Partie 5: Stockage des données

Dans cette partie, nous allons nous concentrer sur comment stocker les données de notre app.

Modules JavaScript

Avant de plonger dans l'univers du stockage de la donnée, il faut se rendre à l'évidence que notre code actuel est un peu brouillon. Nous allons donc le structurer un peu pour qu'il soit plus maintenable et plus compréhensible.

Laissez-moi donc vous introduire les modules JS !

Comme dans beaucoup de choses en technique, l'explication est dans le nom !

En effet, un module JS et un bout de JS externe au programme principal que l'on va venir importer sur le programme principal pour qu'il puisse utiliser les fonctions du dit module. Cela va nous obliger à penser notre code en plus petits bouts et donc à le *modulariser*. Notre code sera donc éclaté en plus petits modules (à la dénomination bien pensée) pour que l'on puisse facilement l'utiliser, mais aussi réparer seulement la partie qui pourrait faire défaut (comme dans une voiture).

La mise en pratique de ces modules se déroule comme suit:

- Créez un nouveau fichier JS dans un nouveau dossier que vous placerez dans le dossier *dist*

```
//nouvelle architecture de dossier
- dist
    -assets
    -style
    -lib
        -easyDOMmanipulation.js (par exemple)
        -myFirstModule.js (par exemple)
    -index.html
    -main.js
```

- Copier-coller la fonction *DomOn* dans le fichier *easyDOMmanipulation*
- Exporter la fonction *DomOn*

POINT THÉORIQUE: Pour exporter l'élément (par exemple une fonction, mais ce peut aussi être une variable, un objet, etc.) que vous voulez utiliser dans votre programme principal il faut:

```
// dans mon module: myFirstModule.js
const myFunction = (a, b) => a + b;
export { myFunction };
```

```
//dans mon programme principal: main.js
//ATTENTION toujours à la première ligne du fichier
import { myFunction } from "./lib/myFirstModule.js";
//              |
//              |
//              > vous n'êtes pas obligé de reprendre le même nom que
//                celui que porte la fonction dans le module
//suite du code ...
//pour l'utiliser:
const test = myFunction(18, 32);
console.log(test); // output 50
```

Il y a aussi d'autre façon d'exporter ou d'importer les modules JS, la méthode montrée est la plus lisible et la plus courte, car elle exporte le tout en une seule fois. Voir les ressources pour d'autres façons de faire.

Ressources

Modules Doc MDN

Modules Doc WW3SCHOOL

Exercice pratique n°4 A

DONNÉE: le but de cet exercice est de modulariser votre code. Modifier donc votre code pour que ces 3 parties deviennent 3 modules JS à part entière.

- La fonction de manipulation du DOM
- La fonction de changement des dates
- La fonction de templating

ATTENTION: de ne pas oublier de mettre le *type:"module"* dans la balise *script* de votre HTML.

```
<script defer type="module" src="main.js"></script>
```

Indexed DB 1/2

Maintenant que nous avons vu comment rajouter des films à notre interface, nous allons faire une petite expérience:

Reloadez simplement la page des films.

Comme vous pouvez le voir, il nous reste pas mal de boulot...

Bienvenue donc dans le monde fascinant **des bases de données NoSQL côté client** !

NoSQL : Le NoSQL est une variante des bases de données dites SQL ou "traditionelles". Pour ne pas compliquer les choses, nous allons théoriser cela assez simplement. Cependant le NoSQL est un monde à part entière et si vous voulez en savoir plus, je vous y encourage => voir les ressources.

IndexedDB, la base de données que nous allons voir, est donc une base NoSQL de type "clé-valeur". Cela veut dire que les données sont stockées comme dans un objet JS. Une clé = une donnée. Pour rapatrier la valeur, on cherche avec sa clé.

Côté client: Comme dit dans le nom, IndexedDB est une base de données côté client. Cela veut simplement dire que les données sont stockées directement sur le browser du client et non pas une sur un serveur externe. Plus besoin de faire des requêtes par internet pour accéder aux données puisqu'elles sont déjà présentes.

Pour comprendre tout cela, un peu de pratique !

