**IndexedDB**

Contenido

[**1.** **Introducción** 1](#_Toc188360456)

[**2.** **Apertura de una base de datos, “open”** 3](#_Toc188360457)

[**3.** **Almacenar objetos en IndexedDB** 6](#_Toc188360458)

[**4.** **Las transacciones en IndexedDB** 20](#_Toc188360459)

[**5.** **Autocommit** 24](#_Toc188360460)

[**6.** **Manejo de errores en IndexedDB** 28](#_Toc188360461)

[**7.** **Búsquedas en un almacén de objetos** 35](#_Toc188360462)

[**8.** **Borrar valores de un almacén de objetos** 40](#_Toc188360463)

[**9.** **Los cursores en IndexedDB** 44](#_Toc188360464)

1. **Introducción**

**IndexedDB** es una API poderosa que permite a las aplicaciones web trabajar con grandes volúmenes de datos y realizar operaciones avanzadas, como búsquedas, filtros y actualizaciones, todo desde el lado del cliente, sin necesidad de estar conectados a Internet. Si estás desarrollando aplicaciones que requieren un manejo robusto de datos en el navegador, **IndexedDB** será tu mejor aliada.

**Conceptos principales**

1. **Conceptos fundamentales:**
   * ¿Qué es IndexedDB y por qué usarla en tus proyectos?
   * Diferencias clave con otras tecnologías de almacenamiento web (como localStorage y sessionStorage).
   * Modelos de datos basados en objetos.
2. **Configuración y primeros pasos:**
   * Crear una base de datos y gestionar su versión.
   * Entender cómo funcionan los almacenes de objetos (object stores).
   * Uso de índices para optimizar búsquedas.
3. **Operaciones CRUD (Crear, Leer, Actualizar, Eliminar):**
   * Cómo agregar datos a la base.
   * Leer y consultar información utilizando claves e índices.
   * Actualizar y eliminar registros de forma eficiente.
4. **Características avanzadas:**
   * Transacciones: Cómo garantizar operaciones consistentes y seguras.
   * Manejo de errores y mejores prácticas.
   * Integración con aplicaciones offline-first.
5. **Casos prácticos y proyectos:**
   * Crear un sistema de gestión de tareas.
   * Implementar una galería de imágenes offline.
   * Sincronización con servidores para aplicaciones híbridas.

**¿Por qué aprender IndexedDB?**

Con el auge de las **aplicaciones progresivas (PWA)** y la necesidad de ofrecer experiencias fluidas incluso sin conexión, **IndexedDB** se ha convertido en una herramienta esencial para los desarrolladores modernos. A diferencia de otras formas de almacenamiento web, **IndexedDB** permite:

* Manejar grandes cantidades de datos sin comprometer el rendimiento.
* Trabajar con estructuras complejas, como objetos anidados y colecciones.
* Integrarse fácilmente con tecnologías como **Service Workers** para crear aplicaciones web rápidas, confiables y escalables.

Las aplicaciones web progresivas, conocidas como **PWA** (Progressive Web Apps), son aplicaciones web que utilizan tecnologías modernas para proporcionar una experiencia similar a las aplicaciones nativas, combinando lo mejor del desarrollo web y móvil. Son páginas o sitios web que, gracias a ciertas características, se comportan como aplicaciones instalables en dispositivos, ofreciendo funcionalidades avanzadas.

### ****Características principales de las PWA****

1. **Progresivas**:
   * Funcionan en cualquier navegador, sin importar el dispositivo o la versión del sistema operativo, siempre que sea compatible con estándares modernos.
2. **Responsivas**:
   * Se adaptan a diferentes tamaños de pantalla y dispositivos, como smartphones, tablets, laptops o desktops.
3. **Independientes de la conectividad**:
   * Pueden funcionar sin conexión o con una conectividad limitada gracias al uso de **Service Workers**, que almacenan recursos en caché.
4. **Seguras**:
   * Usan HTTPS para garantizar la seguridad y prevenir ataques como la interceptación de datos.
5. **Instalables**:
   * Los usuarios pueden añadirlas a la pantalla de inicio de sus dispositivos sin necesidad de pasar por una tienda de aplicaciones (App Store o Google Play).
6. **Actualizables**:
   * Siempre están actualizadas, ya que las actualizaciones se gestionan automáticamente a través de la red.
7. **Re-enganchables**:
   * Pueden enviar notificaciones push para mantener a los usuarios comprometidos, similar a las aplicaciones nativas.
8. **Descubribles**:
   * Al ser aplicaciones web, son indexadas por motores de búsqueda, lo que las hace más fáciles de encontrar que una app nativa.
9. **Ligeras**:
   * Ocupan menos espacio que las aplicaciones nativas y no requieren instalaciones pesadas.

### ****Ventajas de las PWA****

* **Desarrollo único**: Se construye una sola aplicación que funciona en todas las plataformas.
* **Ahorro de costos**: No se necesita desarrollar aplicaciones separadas para diferentes sistemas operativos (iOS, Android, etc.).
* **Rendimiento mejorado**: Al almacenar en caché recursos clave, se reducen los tiempos de carga.
* **Mayor alcance**: Al ser accesibles desde la web, no dependen de las tiendas de aplicaciones para su distribución.

### ****Ejemplos de PWA****

* **Twitter Lite**: Ofrece una experiencia rápida en redes móviles lentas.
* **Pinterest**: Incrementó su participación de usuarios gracias a su PWA.
* **Starbucks**: Permite a los usuarios navegar por su menú y hacer pedidos incluso sin conexión.
* **Uber**: Funciona de manera óptima en redes de baja velocidad.

En resumen, las PWA son una evolución del desarrollo web, diseñadas para ofrecer experiencias inmersivas, rápidas y confiables que combinan la accesibilidad de las páginas web con la funcionalidad de las aplicaciones nativas.

**IndexedDB** es una base de datos construida dentro del navegador, mucho más potente que localStorage.

* Almacena casi todo tipo de valores por claves, tipos de clave múltiple.
* Soporta transacciones para confiabilidad.
* Soporta consultas de rango por clave, e índices.
* Puede almacenar mucho mayor volumen de datos que localStorage.

Toda esta potencia es normalmente excesiva para las aplicaciones cliente-servidor tradicionales. **IndexedDB** está previsto para aplicaciones fuera de línea, para ser combinado con **ServiceWorkers** y otras tecnologías.

La interfaz nativa de IndexedDB, descrita en la <https://www.w3.org/TR/IndexedDB>, está basada en eventos.

**¿Dónde están los datos?**

Técnicamente, los datos son almacenados bajo el directorio raíz del usuario junto con la configuración personal del navegador, extensiones, etc.

Navegadores y usuarios diferentes tendrán cada uno su propio almacenamiento independiente.

1. **Apertura de una base de datos, “open”**

La apertura de una base de datos con **IndexedDB** se realiza mediante el método **indexedDB.open()**. Este proceso no solo crea o abre la base de datos, sino que también administra versiones, inicializa almacenes de objetos (object stores) y configura índices.

**Pasos para abrir una base de datos**

1. **Llamar a indexedDB.open()**
   * Este método toma como argumentos:
     + **Nombre de la base de datos:** Un string que identifica la base.
     + **Versión (opcional):** Un número entero que define la versión de la base. Si omites este parámetro, la versión será 1.
2. **Manejar los eventos del proceso de apertura**
   * **onsuccess:** Se dispara cuando la base se abre correctamente.
   * **onerror:** Ocurre si hay un problema al abrir la base.
   * **onupgradeneeded:** Se ejecuta si la base de datos necesita ser creada o su versión incrementada.
3. **Acceder al objeto IDBDatabase**
   * Una vez abierta, puedes usar el objeto IDBDatabase para realizar operaciones.

