Activité - Moteur de recherche de jeux vidéo - Étape 3

Nous sommes maintenant en mesure de lister les favoris de l'utilisateur, mais ces données sont perdues à chaque fois que le navigateur est fermé. Il nous faut donc trouver une solution pour enregistrer ces informations.

Plusieurs solutions s'offrent à nous:

- Si nous avions une backend, nous pourrions utiliser ajax pour envoyer les données sur un serveur distant. Cette solution nécessiterait cependant de mettre en place un système d'inscription et d'authentification, afin d'associer les informations au bon utilisateur (et garantir la confidentialité des données).
- L'API IndexDB permet de sauvegarder des données dans le navigateur de l'utilisateur. Elle est complexe à utiliser, mais propose une limite de taille assez large (environ 2Go).
- Une solution plus simple, serait d'utiliser localStorage ({https://developer.mozilla.org/fr/docs/Web/API/Web_Storage_API}). Cette API permet de stocker des informations dans le navigateur de l'utilisateur, et elle est facile à utiliser. Elle possède une limite de taille d'environs 5Mo, mais cela reste acceptable pour notre sujet du moment : enregistrer une liste de jeux.

Nous allons utiliser cette dernière possibilité au cours de cette activité.

Notions de sérialisation et désérialisation en JavaScript avec JSON

La sérialisation et la désérialisation sont deux concepts fondamentaux dans le développement de logiciels, en particulier lorsqu'il s'agit de manipuler des données en JSON avec JavaScript. Ces opérations permettent respectivement de convertir des objets JavaScript en une chaîne JSON et de reconstruire l'objet JavaScript à partir de cette chaîne JSON.

Définition de JSON

JSON (JavaScript Object Notation) est un format de données textuelles léger, destiné à l'échange de données entre applications. Il est facile à lire et à écrire pour les humains, tout en étant facile à analyser et à générer par les machines. Basé sur la notation des objets de JavaScript, JSON peut représenter des données structurées comme des objets, des tableaux, des nombres, des chaînes de caractères, des booléens, et des valeurs nulles. Sa simplicité et sa facilité d'utilisation ont fait de JSON un format standard pour l'envoi de données sur le web et pour la configuration d'applications.

Sérialisation en JavaScript

La sérialisation, dans le contexte de JavaScript et JSON, implique la conversion d'objets JavaScript en une représentation de chaîne JSON. Cette opération est couramment utilisée pour envoyer des données d'un client web à un serveur ou pour sauvegarder des informations dans un format standardisé et facilement transférable.

```
const objet = {
nom: 'Dupont',
age: 30,
adresse: {
rue: '123 rue de Paris',
ville: 'Paris',
pays: 'France'
}
};
```

```
const chaineJSON = JSON.stringify(objet);
console.log(chaineJSON);
```

Dans cet exemple, `JSON.stringify(objet)` convertit l'objet JavaScript en une chaîne JSON. Le résultat est une représentation textuelle de l'objet, incluant toutes ses propriétés et valeurs.

Désérialisation en JavaScript

La désérialisation est le processus inverse de la sérialisation. Elle permet de prendre une chaîne JSON et de la convertir en un objet JavaScript. Cette opération est essentielle pour traiter des données reçues au format JSON, par exemple, lors de l'appel d'APIs web.

```
const chaineJSON = '{"nom":"Dupont","age":30,"adresse":{"rue":"123 rue de Paris","ville":"Paris","pays":"France"}}';

const objet = JSON.parse(chaineJSON);
console.log(objet);
```

`JSON.parse(chaineJSON)` analyse la chaîne JSON et reconstruit l'objet JavaScript correspondant. L'objet résultant peut ensuite être utilisé dans le code JavaScript comme n'importe quel autre objet.

Ces deux opérations, sérialisation et désérialisation, jouent un rôle crucial dans le développement web moderne, permettant une manipulation flexible et efficace des données entre différents systèmes et applications.

