lorsque le moment sera venu d'installer l'application, le navigateur déclenchera un événement `beforeinstallprompt`. À ce moment-là, nous pourrons rendre notre interface visible en changeant la valeur de `install`.

Voici le code que nous allons ajouter juste au dessus de notre JSX (return). Nous l'expliquerons juste après:

```
const deferredPrompt = useRef(null); // Utiliser useRef pour conserver
deferredPrompt à travers les re-rendus
```

```
useEffect(() => {
const handler = (e) => {
e.preventDefault();
deferredPrompt.current = e; // Stocker l'événement dans la propriété
.current de la ref
console.log("Change prompt", deferredPrompt)
setInstall(true);
};

// On place l'eventListener au démarrage
window.addEventListener('beforeinstallprompt', handler);

return () => {
// On retire l'eventListener à la fermeture
window.removeEventListener('beforeinstallprompt', handler);
};
}, []);

const handleInstall = () => {
deferredPrompt.current.prompt();
deferredPrompt.current.userChoice.then((choiceResult) => {
if (choiceResult.outcome === 'accepted') {
alert("Merci d'avoir installé l'application !")
} else {
console.log('L\'utilisateur a refusé l\'installation');
}
deferredPrompt.current = null;
});
setInstall(false);
}
```

Explication du Code pour Installer une PWA

1. Utilisation de `useRef` pour conserver `deferredPrompt`
    - `useRef(null)` crée une référence persistante à travers les re-rendus du composant.
    - Utilisé ici pour stocker l'événement `beforeinstallprompt`, permettant de contrôler l'invite d'installation.
2. Gestion de l'événement `beforeinstallprompt` avec `useEffect`

- `useEffect` s'exécute au montage et au démontage du composant, grâce à son tableau de dépendances vide `[]`.
- Assure que l'écouteur d'événements est proprement ajouté et retiré.
- Dans `useEffect`:
- Un gestionnaire pour `beforeinstallprompt` prévient l'affichage automatique de l'invite.
- `e.preventDefault()` empêche ce comportement par défaut.
- `deferredPrompt.current = e;` stocke l'événement pour utilisation future.
- `setInstall(true);` active l'affichage de l'interface utilisateur pour l'installation.

3. Fonction `handleInstall` pour déclencher l'installation

- `handleInstall` est appelé via une action utilisateur, typiquement un clic sur un bouton.
- `deferredPrompt.current.prompt();` affiche l'invite d'installation.
- `deferredPrompt.current.userChoice` gère la réponse de l'utilisateur.
- Affiche un message de remerciement si l'installation est acceptée.
- Réinitialise `deferredPrompt.current` à `null` après usage.
- `setInstall(false);` cache l'interface d'installation après interaction.

4. Conclusion

- Ce bloc de code permet de contrôler précisément quand proposer l'installation d'une PWA, offrant une expérience utilisateur améliorée et personnalisée autour de cette fonctionnalité.

> Votre Progressive Web App est maintenant installable !!!

L'événement `beforeinstallprompt` contient une méthode `prompt` qui déclenche l'action d'installation lorsqu'elle est appelée. Nous allons donc sauvegarder cet événement dans le state `install`. Sera nous permettra à la fois de le faire varier pour afficher notre message d'installation (il ne sera plus false), mais également de pouvoir lancer la méthode `prompt` quand l'utilisateur cliquera sur notre bouton `Installer`.

```
useEffect(() => {
const handler = (e) => {
```

```
e.preventDefault();
setInstall(e.prompt);
};

window.addEventListener('beforeinstallprompt', handler);

return () => {
window.removeEventListener('beforeinstallprompt', handler);
};
}, []);
```

5. Tester votre application