Utiliser l’API de stockage IndexedDB en JavaScript

En plus de Web Storage, qui est apparu avec le HTML5, il existe une autre API qui va nous permettre de stocker des données côté client : l’API IndexedDB.

Présentation

L’API IndexedDB est une API de stockage de données côté client qui va être utilisée pour stocker des quantités importantes de données structurées.

La quantité de données qui va pouvoir être stockée est beaucoup plus grande que ce qu’on pourrait stocker avec Web Storage et cela rend donc IndexedDB plus puissante que Web Storage.

Les opérations effectuées par IndexedDB sont réalisées de manière asynchrone, et ceci afin de ne pas bloquer le reste de la page.

Notez également qu’IndexedDB respecte la politique de même origine, ce qui signifie qu’on pourra accéder aux données stockées pour le domaine courant uniquement.

En pratique, pour utiliser IndexedDB, on suivra le schéma suivant :

1. On ouvre une connexion à la base de données
2. On crée un objet de stockage ;
3. On initie une transaction ;
4. On effectue des requêtes ;
5. On crée des gestionnaires d’évènements liés au résultat de nos requêtes.

Ouverture de la connexion à la base de données

Pour travailler avec IndexedDB, nous allons avant tout vérifier que le navigateur supporte indexedDB

```
document.addEventListener("DOMContentLoaded", function(){
      if("indexedDB" in window) {
            console.log("YES!!! I CAN DO IT!!! WOOT!!!")
      }
},false)
```

puis ouvrir une base de données que l'on nommera « db » de version 1.

```
var openRequest = indexedDB.open("db", 1)
```

L'ouverture de la base de données peut déclencher quatre types d'événements différents :

- **success**
- **error**
- **upgradeneeded** : est utilisé à la fois lorsque l'utilisateur ouvre la base de données pour la première fois et lorsque vous modifiez la version.
- **blocked** : ne ce produit pas généralement, mais peut se déclencher si une connexion précédente n'a jamais été fermée.

La propriété IndexedDB utilisée, est une propriété du mixin WindowOrWorkerGlobalScope (implémenté par window). Cette propriété renvoie un objet IDBFactory.

L'interface IDBFactory fait partie de l'API IndexedDB et permet aux applications d'accéder à des bases de données de façon asynchrone. Cette interface nous fournit une méthode open() qui permet d'ouvrir une connexion à une base de données.

On va donc utiliser cette méthode open() avec notre objet (propriété) IndexedDB. La méthode open() prend en argument obligatoire le nom de la base de données qu'on souhaite ouvrir ainsi que la version de cette base de données en argument facultatif.

La méthode open() renvoie un objet IDBOpenRequest et effectue l'opération d'ouverture de la connexion à la base de données de manière asynchrone.

Si l'ouverture réussit, un évènement success est déclenché sur l'objet IDBOpenRequest renvoyé par open(). La propriété result de cet évènement aura alors comme valeur la valeur de l'objet IDBDatabase associé à la connexion.

Si l'ouverture de la connexion échoue, un évènement error est déclenché sur l'objet IDBOpenRequest renvoyé par open().

La version de la base de données détermine son organisation et notamment les objets stockés et leur structure. Par défaut, le numéro de version retenu est 1.

Si le numéro de version de la base de données qu'on souhaite ouvrir est inférieur au numéro fourni à open(), un évènement upgradeneeded est déclenché pour nous permettre de mettre à jour la base de données. Si la mise à jour se passe bien, un évènement success est déclenché.

Ici, on va créer trois gestionnaires qui vont gérer les évènements success, error et upgradeneeded :

```
var idbSupported = false
var db

document.addEventListener("DOMContentLoaded", function(){
    // Vérifie que IndexedDB est supporté
    if("indexedDB" in window) {
        console.log("YES!!! I CAN DO IT!!! WOOT!!!");
        idbSupported = true
    }

    if(idbSupported) {
        // Ouverture de la base de données « db »
        var openRequest = indexedDB.open("db", 1)
        // Mise à jour si version inférieure
        openRequest.onupgradeneeded = function(e) {
            console.log("Upgrading...")
        }
        // En cas de succès
        openRequest.onsuccess = function(e) {
            console.log("Success!")
            db = e.target.result
        }
        // En cas d'erreur
        openRequest.onerror = function(e) {
            console.log("Error")
            console.dir(e)
        }
    }
},false)
```

Lorsqu'on crée une nouvelle base de données ou si on met à jour la version de notre base de données, on doit créer les nouveaux objets de stockage pour cette version de la base dans le gestionnaire de upgradeneeded. Les objets créés dans la version précédente seront automatiquement disponibles ; il est inutile de les copier.