**Ejemplo básico de apertura**

Supongamos que quieres abrir una base de datos llamada "MiTienda" en su versión 1:

const request = indexedDB.open("MiTienda", 1);

request.onsuccess = function (event) {

console.log("Base de datos abierta exitosamente.");

const db = event.target.result;

// Aquí puedes trabajar con la base de datos

};

request.onerror = function (event) {

console.error("Error al abrir la base de datos:", event.target.error);

};

request.onupgradeneeded = function (event) {

console.log("Actualización de la base de datos en curso...");

const db = event.target.result;

// Crear un almacén de objetos si no existe

if (!db.objectStoreNames.contains("productos")) {

db.createObjectStore("productos", { keyPath: "id", autoIncrement: true });

console.log("Almacén 'productos' creado.");

}

};

**Explicación de los eventos clave**

#### **1.** onsuccess

* **Cuándo ocurre:**  
  Cuando la base de datos ya existe y se abre correctamente, o después de que onupgradeneeded haya terminado.
* **Qué puedes hacer:**  
  Accedes al objeto IDBDatabase para realizar operaciones.

#### **2.** onerror

* **Cuándo ocurre:**  
  Si hay un error en el proceso de apertura (por ejemplo, falta de permisos o incompatibilidad con el navegador).
* **Qué puedes hacer:**  
  Manejar el error, mostrar un mensaje al usuario, o intentar una recuperación.

#### **3.** onupgradeneeded

* **Cuándo ocurre:**
  + Cuando la base no existe y se debe crear.
  + Cuando el número de versión proporcionado es mayor al actual.
* **Qué puedes hacer:**
  + Crear almacenes de objetos (object stores).
  + Configurar índices y datos iniciales.

**Casos especiales**

1. **Abrir una base sin incrementar la versión**
   * Si solo quieres abrir la base sin realizar cambios estructurales:

const request = indexedDB.open("MiTienda");

1. **Incrementar la versión de la base**
   * Para realizar cambios estructurales (nuevos almacenes o índices), incrementa la versión:

const request = indexedDB.open("MiTienda", 2);

request.onupgradeneeded = function (event) {

const db = event.target.result;

if (!db.objectStoreNames.contains("usuarios")) {

db.createObjectStore("usuarios", { keyPath: "username" });

console.log("Almacén 'usuarios' creado.");

}

};

**Errores comunes**

* **Versión inferior:**  
  No puedes abrir una base con un número de versión inferior al actual.

const request = indexedDB.open("MiTienda", 0); // Error

* **Acceso simultáneo en versiones diferentes:**  
  Si otro cliente intenta abrir la misma base con una versión diferente, obtendrás un error hasta que ese cliente cierre su conexión.

1. **Almacenar objetos en IndexedDB**

**IndexedDB** es una API de almacenamiento basada en el navegador que permite almacenar grandes cantidades de datos estructurados, incluidos objetos, y permite búsquedas eficientes mediante índices. Los datos se organizan en object stores (almacenes de objetos), y los objetos se almacenan en forma de pares clave-valor.

**Pasos básicos para almacenar objetos en IndexedDB**

1. **Abrir o crear una base de datos**: Utilizas el método indexedDB.open() para abrir una conexión.
2. **Crear un almacén de objetos**: Esto se realiza en el evento onupgradeneeded, que se dispara cuando creas una nueva base de datos o la actualizas.
3. **Realizar transacciones**: Para almacenar, leer, actualizar o eliminar datos, necesitas transacciones.

**Ejemplo básico**

Supongamos que queremos almacenar información sobre libros.

#### 1. Crear la base de datos y el almacén de objetos

const request = indexedDB.open("LibraryDB", 1);

request.onupgradeneeded = function(event) {

const db = event.target.result;

// Crear un almacén de objetos llamado "books" con una clave primaria "id"

const objectStore = db.createObjectStore("books", { keyPath: "id" });

// Crear un índice para buscar por título

objectStore.createIndex("title", "title", { unique: false });

};

request.onsuccess = function(event) {

console.log("Base de datos abierta con éxito");

const db = event.target.result;

// Llamar a la función para añadir datos

addBook(db, { id: 1, title: "1984", author: "George Orwell", year: 1949 });

};

request.onerror = function(event) {

console.error("Error al abrir la base de datos:", event.target.errorCode);

};

\_\_\_\_\_\_\_\_\_\_\_ Crear la base de datos y el almacén de objetos \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Vamos a desglosar el código línea por línea

**1. Abrir o crear la base de datos**

const request = indexedDB.open("LibraryDB", 1);

* **indexedDB.open("LibraryDB", 1)**:
  + Intenta abrir una base de datos llamada LibraryDB.
  + Si la base de datos no existe, se creará.
  + El segundo parámetro (1) es el número de versión. Si este número es mayor que la versión actual de la base de datos, se dispara el evento onupgradeneeded.

**2. Evento onupgradeneeded: Crear el almacén de objetos**

request.onupgradeneeded = function(event) {

const db = event.target.result;

* Este evento se dispara cuando:
  + La base de datos se crea por primera vez.
  + La versión de la base de datos cambia.
* **const db = event.target.result**:
  + Obtiene la referencia al objeto IDBDatabase, que representa la base de datos abierta o creada.

const objectStore = db.createObjectStore("books", { keyPath: "id" });

* **db.createObjectStore("books", { keyPath: "id" })**:
  + Crea un almacén de objetos llamado "books".
  + Define la clave primaria con keyPath: "id", lo que significa que cada objeto almacenado en "books" debe tener un atributo único llamado id.

objectStore.createIndex("title", "title", { unique: false });

* **objectStore.createIndex("title", "title", { unique: false })**:
  + Crea un índice para el campo "title" dentro del almacén "books".
  + Esto permite búsquedas rápidas por el atributo title.
  + unique: false indica que varios objetos pueden tener el mismo valor para title.

**3. Evento onsuccess: Base de datos lista para usar**

request.onsuccess = function(event) {

console.log("Base de datos abierta con éxito");

const db = event.target.result;

* **request.onsuccess**:
  + Se dispara cuando la conexión con la base de datos se establece con éxito.
  + Aquí puedes comenzar a interactuar con la base de datos.
* **const db = event.target.result**:
  + Obtiene la referencia al objeto IDBDatabase para realizar operaciones (como insertar, leer o actualizar datos).

addBook(db, { id: 1, title: "1984", author: "George Orwell", year: 1949 });

* **addBook(db, {...})**:
  + Llama a una función externa llamada addBook (no definida en este fragmento) para agregar un objeto al almacén "books".
  + El objeto contiene:

{

id: 1,

title: "1984",

author: "George Orwell",

year: 1949

}

* + Este objeto se almacenará en el almacén "books", donde id será la clave primaria.

**4. Evento onerror: Manejo de errores**

request.onerror = function(event) {

console.error("Error al abrir la base de datos:", event.target.errorCode);

};

* **request.onerror**:
  + Se dispara si hay un problema al abrir o crear la base de datos.
  + **event.target.errorCode**:
    - Proporciona el código de error específico para diagnosticar el problema.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#### 2. Insertar un objeto en el almacén

function addBook(db, book) {

const transaction = db.transaction(["books"], "readwrite");

const objectStore = transaction.objectStore("books");

const request = objectStore.add(book);

request.onsuccess = function() {

console.log("Libro añadido con éxito:", book);

};

request.onerror = function(event) {

console.error("Error al añadir el libro:", event.target.error);

};

}

\_\_\_\_\_\_\_\_\_\_\_ Insertar un objeto en el almacén \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**1. Declaración de la función**

function addBook(db, book) {

* **db**: Es la conexión activa con la base de datos. Este es un objeto de tipo IDBDatabase, obtenido previamente al abrir la base de datos.
* **book**: Es el objeto que se quiere añadir al almacén "books". Por ejemplo:

{

id: 1,

title: "1984",

author: "George Orwell",

year: 1949

}

**2. Crear una transacción**

const transaction = db.transaction(["books"], "readwrite");

* **db.transaction()**:
  + Inicia una transacción para realizar operaciones sobre uno o más almacenes de la base de datos.
  + El primer argumento es una lista (array) de los nombres de los almacenes que estarán disponibles en esta transacción, en este caso, solo "books".
  + El segundo argumento es el modo de la transacción:
    - "readwrite": Permite leer y escribir datos.
    - "readonly": Permite solo lecturas.
* **transaction**: Es un objeto IDBTransaction que gestiona la operación y proporciona acceso al almacén de objetos.

**3. Acceder al almacén de objetos**

const objectStore = transaction.objectStore("books");

* **transaction.objectStore("books")**:
  + Obtiene una referencia al almacén de objetos "books" donde se almacenarán los datos.
  + Este almacén fue creado previamente en el evento onupgradeneeded.