Ressources

NoSQL pour allez (beaucoup) plus loin

IndexeDB Doc MDN

Pour commencer

> Pour avoir un retour graphique sur ce qui se passe dans la base de données => clic droit sur la page web de la PWA => inspecter => application => indexedDB (dénommé plus tard GNU)

> Toutes les prochaines actions se font après les *import* mais avant tout le reste du code.

1. Commençons par ouvrir une connexion:

```
const request = window.indexedDB.open("cinetheque", 1); //nom de la base de donnée et sa version
let db;
//variable qui stockera l'instance de la base de donnée sur laquelle nous travaillerons
```

2. Il peut arriver 2 choses après avoir demandé à ouvrir une connexion => ça marche / ça ne marche pas

```
//ça marche pas
request.onerror = (event) => {
  console.log("Problems, user no allowence");
};

//ça marche
request.onsuccess = (event) => {
  db = event.target.result; //puisque ça marche on récupère l'instance de l'appel que l'on vient de faire.
  // on le store dans db => c'est notre accès à notre base.

  db.onerror = (event) => {
    // si il y a une erreur SUR L'INSTANCE ET NON SUR LA CONNECTION
    // (sur la connection cela c'est fait juste avant ) on la console.log()
    console.error("Database error: " + event.target.errorCode);
  };
};
// on peut déjà voir la base de donnée créée dans le GNU
```

3. Après que la base ait été créée, il faut la structurer

```
request.onupgradeneeded = (event) => {
  // seuleument ici que l'on peut changer la structure de notre database
  db = event.target.result; //on récupère toujours l'instance de notre appel.
  //on crée un objet = une table en DB relationelle)
  const objectStore = db.createObjectStore("movies", {
    keyPath: "numKey", // qui a comme clé unique "numKey"
    autoIncrement: true, // qui s'incrmente toute seule
  });

  // on créer des indexs HYPER IMPORTANT (voir point suivant)
  objectStore.createIndex("titleMovie", "titleMovie", { unique: false });
  objectStore.createIndex("watchingdate", "date", { unique: false });
  objectStore.createIndex("rate", "rate", { unique: false });
  objectStore.createIndex("storagedate", "storagedate", { unique: false });
};
```

Aux dernières lignes on crée des *indexs* cela va nous permettre de trier les données directement dans la base par rapport à certaines valeurs. Et c'est ça un des grands points forts d'IndexedDB.

```
objectStore.createIndex("watchingdate", "date", { unique: false });
//param1 (watchingdate) = nom de l'index
//param2 (date) = clé de la valeur à trié/indexé
//param3 = la possibilité d'avoir des doublons ou pas
```

4. Il reste à rentrer des données dans la base maintenant.
   Reprenons donc le bout de code que nous avions pour reprendre les données de notre formulaire. Nous allons maintenant vouloir les mettre dans notre DB pour qu'elles soient sauvegardées. Pour cela, nous allons avoir besoin d'un petit point théorique sur la gestion des images !

Gestion des images

Jusqu'à présent, nous avions simplement une image de remplissage que nous mettions à tous les films. Passons au stade suivant.

Pour stocker une image dans indexedDB il faut la transformer en **blob**

> Blob: **B**inary **L**arge **Ob**jects

Un blob est un objet binaire (donc pas un objet JS) qui ne peut pas changer (immuable). La force du blob est qu'il est lisible par le browser directement en tant qu'objet binaire donc on peut facilement le manipuler pour le transformer de "lisible pour la machine" à "lisible pour l'humain" du binaire au pixel !
Pour transformer notre image reçue de l'utilisateur en blob:

```
// on vient créer une url pour aller recherhe l'image dans les dossiers temporaire du système
const url = window.URL.createObjectURL(picture);
// on va cherhce l'image  (ne pas oublier de mettre async sur la fonction DomOn qu'on appelle)
const response = await fetch(url);
// on vient transformer le fichier en blob
//en soit le fichier est déjà transformer en blob à la création de l'url
//là on vient juste nettoyer "response" pour n'avoir que le blob
const blob = await response.blob();
```

Ressources

Blob Doc MDN

Nous avons donc maintenant le bon format (blob) pour sauvegarder notre image dans notre DB. Il faut maintenant savoir **comment** sauvegarder nos données dans la DB

Nous allons faire ce qui s'appelle une *transaction*.