Le localStorage

Le `localStorage` est une fonctionnalité de stockage web qui permet aux sites web de stocker des données directement dans le navigateur de l'utilisateur. Contrairement aux cookies, les données stockées via `localStorage` ne sont pas envoyées au serveur à chaque requête HTTP. `localStorage` offre une capacité de stockage plus importante, typiquement jusqu'à 5MB ou plus, et les données y sont stockées de manière persistante, c'est-à-dire qu'elles restent disponibles après la fermeture et réouverture du navigateur.

`localStorage` est un système de stockage qui fonctionne sur le principe clé/valeur au sein du navigateur de l'utilisateur. Chaque donnée stockée dans `localStorage` est associée à une clé unique sous forme de chaîne de caractères, ce qui permet de récupérer, modifier ou supprimer

facilement la valeur correspondante. Les valeurs stockées doivent également être des chaînes de caractères. Cela signifie que pour sauvegarder des objets ou d'autres types de données structurées, il est nécessaire de les convertir en une chaîne de caractères, généralement au format JSON, avant de les stocker.

L'utilisation de `localStorage` est simple et directe. Voici comment on peut stocker et récupérer des données :

```
// Pour stocker des données : on sérialise et on stocke en utilisant une clé
localStorage.setItem('clé', 'valeur');

// Pour récupérer des données : on utilise la clé correspondante
const valeur = localStorage.getItem('clé');

// Pour supprimer des données spécifiques
localStorage.removeItem('clé');

// Pour effacer toutes les données stockées
localStorage.clear();
```

Une limitation importante du `localStorage` est qu'il ne peut stocker que des chaînes de caractères. Pour stocker des structures de données complexes comme des objets ou des tableaux, il est nécessaire de les convertir en une chaîne de caractères - un processus connu sous le nom de sérialisation. En JavaScript, cela est souvent réalisé en utilisant `JSON.stringify()` pour sérialiser les données avant de les stocker, et `JSON.parse()` pour les désérialiser après les avoir récupérées du `localStorage`.

```
// Sérialisation et stockage d'un objet
const utilisateur = { nom: 'Dupont', age: 30 };
localStorage.setItem('utilisateur', JSON.stringify(utilisateur));

// Récupération et désérialisation de l'objet
const utilisateurStocke = JSON.parse(localStorage.getItem('utilisateur'));
console.log(utilisateurStocke);
```

Cette technique permet d'utiliser `localStorage` pour stocker de manière persistante des données complexes entre les sessions de navigation, tout en contournant sa limitation de ne stocker que

des chaînes de caractères.

Création d'une fonction de sauvegarde

Lorsqu'on met à jour un `state`, l'interface est rafraichie. On peut donc utiliser `useEffect` pour détecter le moment où notre `state` `bookmarks` a été modifié, et sauvegarder nos données dans le localStorage.

Dans le composant `App`, ajoutons la logique suivante:

```
useEffect( () => {
localStorage.setItem("bookmarks", JSON.stringify(bookmarks));
}, [bookmarks])
```

Ici, `useEffect` est utilisé pour sauvegarder les données de `bookmarks` dans le stockage local du navigateur chaque fois que `bookmarks` change. Le tableau de dépendances `[bookmarks]` indique à React que l'effet doit s'exécuter chaque fois que la valeur de `bookmarks` est modifiée. Cela comprend le premier rendu du composant où `useEffect` s'exécute également, assurant que les données initiales de `bookmarks` sont sauvegardées dès le début.

La fonction de callback à l'intérieur de `useEffect` utilise `localStorage.setItem` pour stocker les données. `JSON.stringify(bookmarks)` convertit l'objet `bookmarks` en une chaîne JSON pour qu'il puisse être correctement stocké dans le localStorage, qui ne peut stocker que des chaînes de caractères.