**4. Añadir un objeto al almacén**

const request = objectStore.add(book);

* **objectStore.add(book)**:
  + Intenta añadir el objeto book al almacén "books".
  + La clave primaria (id) del objeto debe ser única. Si ya existe un objeto con la misma clave primaria, el intento fallará.
* **request**: Es un objeto IDBRequest que representa la operación asincrónica de añadir el objeto.

**5. Manejo del éxito**

request.onsuccess = function() {

console.log("Libro añadido con éxito:", book);

};

* **request.onsuccess**:
  + Se ejecuta cuando el objeto book se ha añadido correctamente al almacén.
  + Aquí, se imprime un mensaje de éxito junto con el objeto añadido.

**6. Manejo de errores**

request.onerror = function(event) {

console.error("Error al añadir el libro:", event.target.error);

};

* **request.onerror**:
  + Se ejecuta si ocurre un error al intentar añadir el objeto.
  + **event.target.error**:
    - Proporciona detalles sobre el error (por ejemplo, clave duplicada).

#### 3. Leer datos del almacén

function getBookById(db, id) {

const transaction = db.transaction(["books"], "readonly");

const objectStore = transaction.objectStore("books");

const request = objectStore.get(id);

request.onsuccess = function(event) {

if (request.result) {

console.log("Libro encontrado:", request.result);

} else {

console.log("No se encontró el libro con ID:", id);

}

};

request.onerror = function(event) {

console.error("Error al obtener el libro:", event.target.error);

};

}

\_\_\_\_\_\_\_\_\_\_\_ Leer datos del almacén \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Este código define una función que busca y obtiene un libro específico de un almacén de objetos en una base de datos **IndexedDB** utilizando su ID. Vamos a desglosarlo paso a paso:

**1. Declaración de la función**

function getBookById(db, id) {

* **db**: Es la conexión activa con la base de datos. Este es un objeto de tipo IDBDatabase.
* **id**: Es el identificador único del libro que se desea buscar. Este valor corresponde al keyPath definido previamente ("id" en el almacén "books").

**2. Crear una transacción**

const transaction = db.transaction(["books"], "readonly");

* **db.transaction()**:
  + Inicia una transacción para interactuar con el almacén "books".
  + El primer argumento es una lista con los nombres de los almacenes que estarán disponibles en esta transacción (aquí, solo "books").
  + El segundo argumento especifica el modo de la transacción:
    - "readonly": Solo permite leer datos (no escribir ni modificar).
* **transaction**: Es un objeto IDBTransaction que gestiona esta operación.

**3. Obtener el almacén de objetos**

const objectStore = transaction.objectStore("books");

* **transaction.objectStore("books")**:
  + Obtiene una referencia al almacén de objetos "books" donde se almacenan los datos.
  + Este almacén fue creado previamente en el evento onupgradeneeded.

**4. Buscar el libro por ID**

const request = objectStore.get(id);

* **objectStore.get(id)**:
  + Realiza una operación para buscar un objeto específico dentro del almacén "books", utilizando el valor de la clave primaria (id).
  + Devuelve un objeto IDBRequest, que representa la operación asincrónica de búsqueda.

**5. Manejo del éxito**

request.onsuccess = function(event) {

if (request.result) {

console.log("Libro encontrado:", request.result);

} else {

console.log("No se encontró el libro con ID:", id);

}

};

* **request.onsuccess**:
  + Se ejecuta cuando la operación de búsqueda se completa con éxito.
  + **request.result**:
    - Contiene el objeto encontrado si existe un libro con el ID especificado.
    - Si no hay coincidencias, devuelve undefined.
* **Acciones**:
  + Si el libro se encuentra, imprime el objeto del libro en la consola:
  + Libro encontrado: { id: 1, title: "1984", author: "George Orwell", year: 1949 }
  + Si no se encuentra, muestra un mensaje indicando que no hay un libro con el ID proporcionado.

**6. Manejo de errores**

request.onerror = function(event) {

console.error("Error al obtener el libro:", event.target.error);

};

* **request.onerror**:
  + Se ejecuta si ocurre un error durante la operación de búsqueda.
  + **event.target.error**:
    - Proporciona detalles sobre el error.
* Ejemplo de mensaje de error en consola:
* Error al obtener el libro: [DOMException: Error while fetching key]

**Ejemplo de uso**

// Supongamos que `db` es la conexión activa a la base de datos

getBookById(db, 1);

**Caso exitoso**: Si existe un libro con id: 1, la consola mostrará:

Libro encontrado: { id: 1, title: "1984", author: "George Orwell", year: 1949 }

**Caso no exitoso**: Si no hay ningún libro con ese ID, la consola mostrará:

No se encontró el libro con ID: 1

**Caso de error**: Si ocurre un error (por ejemplo, si la base de datos está cerrada), la consola mostrará:

Error al obtener el libro: [DOMException: Error while fetching key]

#### 4. Actualizar un objeto existente

function updateBook(db, updatedBook) {

const transaction = db.transaction(["books"], "readwrite");

const objectStore = transaction.objectStore("books");

const request = objectStore.put(updatedBook);

request.onsuccess = function() {

console.log("Libro actualizado con éxito:", updatedBook);

};

request.onerror = function(event) {

console.error("Error al actualizar el libro:", event.target.error);

};

}

\_\_\_\_\_\_\_\_\_\_\_ Actualizar objeto existente\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Este código define una función que actualiza un libro en el almacén de objetos "books" de una base de datos **IndexedDB**. Analicemos su funcionamiento en detalle:

**1. Declaración de la función**

function updateBook(db, updatedBook) {

* **db**: Es la conexión activa con la base de datos. Este es un objeto de tipo IDBDatabase, que se obtiene al abrir la base de datos.
* **updatedBook**: Es un objeto que contiene los datos del libro que se desea actualizar. Este objeto debe incluir un campo con el mismo id que el libro existente en el almacén para identificarlo correctamente.

Ejemplo de updatedBook:

{

id: 1,

title: "1984 (Edición Revisada)",

author: "George Orwell",

year: 1950

}

**2. Crear una transacción**

const transaction = db.transaction(["books"], "readwrite");

* **db.transaction()**:
  + Inicia una transacción para interactuar con uno o más almacenes.
  + El primer argumento es una lista con los nombres de los almacenes que estarán disponibles en esta transacción (aquí, solo "books").
  + El segundo argumento especifica el modo de la transacción:
    - "readwrite": Permite leer y escribir datos.
* **transaction**: Es un objeto IDBTransaction que gestiona las operaciones dentro de esta transacción.

**3. Obtener el almacén de objetos**

const objectStore = transaction.objectStore("books");

* **transaction.objectStore("books")**:
  + Obtiene una referencia al almacén de objetos "books".
  + Este almacén fue definido previamente durante la creación de la base de datos.

**4. Actualizar el objeto**

const request = objectStore.put(updatedBook);

* **objectStore.put(updatedBook)**:
  + Actualiza un objeto existente en el almacén basado en su clave primaria (id).
  + Si no existe un objeto con el mismo id, este método **creará un nuevo objeto** en el almacén.
* **request**: Es un objeto IDBRequest que representa la operación asincrónica de actualización.

**5. Manejo del éxito**

request.onsuccess = function() {

console.log("Libro actualizado con éxito:", updatedBook);

};

* **request.onsuccess**:
  + Se ejecuta cuando el objeto updatedBook se actualiza correctamente en el almacén.
  + Imprime un mensaje de éxito junto con el objeto actualizado.

Ejemplo de salida en consola:

Libro actualizado con éxito: { id: 1, title: "1984 (Edición Revisada)", author: "George Orwell", year: 1950 }

**6. Manejo de errores**

request.onerror = function(event) {

console.error("Error al actualizar el libro:", event.target.error);

};

* **request.onerror**:
  + Se ejecuta si ocurre un error durante la operación de actualización.
  + **event.target.error**:
    - Proporciona detalles sobre el error.

Ejemplo de mensaje de error en consola:

Error al actualizar el libro: [DOMException: ConstraintError]

Un posible error podría ser un **conflicto de clave** (si el almacén tiene restricciones adicionales, como índices únicos).