ATTENTION: ici nous nous occupons que de la donnée et de sa sauvegarde et non pas de son display. Quelques lignes de codes sont donc à enlever.

```
db.transaction(["movies"], "readwrite").objectStore("movies").add(movie);
// 1. On fait une transaction sur l'objet (la table) movies,
//    ici on dit juste quels sont les objets qui seront impacter par la transaction,
// 2. On va lire mais surtout écrire dedans
// 3. Puis on va sauvegarder dans l'objet (la table) movies
//    (ici on dit quel objet va être modifier)
// 4. Pour finir on ajoute (add) l'onjets JS
```

Ce qui nous donne globalement pour la fonction remastérisée DomOn dont on avait déjà codé un bout avant:

```
domOn("#add-movie-button", "click", async () => {
  //récupération formulaire
  const movieTitle = document.querySelector("#title-movie").value;
  const viewingDate = document.querySelector("#date").value;
  const picture = document.querySelector("#picture").files[0];
  const stars = document.querySelector("#star").value;

  //Store it !
  //transform image as a blob
  const url = window.URL.createObjectURL(picture);
  const response = await fetch(url);
  const blob = await response.blob();

  const movie = {
    storagedate: new Date(Date.now()),
    title: movieTitle,
    date: new Date(viewingDate),
    picture: blob,
    rate: stars,
  };

  db.transaction(["movies"], "readwrite").objectStore("movies").add(movie);
});
```

Ce qui nous donne globalement pour la mise en place de la DB:

```
//IndexedDB
const request = window.indexedDB.open("cinetheque", 1);
let db;

//ça marche pas
request.onerror = (event) => {
  console.log("Problems, user no allowence");
};

//ça marche
request.onsuccess = (event) => {
  db = event.target.result;
  db.onerror = (event) => {
    console.error("Database error: " + event.target.errorCode);
  };
};

request.onupgradeneeded = (event) => {
  db = event.target.result;
  db.onerror = (event) => {
    console.error("Database error: " + event.target.errorCode);
  };
  const objectStore = db.createObjectStore("movies", {
    keyPath: "numKey",
    autoIncrement: true,
  });
  objectStore.createIndex("titleMovie", "titleMovie", { unique: false });
  objectStore.createIndex("watchingdate", "date", { unique: false });
  objectStore.createIndex("rate", "rate", { unique: false });
  objectStore.createIndex("storagedate", "storagedate", { unique: false });
};
```

IndexedDB 2/2

Nos données sont maintenant sauvegardées !

Manque plus qu'à les incorporer à notre HTML. Nous avons déjà vu le template donc il ne manque plus qu'à aller chercher les données stockées pour les mettre dans notre template.

Pour aller chercher une donnée dans la base, c'est comme ceci:

```
db.transaction("movies").objectStore("movies").getAll().onsuccess = (event) => {
  const myresults = event.target.result;
};
```

Exercice pratique n°4 B

DONNÉE: le but est de coder une fonction qui sera appelée en tout début de sccript et qui affichera le DOM des films en reprenant les données de la base.

ATTENTION: quand du DOM est injecté, par du JS par exemple, le reste du DOM reste inchangé, il ne se supprime pas tout seul 😄.

TIPS: vous pouvez maintenant ajouter la feuille de style CSS présente dans la solution de l'ExPratique4 pour donner un peu de panache à votre app, mais aussi pour essayer le challenge suivant.

CHALLENGE: pour les *stars* allez voir dans le CSS sous *.rating0:before* et rajouter ce code au *main.js*. En rajoutant la classe *rating0* à la balise HTML *span* qui comporte déjà la classe *.rating* vous obtenez d'un point de vue graphique des étoiles rouges. Maintenant, reste plus qu'à faire correspondre la donnée dans la DB et la bonne classe HTML à mettre dans le template.

```
const tabRating = [
  "rating0",
  "rating1",
  "rating2",
  "rating3",
  "rating4",
  "rating5",
];
//console.log(tabrating[2]) output rating2
```