De plus, si on essaie de créer un objet de stockage avec un nom déjà existant (ou si on essaie de supprimer un objet de stockage avec un nom qui n'existe pas encore), une erreur sera renvoyée.

Notez que si l'évènement upgradeneeded quitte avec succès, l'évènement success de la requête d'ouverture de la base de données sera déclenchée.

Dans le cas où l'évènement success est déclenché (cas où la connexion s'est effectuée avec succès), openRequest.result est une instance de IDBDatabase et va donc représenter notre connexion.

Création d'un objet de stockage ou « object store »

Les objets de stockage vont stocker les données. Si vous connaissez un petit peu le fonctionnement des bases de données MySQL ou autres, vous pouvez considérer que nos objets de stockage vont être l'équivalent des tables.

Une base de données peut avoir plusieurs objets de stockage et ces objets de stockage peuvent stocker quasiment toutes formes de données. Ces objets de stockage peuvent stocker plusieurs valeurs, et chaque valeur doit être associée à une clef unique au sein d'un objet de stockage.

On va pouvoir passer la clef manuellement en même temps qu'on ajoute une valeur dans l'objet de stockage (ce qui peut être pratique dans le cas où on stocke une valeur primitive) ou définir une propriété qui servira de clef dans le cas où on stocke des objets. On peut également demander à ce que les clefs soient générées automatiquement.

La création ou la modification des objets de stockage va toujours se faire lors de la mise à jour de la version de la base de données, c'est-à-dire au sein du gestionnaire d'évènements `upgradeneeded`.
Pour créer un objet de stockage, on va utiliser la méthode `createObjectStore()`. Cette méthode prend le nom de l'objet de stockage en premier argument ainsi qu'un objet (facultatif) en second argument qui va nous permettre de définir une clef et renvoie un objet appartenant à l'interface `IDBObjectStore`.

```
 if(idbSupported) {
      var openRequest = indexedDB.open("db", 1);

      openRequest.onupgradeneeded = function(e) {
            console.log("Upgrading...")
            db = openRequest.result
            // Si la table 'store' n'existe pas elle sera créée. Elle aura une
clef primaire 'id'
            if (!db.objectStoreNames.contains('store')){
                  db.createObjectStore('store', {keyPath: 'id'});
            }
      }

      …
```

Pour définir une clef, on va utiliser l'une des propriétés `keyPath` ou `autoIncrement` de cette interface.

La propriété `keyPath` nous permet de définir une propriété qu'IndexedDB utilisera comme clef.
La propriété autoIncrement prend une valeur booléenne. Si la valeur passée est `true`, alors la clef pour chaque objet stocké sera générée automatiquement, en s'incrémentant à chaque fois.

```
upgradeDb.createObjectStore(products, {keyPath: 'id'});
```

L'exemple ci-dessus, va créer un objet de stockage 'products' et qui aura comme clé 'id'.

```
upgradeDb.createObjectStore('notes', {autoIncrement:true})
```

Ici, création d'une clef auto-incrémente.

```
upgradeDb.createObjectStore('products', {keyPath: 'id', autoIncrement:true})
```

La clef id sera auto-incrémente.

On peut également définir un index :

```
objectStore.createIndex('indexName', 'property', options)
```

```
var notesOS = upgradeDb.createObjectStore('notes', {autoIncrement: true})
notesOS.createIndex('title', 'title', {unique: false})
```

Initiation d'une transaction

On appelle « transaction » un groupe d'opérations dont le destin est lié. L'idée principale à retenir à propos des transactions est la suivante :

Les différentes opérations doivent toutes réussir indépendamment pour que la transaction soit un succès. Si une opération échoue, alors la transaction et donc l'ensemble des opérations échouent.

Dans notre contexte, les transactions vont s'effectuer à partir de l'objet symbolisant la connexion à la base de données (notre instance de IDBDatabase). Pour démarrer une nouvelle transaction, nous allons utiliser la méthode transaction() à partir de cet objet.

Cette méthode va prendre deux arguments :

- La liste d'objets de stockage que la transaction va traiter (obligatoire)

- Le type ou le mode de transaction souhaité (facultatif).

On peut choisir parmi trois modes de transaction :
- readonly (lecture seule),
- readwrite (lecture et écriture)
- versionchange (changement de version).