**Ejemplo de uso**

const updatedBook = {

id: 1,

title: "1984 (Edición Revisada)",

author: "George Orwell",

year: 1950

};

// Supongamos que `db` es la conexión activa a la base de datos

updateBook(db, updatedBook);

#### 5. Eliminar un objeto

function deleteBook(db, id) {

const transaction = db.transaction(["books"], "readwrite");

const objectStore = transaction.objectStore("books");

const request = objectStore.delete(id);

request.onsuccess = function() {

console.log("Libro eliminado con éxito con ID:", id);

};

request.onerror = function(event) {

console.error("Error al eliminar el libro:", event.target.error);

};

}

**Notas importantes**

1. **Claves primarias**:
   * Si especificas un keyPath, cada objeto debe incluir esa clave. Por ejemplo, { id: 1 }.
   * También puedes usar claves generadas automáticamente configurando autoIncrement: true.
2. **Índices**:
   * Los índices permiten búsquedas eficientes. Por ejemplo, puedes buscar libros por title usando objectStore.index("title").get().
3. **Transacciones**:
   * Toda operación (lectura/escritura) se realiza dentro de una transacción. Especifica si la transacción es "readonly" o "readwrite".

\_\_\_\_\_\_\_\_\_\_\_ Eliminar objeto \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Este código define una función que elimina un libro del almacén de objetos "books" en una base de datos **IndexedDB**. A continuación, te explico cada parte del código:

**1. Declaración de la función**

function deleteBook(db, id) {

* **db**: Es la conexión activa con la base de datos. Este es un objeto de tipo IDBDatabase, que se obtiene al abrir la base de datos.
* **id**: Es el identificador del libro que se desea eliminar. Este identificador corresponde al valor de la clave primaria (keyPath) definida en el almacén "books".

Por ejemplo, si el almacén contiene un libro con el siguiente objeto:

{ id: 1, title: "1984", author: "George Orwell", year: 1949 }

Llamar a deleteBook(db, 1) eliminará este libro.

**2. Crear una transacción**

const transaction = db.transaction(["books"], "readwrite");

* **db.transaction()**:
  + Inicia una transacción para interactuar con el almacén "books".
  + El primer argumento es un array con los nombres de los almacenes que estarán disponibles en esta transacción (aquí, solo "books").
  + El segundo argumento especifica el modo de la transacción:
    - "readwrite": Permite realizar operaciones de lectura y escritura.
* **transaction**: Es un objeto IDBTransaction que gestiona todas las operaciones dentro de esta transacción.

**3. Obtener el almacén de objetos**

const objectStore = transaction.objectStore("books");

* **transaction.objectStore("books")**:
  + Obtiene una referencia al almacén de objetos "books".
  + Este almacén fue creado previamente al configurar la base de datos.

**4. Eliminar el objeto**

const request = objectStore.delete(id);

* **objectStore.delete(id)**:
  + Solicita la eliminación del objeto con el valor de clave primaria especificado por id.
  + Si no se encuentra un objeto con esa clave, no se realiza ninguna acción.
* **request**: Es un objeto IDBRequest que representa la operación asincrónica de eliminación.

**5. Manejo del éxito**

request.onsuccess = function() {

console.log("Libro eliminado con éxito con ID:", id);

};

* **request.onsuccess**:
  + Se ejecuta cuando el objeto con el identificador id se elimina con éxito del almacén.
  + Imprime un mensaje de éxito en la consola.

Ejemplo de salida en consola:

Libro eliminado con éxito con ID: 1

**6. Manejo de errores**

request.onerror = function(event) {

console.error("Error al eliminar el libro:", event.target.error);

};

* **request.onerror**:
  + Se ejecuta si ocurre un error durante la operación de eliminación.
  + **event.target.error**:
    - Proporciona detalles sobre el error.

Ejemplo de mensaje de error en consola:

Error al eliminar el libro: [DOMException: ConstraintError]

Un posible error podría ser un problema con la transacción o una restricción en el almacén.

**Ejemplo de uso**

Supongamos que ya existe un almacén "books" con un libro que tiene id: 1. Puedes eliminarlo de la siguiente forma:

deleteBook(db, 1);

**Comportamiento esperado**

1. **Caso exitoso**: Si existe un libro con id: 1, este se eliminará del almacén y se mostrará:

Libro eliminado con éxito con ID: 1

1. **Caso sin coincidencias**: Si no existe un libro con id: 1, el método delete no hace nada. Sin embargo, se considera una operación exitosa, y el mensaje será el mismo:

Libro eliminado con éxito con ID: 1

1. **Caso de error**: Si ocurre un problema (como un fallo en la transacción), se imprimirá un mensaje de error.

**Consideraciones adicionales**

* Este método no verifica si el objeto existe antes de eliminarlo.
* Si necesitas una confirmación previa, puedes usar el método get para verificar su existencia antes de llamar a delete.

1. **Las transacciones en IndexedDB**

Las **transacciones en IndexedDB** son una parte fundamental para realizar operaciones de lectura y escritura en la base de datos. Una transacción es un contexto de ejecución en el que puedes interactuar con uno o más almacenes de objetos (**object stores**) de manera segura y controlada. Todas las operaciones de acceso a datos (lectura, escritura, actualización o eliminación) deben realizarse dentro de una transacción.

**Características principales de las transacciones en IndexedDB**

1. **Atomicidad**:
   * Todas las operaciones en una transacción se ejecutan completamente o no se ejecutan en absoluto. Si ocurre un error, todos los cambios realizados en esa transacción se revierten automáticamente.
2. **Aislamiento**:
   * Cada transacción se ejecuta de forma aislada. Las modificaciones realizadas por una transacción no son visibles para otras transacciones hasta que la primera se completa.
3. **Modos de acceso**:
   * "readonly": Permite solo leer datos del almacén de objetos.
   * "readwrite": Permite leer y modificar datos en el almacén de objetos.
4. **Duración limitada**:
   * Una transacción tiene un tiempo de vida corto. Se cierra automáticamente al completar todas las operaciones asincrónicas asociadas.

**Cómo crear una transacción**

Para crear una transacción, utilizas el método transaction() del objeto IDBDatabase.

**Sintaxis básica**

const transaction = db.transaction(storeNames, mode);

* **storeNames**:
  + Un string o un array de strings que especifica los nombres de los almacenes de objetos que serán accesibles en la transacción.
  + Ejemplo: "books" o ["books", "authors"].
* **mode** (opcional):
  + "readonly": Solo lectura.
  + "readwrite": Lectura y escritura.
* **transaction**:
  + Representa la transacción activa. Es un objeto de tipo IDBTransaction.

**Eventos de una transacción**

1. **oncomplete**:
   * Se dispara cuando la transacción finaliza correctamente.
   * Útil para realizar acciones posteriores a las operaciones.
2. **onerror**:
   * Se dispara si ocurre un error en alguna operación dentro de la transacción.
3. **onabort**:
   * Se dispara si la transacción se cancela manualmente o por un error no manejado.

**Ejemplo de uso básico**

**Crear una transacción y agregar un objeto**

function addBook(db, book) {

// Crear una transacción en el almacén "books" en modo "readwrite"

const transaction = db.transaction("books", "readwrite");

// Obtener el almacén de objetos

const objectStore = transaction.objectStore("books");

// Agregar un libro al almacén

const request = objectStore.add(book);

// Manejar el éxito de la operación

request.onsuccess = function() {

console.log("Libro agregado:", book);

};

// Manejar errores

request.onerror = function(event) {

console.error("Error al agregar el libro:", event.target.error);

};

// Manejar la finalización de la transacción

transaction.oncomplete = function() {

console.log("Transacción completada.");

};

// Manejar errores en la transacción

transaction.onerror = function(event) {

console.error("Error en la transacción:", event.target.error);

};

}

**Modos de acceso en transacciones**

**1. Modo "readonly"**

* Se utiliza cuando solo necesitas leer datos de la base de datos.
* Operaciones permitidas: get, getAll, count, y openCursor.

function getAllBooks(db) {

const transaction = db.transaction("books", "readonly");

const objectStore = transaction.objectStore("books");

const request = objectStore.getAll();

request.onsuccess = function() {

console.log("Libros encontrados:", request.result);

};

transaction.onerror = function(event) {

console.error("Error al leer los libros:", event.target.error);