En résumé, cet effet garantit que chaque fois que les bookmarks sont ajoutés, supprimés ou modifiés, la version la plus récente est automatiquement sauvegardée dans le stockage local du navigateur, permettant de persister les données entre les sessions de navigation.

> Utilisez les outils pour développeurs de votre navigateur pour consulter l'état du localStorage. La modification des bookmarks devrait le faire changer.

Chargement des données au démarrage

Lorsque useEffect est utilisé avec un tableau de dépendances contenant bookmarks, cela signifie que le code à l'intérieur de useEffect s'exécutera une fois au démarrage du composant, et ensuite

chaque fois que la valeur de bookmarks change. Cet usage est pratique pour réagir aux changements d'état spécifiques, comme sauvegarder les marque-pages dans le stockage local.

Utiliser useEffect avec un tableau vide en second argument permet d'exécuter le code une seule fois au démarrage du composant. Cela est utile pour les opérations d'initialisation qui ne doivent se produire qu'une seule fois, comme la récupération de données ou la configuration initiale.

Nous pouvons donc profiter de cette fonctionnalité pour récupérer nos données au démarrage.

```
useEffect( () => {
const savedBookmarks = JSON.parse(localStorage.getItem("bookmarks") || "
[]");
setBookmarks(savedBookmarks);
}, [])
```

L'expression || "$$$$" est utilisée dans le contexte de la lecture de données depuis le localStorage ou une source similaire, où il y a une possibilité que la valeur lue soit null ou undefined. Dans cet exemple, l'opérateur || (opérateur logique OR) sert de mécanisme de repli : si la valeur à gauche de l'opérateur est fausse (dans ce cas, si elle est null ou undefined), alors JavaScript évaluera et retournera la valeur à droite de l'opérateur, qui est "[]" ici.

L'utilisation de "$$$$" comme valeur de repli est courante lorsque l'on travaille avec des données qui sont censées être au format JSON d'un tableau. Si la source (par exemple, localStorage.getItem("quelqueChose")) ne retourne pas une valeur valide, "$$$$" fournit une chaîne vide qui représente un tableau vide après avoir été analysée avec JSON.parse(). Cela permet d'éviter les erreurs en s'assurant que le code travaille toujours avec un tableau, même si les données stockées sont absentes ou corrompues.

Gestion des conflits

Nous risquons de rencontrer un problème d'ordre si les données sont enregistrées avant d'être chargées. Nous allons donc mettre en place une vérification.

```
const [dataLoaded, setDataLoaded] = useState(false);

useEffect(() => {
if (dataLoaded) {
console.log("Sauvegarde:", bookmarks)
localStorage.setItem("bookmarks", JSON.stringify(bookmarks));
```

```
}
}, [bookmarks])

useEffect(() => {
const savedBookmarks = JSON.parse(localStorage.getItem("bookmarks") || "
[]");
setBookmarks(savedBookmarks);
console.log("Chargement:", savedBookmarks);
setDataLoaded(true);
}, [])
```

Dans ce contexte, dataLoaded est un état qui indique si les données ont été chargées depuis une source externe (par exemple, localStorage). Au début, dataLoaded est false, indiquant que les données n'ont pas encore été chargées. Lorsque le composant est monté, un effet (useEffect) est déclenché pour charger les données. Après le chargement et la mise à jour de l'état avec ces données, dataLoaded est mis à true pour signaler que le chargement est terminé.

L'utilisation de dataLoaded permet de contrôler le flux des opérations :

Avant le chargement : dataLoaded est false, empêchant certaines actions, comme la sauvegarde des données, qui ne devraient se produire qu'après le chargement complet. Après le chargement : dataLoaded devient true, autorisant des actions qui dépendent de la disponibilité des données chargées.

Cela aide à éviter des problèmes comme la sauvegarde de données non initialisées ou la réalisation d'opérations basées sur des données encore non disponibles.

> Vous savez maintenant sauvegarder et récupérer des données !