Ces modes vont définir quelles manipulations que l'on va pouvoir effectuer sur les données. Par défaut, le mode est readonly.

Pour lire les enregistrements d'un objet de stockage existant, la transaction peut être en mode readonly ou readwrite.

Pour appliquer des changements à un objet de stockage existant, la transaction doit être en mode readwrite.

Pour changer la structure de la base de données (le schéma), ce qui implique de créer ou supprimer des objets de stockage ou des index, la transaction doit être en mode versionchange.

```
// Création d'une transaction de la table 'store' en mode lecture/écriture
// ce mode nous permettra d'ajouter ou de modifier des données
let transaction = db.transaction('store', 'readwrite');

transaction.oncomplete = function(){
    console.log('Transaction terminée');
};
```

Création de requêtes et gestion des résultats

IndexedDB nous permet d'ajouter, de supprimer, de récupérer ou de mettre à jour des données dans notre base de données.

En pratique, pour effectuer ces manipulations, on commencera par créer une transaction puis on récupérera l'objet de stockage de celle-ci.

Ensuite, on va effectuer des requêtes (ajout de données, suppression, etc.) à partir de cet objet IDBObjectStore et on va finalement gérer les cas de succès ou d'erreur liés au résultat de nos requêtes.
L'interface IDBObjectStore nous fournit les différentes méthodes qui vont nous permettre de manipuler nos objets de stockage et notamment :

- Les méthodes put() et add() pour stocker des données dans la base.
- Les méthodes get() et getAll() pour récupérer les données depuis la base.
- Les méthodes delete() et clear() pour supprimer des données.

Pour stocker une nouvelle valeur dans un objet de stockage, par exemple, on pourra écrire un script comme celui-ci :

```
function ajouter(id, nom, prix) {
    // transaction
    let transaction = db.transaction('store', 'readwrite');

    transaction.oncomplete = function(){
        console.log('Transaction terminée');
    };

    let store = transaction.objectStore('store');
    // Element à ajouter
    let item = {
        id: id,
        title: nom,
        price: prix,
        created: new Date()
    };
    // Ajouter l'élément dans 'store'
    let ajout = store.add(item)

    ajout.onsuccess = function(){
        console.log('Produit ajouté avec la clef ' + ajout.result)
    }

    ajout.onerror = function(){
        console.log('Erreur : ' + ajout.error)
    }
```

```
    }
```

On commence donc par initier une transaction à partir de notre objet représentant la connexion à notre base de données (objet appartenant à IDBDatabase).

Notre objet let transaction appartient à IDBTransaction. Cette interface possède une méthode objectStore() qui renvoie un objet IDBObjectStore.

La ligne transaction.objectStore() nous permet d'accéder à notre objet de stockage afin d'effectuer des opérations avec celui-ci. On place le résultat dans une variable qui est un objet IDBObjectStore.

On utilise la méthode add() de l'interface IDBObjectStore qui permet de stocker de nouvelles valeurs dans un objet de stockage. Cette méthode prend une valeur en argument obligatoire et une clef en argument facultatif (la clef est fournie automatiquement seulement si l'objet de stockage ne possède pas d'option keypath ou autoIncrement).

> Pour information, la différence entre les méthodes put() et add() est la suivante :
> si on fournit une clef qui existe déjà pour une valeur à put(), la clef sera modifiée tandis qu'avec add() la requête échouera et une erreur sera générée

On effectue donc ici la requête suivante : « ajoute une nouvelle valeur dans notre objet de stockage ». Nous n'avons alors plus qu'à mettre en place les gestionnaires d'évènements de succès et d'erreur pour cette requête.

Notre objet let request appartient ici à l'interface IDBRequest. Cette interface dispose d'une propriété result qui contient le résultat d'une requête.

Lorsqu'on l'utilise avec une requête de type add(), la valeur de resquest.result est la clef de la valeur qui vient d'être ajoutée.

Cette interface contient également une propriété error qui indique le code de l'erreur survenue durant le traitement de la requête.

On va également pouvoir de manière similaire récupérer des données dans la base ou en supprimer.

Pour récupérer une donnée en particulier, on pourra par exemple utiliser la méthode get(). Cette méthode prend la clef de la valeur qu'on souhaite récupérer en argument.

```
    let lire = store.get(id)
    lire.onsuccess = function(){
```

```
            console.log('Nom du produit ' + lire.result.id + ' : ' + lire.resul
t.title)
        }

        lire.onerror = function(){
            console.log('Erreur : ' + lire.error)
        }
```

Pour supprimer une donnée, on utilisera la fonction delete()

```
        let supprimer = store.delete(id)
```

IndexedDB prend en charge ce qu'on appelle un curseur (cursor). Un curseur vous permet de parcourir les données. Vous pouvez créer des curseurs avec une plage facultative (un filtre de base) et une direction.