};

}

**2. Modo "readwrite"**

* Se utiliza cuando necesitas realizar operaciones de lectura y escritura en los datos.
* Operaciones permitidas: add, put, delete, y cualquier operación permitida en el modo "readonly".

function updateBook(db, book) {

const transaction = db.transaction("books", "readwrite");

const objectStore = transaction.objectStore("books");

const request = objectStore.put(book);

request.onsuccess = function() {

console.log("Libro actualizado:", book);

};

transaction.onerror = function(event) {

console.error("Error al actualizar el libro:", event.target.error);

};

}

**Control de errores y finalización**

**Abortar una transacción manualmente**

Puedes abortar una transacción utilizando el método transaction.abort().

function deleteBookWithAbort(db, id) {

const transaction = db.transaction("books", "readwrite");

const objectStore = transaction.objectStore("books");

const request = objectStore.delete(id);

request.onsuccess = function() {

console.log("Libro eliminado:", id);

// Abortar la transacción manualmente

transaction.abort();

};

transaction.onabort = function() {

console.log("Transacción abortada.");

};

transaction.onerror = function(event) {

console.error("Error en la transacción:", event.target.error);

};

}

**Eventos oncomplete y onerror**

* **oncomplete**: Se ejecuta después de que todas las operaciones de la transacción se hayan procesado correctamente.
* **onerror**: Se ejecuta si ocurre un error en cualquier operación dentro de la transacción.

**Consideraciones importantes**

1. **Las transacciones son limitadas en tiempo**:
   * Se cierran automáticamente después de procesar todas las operaciones asincrónicas.
   * Si intentas realizar operaciones fuera de la transacción activa, obtendrás un error.
2. **Una transacción no puede cambiar de modo**:
   * Por ejemplo, no puedes cambiar una transacción "readonly" a "readwrite".
3. **Uso de múltiples almacenes**:
   * Una transacción puede acceder a varios almacenes si se especifican al momento de su creación:
   * const transaction = db.transaction(["books", "authors"], "readwrite");
4. **Errores y reversión**:
   * Si ocurre un error en una operación dentro de una transacción, todas las operaciones se revierten automáticamente.

1. **Autocommit**

En **IndexedDB**, una base de datos orientada a objetos en el navegador, la culminación automática de las transacciones (también conocida como "autocommit") es una característica que determina cómo y cuándo las transacciones se confirman o finalizan automáticamente.

**1. Concepto de una transacción en IndexedDB**

* Una **transacción** en IndexedDB es un contexto en el que se realizan operaciones como lecturas, escrituras, actualizaciones o eliminaciones de datos.
* Las transacciones pueden ser de dos tipos principales:
  + **Solo lectura (**readonly**)**: Permiten leer datos sin modificarlos.
  + **Lectura/escritura (**readwrite**)**: Permiten leer y modificar datos.

**2. Cómo funciona el autocommit**

* **Inicio de la transacción**: Una transacción comienza cuando invocas el método transaction de una base de datos.
* **Culminación automática**: Una transacción se confirma automáticamente (se "autocommit") en los siguientes casos:
  1. **No hay operaciones pendientes**: Una vez que todas las solicitudes dentro de la transacción han sido procesadas, la transacción se da por completada automáticamente.
  2. **Tiempo de espera (timeout)**: Si no se realizan más operaciones dentro de un intervalo determinado, la transacción puede cerrarse automáticamente.
* Cuando una transacción se completa automáticamente, los cambios realizados durante la transacción se **confirman permanentemente** en la base de datos.

**3. Importancia del autocommit**

* **Eficiencia**: Al permitir que las transacciones se completen automáticamente, se asegura que no permanezcan abiertas más tiempo del necesario, lo que mejora el rendimiento y evita bloqueos.
* **Seguridad de los datos**: Si una transacción se confirma automáticamente, los datos son persistentes y están seguros ante fallos posteriores del sistema.

**4. Ejemplo práctico**

const request = indexedDB.open("miBaseDeDatos", 1);

request.onsuccess = function(event) {

// Aquí obtenemos la conexión a la base de datos

const db = event.target.result;

// Iniciar una transacción readwrite

const transaction = db.transaction("miObjetoAlmacen", "readwrite");

const store = transaction.objectStore("miObjetoAlmacen");

// Agregar un elemento al almacén de objetos

store.add({ id: 1, nombre: "Ejemplo" });

// No es necesario llamar a `transaction.commit()`, se autocompleta

transaction.oncomplete = function() {

console.log("Transacción completada automáticamente.");

};

transaction.onerror = function(event) {

console.error("Error en la transacción:", event.target.error);

};

};

**Vamos a desglosar y explicar el código paso a paso para entender qué hace.**

**1. Apertura de la base de datos**

const request = indexedDB.open("miBaseDeDatos", 1);

* indexedDB.open("miBaseDeDatos", 1): Intenta abrir una conexión a una base de datos llamada "miBaseDeDatos" con la versión 1.
  + Si la base de datos no existe, se creará automáticamente.
  + Si la base de datos ya existe pero tiene una versión diferente, se activará el evento onupgradeneeded para actualizarla.
  + Si la base de datos ya existe con la versión 1, simplemente se abrirá.

**2. Evento onsuccess: Conexión exitosa**

request.onsuccess = function(event) {

const db = event.target.result;

// ...

};

* request.onsuccess: Se ejecuta cuando la conexión a la base de datos se establece correctamente.
* event.target.result: Contiene la instancia de la base de datos (IDBDatabase) que nos permite interactuar con ella. Se asigna a la variable db.

**3. Creación de una transacción**

const transaction = db.transaction("miObjetoAlmacen", "readwrite");

const store = transaction.objectStore("miObjetoAlmacen");

* db.transaction("miObjetoAlmacen", "readwrite"): Inicia una transacción para el almacén de objetos llamado "miObjetoAlmacen".
  + "readwrite" indica que se pueden realizar operaciones de lectura y escritura.
  + Si el almacén "miObjetoAlmacen" no existe, la operación fallará.
* transaction.objectStore("miObjetoAlmacen"): Obtiene una referencia al almacén de objetos (o tabla) "miObjetoAlmacen" para interactuar con él.

**4. Agregar un objeto al almacén**

store.add({ id: 1, nombre: "Ejemplo" });

* store.add({ id: 1, nombre: "Ejemplo" }): Intenta insertar un objeto con las propiedades:
  + id: Identificador único del objeto.
  + nombre: Nombre o descripción del objeto.
* Si el valor de id ya existe en el almacén y no está configurado como clave duplicada, se producirá un error.

**5. Manejo de la transacción**

transaction.oncomplete = function() {

console.log("Transacción completada automáticamente.");

};

transaction.onerror = function(event) {

console.error("Error en la transacción:", event.target.error);

};

* transaction.oncomplete: Este evento se activa cuando todas las operaciones de la transacción se han completado correctamente. En este caso, se imprime un mensaje indicando que la transacción se completó automáticamente.
* transaction.onerror: Este evento se activa si ocurre algún error durante la transacción (por ejemplo, si intentas agregar un objeto con una clave duplicada). En ese caso, imprime el error en la consola.

**Resumen de lo que hace el código**

1. Abre o crea una base de datos llamada "miBaseDeDatos" con la versión 1.
2. Una vez abierta con éxito, inicia una transacción en el almacén de objetos "miObjetoAlmacen" en modo de lectura/escritura.
3. Agrega un objeto con id: 1 y nombre: "Ejemplo" al almacén.
4. Espera a que la transacción se complete o falle:
   * Si tiene éxito, imprime un mensaje en la consola.
   * Si falla, muestra el error en la consola.

**Consideraciones importantes**

* Si "miObjetoAlmacen" no existe, el código fallará. Debes asegurarte de crear el almacén de objetos en el evento onupgradeneeded cuando configures la base de datos por primera vez.
* **Ejemplo de configuración en** onupgradeneeded**:**

request.onupgradeneeded = function(event) {

const db = event.target.result;

db.createObjectStore("miObjetoAlmacen", { keyPath: "id" }); // Crear el almacén con "id" como clave primaria

};

1. **Manejo de errores en IndexedDB**

**El manejo de errores en IndexedDB es crucial para garantizar la estabilidad y la gestión adecuada de las operaciones en la base de datos. IndexedDB proporciona varios mecanismos para capturar y manejar errores que pueden ocurrir durante la apertura de la base de datos, la ejecución de transacciones y las operaciones específicas en los almacenes de objetos.**

**1. Eventos de error en IndexedDB**

IndexedDB utiliza eventos para notificar errores. Los principales eventos relacionados con errores son:

* onerror: Se dispara cuando ocurre un error durante una operación.
* onabort: Se dispara cuando una transacción es abortada explícita o implícitamente (por ejemplo, debido a un error).
* onblocked: Se dispara si una nueva versión de la base de datos no puede activarse porque aún hay conexiones activas en la versión anterior.

**2. Cómo manejar errores comunes**

**a) Errores al abrir la base de datos**

Cuando intentas abrir una base de datos con indexedDB.open(), pueden ocurrir errores como:

* Permisos denegados.
* La base de datos está bloqueada por otra conexión.

**Ejemplo:**

const request = indexedDB.open("miBaseDeDatos", 1);

request.onerror = function(event) {

console.error("Error al abrir la base de datos:", event.target.error);

};

request.onsuccess = function(event) {

console.log("Base de datos abierta con éxito.");

};

**b) Errores durante la actualización de la base de datos (onupgradeneeded)**

El evento onupgradeneeded puede generar errores si intentas realizar cambios no válidos, como duplicar claves.

**Ejemplo:**

request.onupgradeneeded = function(event) {

const db = event.target.result;

try {

db.createObjectStore("miObjetoAlmacen", { keyPath: "id" });

} catch (error) {

console.error("Error durante la actualización de la base de datos:", error.message);

}

};

**c) Errores en transacciones**

Las transacciones tienen un manejador de eventos onerror que captura problemas como claves duplicadas o violaciones de restricciones.

**Ejemplo:**

const transaction = db.transaction("miObjetoAlmacen", "readwrite");

const store = transaction.objectStore("miObjetoAlmacen");

store.add({ id: 1, nombre: "Ejemplo" });

transaction.oncomplete = function() {

console.log("Transacción completada con éxito.");

};

transaction.onerror = function(event) {

console.error("Error en la transacción:", event.target.error);

};

**d) Manejo de abortos de transacción**

Las transacciones pueden ser abortadas manualmente con transaction.abort() o automáticamente si ocurre un error.

**Ejemplo:**

transaction.onabort = function(event) {

console.error("Transacción abortada:", event.target.error);

};

**e) Errores en operaciones individuales**

Cada operación (add, put, get, delete, etc.) genera su propia solicitud (IDBRequest) que puede manejar errores individualmente.

**Ejemplo:**

const request = store.add({ id: 1, nombre: "Ejemplo" });

request.onsuccess = function() {

console.log("Elemento agregado con éxito.");

};

request.onerror = function(event) {

console.error("Error al agregar el elemento:", event.target.error);

};

**3. Error Global (window.onerror)**

Si un error no se maneja localmente, puedes capturarlo usando un manejador global de errores:

**Ejemplo:**

window.onerror = function(message, source, lineno, colno, error) {

console.error("Error global capturado:", message, error);

};

**4. Mejores prácticas para manejar errores**

1. **Usar manejadores de eventos (**onerror **y** onabort**)**: Captura errores en todos los niveles (base de datos, transacciones y operaciones) para asegurarte de que no pasen desapercibidos.
2. **Registrar mensajes de error útiles**: Usa event.target.error para obtener detalles del error.
3. **Validar datos antes de las operaciones**: Asegúrate de que los datos cumplen con los requisitos del esquema para evitar errores comunes como claves duplicadas.
4. **Usar bloques** try-catch **en** onupgradeneeded: Los cambios en la estructura de la base de datos son propensos a errores, por lo que es útil manejarlos explícitamente.
5. **Cerrar conexiones inactivas**: Para evitar errores de bloqueo (blocked), asegúrate de cerrar conexiones no utilizadas con db.close().
6. **Proporcionar retroalimentación al usuario**: Informa al usuario si ocurre un error crítico y sugiere posibles acciones correctivas.

**5. Ejemplo completo de manejo de errores**

const request = indexedDB.open("miBaseDeDatos", 1);

request.onerror = function(event) {

console.error("No se pudo abrir la base de datos:", event.target.error);

};

request.onupgradeneeded = function(event) {

const db = event.target.result;

try {

db.createObjectStore("miObjetoAlmacen", { keyPath: "id" });

console.log("Almacén de objetos creado con éxito.");

} catch (error) {

console.error("Error al crear el almacén de objetos:", error.message);

}

};

request.onsuccess = function(event) {

const db = event.target.result;

const transaction = db.transaction("miObjetoAlmacen", "readwrite");

const store = transaction.objectStore("miObjetoAlmacen");

const addRequest = store.add({ id: 1, nombre: "Ejemplo" });

addRequest.onerror = function(event) {

console.error("Error al agregar el elemento:", event.target.error);

};

transaction.oncomplete = function() {

console.log("Transacción completada con éxito.");

};

transaction.onerror = function(event) {

console.error("Error en la transacción:", event.target.error);

};

};

**Código con errores intencionales**

const request = indexedDB.open("miBaseDeDatos", 1);

request.onsuccess = function(event) {

const db = event.target.result;

// Error 1: Intentar acceder a un almacén que no existe

const transaction = db.transaction("almacenInexistente", "readwrite");

const store = transaction.objectStore("almacenInexistente");

// Error 2: Agregar un objeto sin cumplir la clave primaria definida

store.add({ nombre: "SinClavePrimaria" }); // Falta el campo `id`

transaction.oncomplete = function() {

console.log("Transacción completada.");

};

transaction.onerror = function(event) {

console.error("Error en la transacción:", event.target.error);

};

};

request.onupgradeneeded = function(event) {

const db = event.target.result;

// Error 3: Intentar crear un almacén con un nombre duplicado

db.createObjectStore("miObjetoAlmacen", { keyPath: "id" });

db.createObjectStore("miObjetoAlmacen", { keyPath: "id" }); // Error duplicado

};

request.onerror = function(event) {

console.error("Error al abrir la base de datos:", event.target.error);

};

**Errores que deberías observar**

**Error al acceder a un almacén inexistente:**

* + El almacén "almacenInexistente" no existe, por lo que la transacción fallará.
  + Mensaje esperado: "NotFoundError: The object store 'almacenInexistente' does not exist"

**Error al agregar un objeto sin clave primaria:**

* + El almacén "miObjetoAlmacen" (si existe) requiere un campo id como clave primaria. La omisión de este campo generará un error.
  + Mensaje esperado: "DataError: KeyPath was not satisfied"

**Error al crear un almacén duplicado:**

* + En el evento onupgradeneeded, se intenta crear un almacén con el mismo nombre dos veces, lo que genera un error.
  + Mensaje esperado: "ConstraintError: An object store with the specified name already exists"

**Error general al abrir la base de datos:**

* + Si ocurre un error al abrir la base de datos (por ejemplo, permisos denegados), se capturará en el evento request.onerror.