A titre d'exemple, le bloc de code suivant ouvre un curseur pour récupérer toutes les données d'un store object. Comme tout ce que nous avons fait avec les données, c'est asynchrone et dans une transaction :

```
        let transaction = db.transaction('store', 'readwrite');

        const objectStore = transaction.objectStore('store');

        objectStore.openCursor().onsuccess = function(event) {
            var cursor = event.target.result;
            if (cursor) {
                console.log('id: ' + cursor.value.id)
                // va à l'élément suivant
                cursor.continue();
            }

        }
```

Les curseurs nous permettrons de mettre à jour des données également.

```
                let transaction = db.transaction('store', 'readwrite');

                const objectStore = transaction.objectStore('store');

                objectStore.openCursor().onsuccess = function(event) {
                    var cursor = event.target.result;
                    if (cursor) {
                        console.log('id: ' + cursor.value.id)
                        if (cursor.value.id == index) {
                            const updateData = cursor.value;

                            updateData.title = titre;
```

```
                    updateData.price = prix;
                    const request = cursor.update(updateData);
                    request.onsuccess = function() {
                        console.log('Produit MAJ')

                    }
                }
                cursor.continue();
            }

        }
```

Les enregistrements peuvent également être récupérés grâce à des intervalles de clefs.

Un intervalle de clef peut être une seule valeur ou un intervalle avec des bornes inférieure et supérieure. Si l'intervalle possède ces deux bornes, il est dit borné. S'il n'a aucune borne, il est non-borné.

Un intervalle de clef bornée peut être ouvert (les bornes sont exclues) ou fermé (les bornes sont inclues).

Pour récupérer les différentes clefs d'un intervalle donné, on peut utiliser les fonctions suivantes :

Intervalle	**Code**
Toutes les clefs ≤ **x**	IDBKeyRange.upperBound(**x**)
Toutes les clefs < **x**	IDBKeyRange.upperBound(**x**, true)
Toutes les clefs ≥ **y**	IDBKeyRange.lowerBound(**y**)
Toutes les clefs > **y**	IDBKeyRange.lowerBound(**y**, true)
Toutes les clefs ≥ **x** && ≤ **y**	IDBKeyRange.bound(**x**, **y**)
Toutes les clefs > **x** &&< **y**	IDBKeyRange.bound(**x**, **y**, true, true)
Toutes les clefs > **x** && ≤ **y**	IDBKeyRange.bound(**x**, **y**, true, false)
Toutes les clefs ≥ **x** &&< **y**	IDBKeyRange.bound(**x**, **y**, false, true)
La clef = **z**	IDBKeyRange.only(**z**)

Reprenons le script précédent et adaptons le afin qu'il n'affiche que les clefs dont la valeur est égale à 'index':

```
                Let keyRangeValue = IDBKeyRange.only(index)
                let transaction = db.transaction('store', 'readwrite');
                const objectStore = transaction.objectStore('store');

                objectStore.openCursor(keyRangeValue).onsuccess = function(eve
nt) {
                    var cursor = event.target.result;
                    if (cursor) {
                         const updateData = cursor.value;

                         updateData.title = titre;
                         updateData.price = prix;
                         const request = cursor.update(updateData);
                         request.onsuccess = function() {
                               console.log('Produit MAJ')

                         }
                         cursor.continue();
                    }

                }
```

La clef étant unique, seul un enregistrement sera trouvé ! Pouvez-vous améliorer le script ?

En résumé

L’API IndexedDb permet de stocker des quantités importantes de données structurées dans le navigateur de vos visiteurs.

Ces API fonctionne principalement de manière asynchrone et adhère au principe de « same-origin policy » (politique de même origine).

IndexedDB est une API orienté objet : les données vont être stockées dans des objets de stockage ou « object store ». Les données sont stockées sous la forme de paires clef / valeur. Les valeurs peuvent êtres des objets structurés, et les clés peuvent être des propriétés de ces objets.

Cette API est construite autour d’un modèle de base de données transactionnelles : les différentes manipulations vont s’effectuer dans un contexte de transaction.

Durant ces transactions, on va effectuer des requêtes pour manipuler nos données. Ces requêtes sont des objets qui reçoivent les événements DOM de succès ou d’échec.