**Cómo probar y manejar los errores**

* Ejecuta este código en el navegador (en la consola de desarrollo).
* Observa los mensajes de error en la consola y su origen.
* Agrega manejadores adicionales para capturar y detallar cada error:

**Ejemplo:**

transaction.onerror = function(event) {

console.error("Error detallado en la transacción:", event.target.error);

};

request.onupgradeneeded = function(event) {

try {

const db = event.target.result;

db.createObjectStore("miObjetoAlmacen", { keyPath: "id" });

db.createObjectStore("miObjetoAlmacen", { keyPath: "id" }); // Intento duplicado

} catch (error) {

console.error("Error durante la actualización:", error.message);

}

};

1. **Búsquedas en un almacén de objetos**

En **IndexedDB**, las búsquedas en un almacén de objetos (object store) se realizan utilizando los métodos del almacén y, en ocasiones, índices (index) cuando necesitas buscar por una propiedad específica que no es la clave primaria. A continuación, te explico los conceptos básicos y los métodos para realizar búsquedas:

**1. Métodos principales para búsquedas en un almacén de objetos**

El almacén de objetos (IDBObjectStore) ofrece varios métodos para buscar datos. Los más comunes son:

**a. get()**

* Recupera un único registro por su clave primaria.
* Retorna el valor asociado con la clave si existe, o undefined si no se encuentra.

const transaction = db.transaction("miAlmacen", "readonly");  
const objectStore = transaction.objectStore("miAlmacen");  
  
const request = objectStore.get(1); // Buscar el registro con clave primaria 1  
  
request.onsuccess = (event) => {  
 const resultado = event.target.result;  
 if (resultado) {  
 console.log("Registro encontrado:", resultado);  
 } else {  
 console.log("No se encontró el registro");  
 }  
};  
  
request.onerror = (event) => {  
 console.error("Error al buscar el registro:", event.target.error);  
};

**b. getAll()**

* Recupera todos los registros del almacén o una parte específica si se le pasa un rango.
* Muy útil para obtener múltiples resultados sin necesidad de recorrer un cursor.

const transaction = db.transaction("miAlmacen", "readonly");  
const objectStore = transaction.objectStore("miAlmacen");  
  
const request = objectStore.getAll(); // Obtiene todos los registros  
  
request.onsuccess = (event) => {  
 console.log("Registros encontrados:", event.target.result);  
};  
  
request.onerror = (event) => {  
 console.error("Error al buscar registros:", event.target.error);  
};

Si necesitas buscar dentro de un rango específico, puedes usar un objeto IDBKeyRange:

const rango = IDBKeyRange.bound(1, 5); // Claves entre 1 y 5, inclusivas  
const request = objectStore.getAll(rango);  
  
request.onsuccess = (event) => {  
 console.log("Registros encontrados en el rango:", event.target.result);  
};

**c. openCursor()**

* Permite recorrer todos los registros del almacén uno por uno.
* Útil cuando necesitas buscar datos que no están directamente indexados o aplicar filtros personalizados.

const transaction = db.transaction("miAlmacen", "readonly");  
const objectStore = transaction.objectStore("miAlmacen");  
  
const request = objectStore.openCursor();  
  
request.onsuccess = (event) => {  
 const cursor = event.target.result;  
 if (cursor) {  
 console.log("Clave:", cursor.key, "Valor:", cursor.value);  
 cursor.continue(); // Continúa con el siguiente registro  
 } else {  
 console.log("No hay más registros");  
 }  
};  
  
request.onerror = (event) => {  
 console.error("Error al usar el cursor:", event.target.error);  
};

También puedes filtrar con un rango:

const rango = IDBKeyRange.bound(10, 20); // Claves entre 10 y 20  
const request = objectStore.openCursor(rango);  
  
request.onsuccess = (event) => {  
 const cursor = event.target.result;  
 if (cursor) {  
 console.log("Clave:", cursor.key, "Valor:", cursor.value);  
 cursor.continue();  
 }  
};

**2. Uso de índices (index) para búsquedas por propiedades no clave**

Si deseas buscar datos basándote en una propiedad diferente a la clave primaria, necesitas un índice. Un índice es como un atajo que permite buscar rápidamente por una propiedad.

**a. Crear un índice**

El índice se crea durante la etapa de actualización (onupgradeneeded) de la base de datos:

const request = indexedDB.open("MiBaseDeDatos", 2);  
  
request.onupgradeneeded = (event) => {  
 const db = event.target.result;  
  
 const objectStore = db.createObjectStore("miAlmacen", { keyPath: "id" });  
  
 // Crear un índice para la propiedad "nombre"  
 objectStore.createIndex("porNombre", "nombre", { unique: false });  
};

**b. Buscar con un índice**

Una vez que tienes un índice, puedes realizar búsquedas basadas en esa propiedad:

**Usando get() en un índice:**

const transaction = db.transaction("miAlmacen", "readonly");  
const objectStore = transaction.objectStore("miAlmacen");  
  
const index = objectStore.index("porNombre"); // Accede al índice "porNombre"  
  
const request = index.get("Juan"); // Buscar por el nombre "Juan"  
  
request.onsuccess = (event) => {  
 console.log("Resultado encontrado:", event.target.result);  
};  
  
request.onerror = (event) => {  
 console.error("Error al buscar en el índice:", event.target.error);  
};

**Usando getAll() en un índice:**

const request = index.getAll("Juan"); // Obtiene todos los registros con nombre "Juan"  
  
request.onsuccess = (event) => {  
 console.log("Resultados encontrados:", event.target.result);  
};

**Usando openCursor() en un índice:**

Esto es útil para recorrer múltiples registros que coinciden con un criterio.

const rango = IDBKeyRange.only("Juan"); // Buscar solo por el nombre "Juan"  
const cursorRequest = index.openCursor(rango);  
  
cursorRequest.onsuccess = (event) => {  
 const cursor = event.target.result;  
 if (cursor) {  
 console.log("Resultado:", cursor.value);  
 cursor.continue(); // Continua con el siguiente  
 } else {  
 console.log("No hay más registros");  
 }  
};

**3. Uso de rangos (IDBKeyRange)**

Los rangos son muy útiles para búsquedas avanzadas y pueden combinarse con índices y cursores:

* **only(valor)**: Busca una clave exacta.
* **lowerBound(valor)**: Busca desde el valor especificado hasta el final.
* **upperBound(valor)**: Busca desde el inicio hasta el valor especificado.
* **bound(inicio, fin, excluyeInicio, excluyeFin)**: Busca entre dos valores (inclusivos o exclusivos).

Ejemplo: Buscar nombres en un rango alfabético.

const rango = IDBKeyRange.bound("A", "M"); // Buscar entre A y M  
const request = index.openCursor(rango);  
  
request.onsuccess = (event) => {  
 const cursor = event.target.result;  
 if (cursor) {  
 console.log("Nombre:", cursor.key, "Registro:", cursor.value);  
 cursor.continue();  
 }  
};

**4. Mejores prácticas**

* **Usa índices para búsquedas rápidas:** Son más eficientes que recorrer todo el almacén.
* **Valida los resultados:** Siempre verifica si los datos obtenidos son válidos (null o undefined).
* **Maneja errores:** Siempre define un onerror en cada operación para capturar problemas.
* **Filtra con rangos:** Reduce la cantidad de datos procesados utilizando IDBKeyRange.

Con estas técnicas, puedes realizar búsquedas flexibles y eficientes en un almacén de objetos de **IndexedDB**.

1. **Borrar valores de un almacén de objetos**

Para borrar valores de un almacén de objetos en **IndexedDB**, puedes usar los métodos provistos por el objeto IDBObjectStore. Los más comunes son:

**1. Métodos para borrar valores**

Hay tres métodos principales para eliminar datos de un almacén:

**a. delete()**

* Borra un único registro por su clave.
* Si no se encuentra la clave, no hace nada.

**Ejemplo: Borrar un registro específico**

const transaction = db.transaction("miAlmacen", "readwrite"); // Transacción de lectura/escritura  
const objectStore = transaction.objectStore("miAlmacen");  
  
const request = objectStore.delete(1); // Borra el registro con clave 1  
  
request.onsuccess = () => {  
 console.log("Registro borrado con éxito");  
};  
  
request.onerror = (event) => {  
 console.error("Error al borrar el registro:", event.target.error);  
};

**b. clear()**

* Borra **todos los registros** del almacén de objetos.
* Esta operación elimina todos los datos sin necesidad de especificar una clave o rango.

**Ejemplo: Borrar todos los registros**

const transaction = db.transaction("miAlmacen", "readwrite");  
const objectStore = transaction.objectStore("miAlmacen");  
  
const request = objectStore.clear(); // Limpia todo el almacén  
  
request.onsuccess = () => {  
 console.log("Todos los registros fueron borrados");  
};  
  
request.onerror = (event) => {  
 console.error("Error al limpiar el almacén:", event.target.error);  
};

**c. Usar un cursor para borrar registros seleccionados**

* Cuando necesitas eliminar registros que cumplen ciertos criterios, puedes usar un cursor para recorrer los datos y llamar al método delete() en cada elemento que quieras borrar.

**Ejemplo: Borrar registros que cumplen una condición**

const transaction = db.transaction("miAlmacen", "readwrite");  
const objectStore = transaction.objectStore("miAlmacen");  
  
const cursorRequest = objectStore.openCursor();  
  
cursorRequest.onsuccess = (event) => {  
 const cursor = event.target.result;  
 if (cursor) {  
 const valor = cursor.value;  
  
 // Condición: Borrar registros donde el campo "activo" es false  
 if (!valor.activo) {  
 const deleteRequest = cursor.delete(); // Borra el registro actual  
 deleteRequest.onsuccess = () => {  
 console.log("Registro borrado:", valor);  
 };  
 }  
  
 cursor.continue(); // Continúa con el siguiente registro  
 } else {  
 console.log("Se recorrieron todos los registros");  
 }  
};  
  
cursorRequest.onerror = (event) => {  
 console.error("Error al usar el cursor:", event.target.error);  
};

**2. Borrar datos en un rango**

Si solo necesitas eliminar registros dentro de un rango específico de claves, puedes combinar IDBKeyRange con el método delete().

**Ejemplo: Borrar registros en un rango**

const transaction = db.transaction("miAlmacen", "readwrite");  
const objectStore = transaction.objectStore("miAlmacen");  
  
// Definir un rango: Claves entre 10 y 20  
const rango = IDBKeyRange.bound(10, 20);  
  
const request = objectStore.delete(rango);  
  
request.onsuccess = () => {  
 console.log("Registros en el rango fueron borrados");  
};  
  
request.onerror = (event) => {  
 console.error("Error al borrar registros en el rango:", event.target.error);  
};

**3. Pasos generales para borrar valores**

1. **Iniciar una transacción:**
   1. Necesitas una transacción con permisos de lectura/escritura (readwrite) para modificar los datos.
2. **Acceder al almacén:**
   1. Usa transaction.objectStore("nombreAlmacen") para obtener el almacén de objetos.
3. **Realizar la operación de borrado:**
   1. Usa delete(), clear() o un cursor para borrar los datos.
4. **Manejar eventos:**
   1. Siempre define onsuccess y onerror para controlar los resultados y posibles errores.

**4. Manejo de errores**

Asegúrate de manejar correctamente los errores al borrar datos:

* **Clave inexistente:** Si la clave no existe en el almacén, delete() no arrojará error; simplemente no hará nada.
* **Transacción fallida:** Si algo falla en la transacción (como un error de escritura), aparecerá en transaction.onerror.

**5. Ejemplo completo**

Borrar todos los registros de un almacén y luego agregar un mensaje de confirmación:

const request = indexedDB.open("MiBaseDeDatos", 1);  
  
request.onsuccess = (event) => {  
 const db = event.target.result;  
  
 const transaction = db.transaction("miAlmacen", "readwrite");  
 const objectStore = transaction.objectStore("miAlmacen");  
  
 const clearRequest = objectStore.clear();  
  
 clearRequest.onsuccess = () => {  
 console.log("Todos los registros fueron borrados exitosamente");  
 };  
  
 clearRequest.onerror = (event) => {  
 console.error("Error al limpiar el almacén:", event.target.error);  
 };  
};  
  
request.onerror = (event) => {  
 console.error("Error al abrir la base de datos:", event.target.error);  
};

1. **Los cursores en IndexedDB**

Son herramientas que permiten recorrer y operar sobre múltiples registros de un almacén de objetos o índice de forma iterativa. Son especialmente útiles cuando necesitas procesar grandes volúmenes de datos, realizar búsquedas personalizadas o trabajar con rangos de claves.

**1. Qué es un cursor**

Un cursor es un objeto que te permite:

* **Recorrer registros** en un almacén de objetos o índice, uno por uno.
* Acceder a la **clave** (cursor.key) y al **valor** (cursor.value) de cada registro.
* Realizar operaciones como actualizar o eliminar datos mientras recorres los registros.

El cursor avanza usando el método cursor.continue().

**2. Crear y usar un cursor**

Para crear un cursor, utilizas uno de los siguientes métodos del objeto IDBObjectStore o IDBIndex:

* **openCursor()**: Recorre todos los registros de un almacén o índice.
* **openKeyCursor()**: Recorre únicamente las claves, sin cargar los valores asociados.

**Ejemplo básico: Recorrer registros con openCursor()**

const transaction = db.transaction("miAlmacen", "readonly");  
const objectStore = transaction.objectStore("miAlmacen");  
  
const cursorRequest = objectStore.openCursor();  
  
cursorRequest.onsuccess = (event) => {  
 const cursor = event.target.result;  
  
 if (cursor) {  
 console.log("Clave:", cursor.key);  
 console.log("Valor:", cursor.value);  
  
 cursor.continue(); // Avanza al siguiente registro  
 } else {  
 console.log("Se recorrieron todos los registros.");  
 }  
};  
  
cursorRequest.onerror = (event) => {  
 console.error("Error al abrir el cursor:", event.target.error);  
};

**3. Avanzar con el cursor**

El método cursor.continue() mueve el cursor al siguiente registro disponible.

**Ejemplo: Avanzar manualmente**

cursorRequest.onsuccess = (event) => {  
 const cursor = event.target.result;  
  
 if (cursor) {  
 console.log("Procesando registro:", cursor.value);  
  
 // Avanzar manualmente si la clave es menor a 50  
 if (cursor.key < 50) {  
 cursor.continue();  
 } else {  
 console.log("Se detuvo el recorrido en la clave 50.");  
 }  
 }  
};

**4. Usar un rango con el cursor**

Puedes limitar el conjunto de registros que el cursor recorrerá utilizando IDBKeyRange.

**Ejemplo: Recorrer registros en un rango**

const rango = IDBKeyRange.bound(10, 20); // Claves entre 10 y 20  
  
const cursorRequest = objectStore.openCursor(rango);  
  
cursorRequest.onsuccess = (event) => {  
 const cursor = event.target.result;  
  
 if (cursor) {  
 console.log("Clave:", cursor.key, "Valor:", cursor.value);  
  
 cursor.continue(); // Avanza al siguiente registro dentro del rango  
 } else {  
 console.log("Se recorrieron todos los registros en el rango.");  
 }  
};

**5. Modificar datos con el cursor**

Puedes usar el método cursor.update(valor) para actualizar un registro mientras lo recorres.

**Ejemplo: Actualizar un registro**

const cursorRequest = objectStore.openCursor();  
  
cursorRequest.onsuccess = (event) => {  
 const cursor = event.target.result;  
  
 if (cursor) {  
 if (cursor.value.activo === false) {  
 // Modificar el campo "activo" a true  
 const registroActualizado = { ...cursor.value, activo: true };  
  
 const updateRequest = cursor.update(registroActualizado);  
 updateRequest.onsuccess = () => {  
 console.log("Registro actualizado:", registroActualizado);  
 };  
 }  
  
 cursor.continue(); // Avanza al siguiente registro  
 }  
};

**6. Borrar registros con el cursor**

Puedes usar el método cursor.delete() para eliminar un registro mientras recorres los datos.

**Ejemplo: Borrar registros que cumplan una condición**

const cursorRequest = objectStore.openCursor();  
  
cursorRequest.onsuccess = (event) => {  
 const cursor = event.target.result;  
  
 if (cursor) {  
 if (!cursor.value.activo) {  
 const deleteRequest = cursor.delete();  
 deleteRequest.onsuccess = () => {  
 console.log("Registro borrado:", cursor.value);  
 };  
 }  
  
 cursor.continue(); // Avanza al siguiente registro  
 }  
};

**7. Tipos de rangos (IDBKeyRange)**

Puedes definir diferentes tipos de rangos para controlar qué registros recorre el cursor.

|  |  |
| --- | --- |
| Método | Descripción |
| IDBKeyRange.only(valor) | Crea un rango para una clave específica. |
| IDBKeyRange.lowerBound(valor, true) | Crea un rango desde un límite inferior. |
| IDBKeyRange.upperBound(valor, true) | Crea un rango hasta un límite superior. |
| IDBKeyRange.bound(valor1, valor2) | Crea un rango entre dos límites (incluye ambos). |

**Ejemplo: Usar diferentes rangos**

// Claves mayores o iguales a 10  
const rango1 = IDBKeyRange.lowerBound(10);  
  
// Claves menores a 50  
const rango2 = IDBKeyRange.upperBound(50, true);  
  
// Claves entre 20 y 30, excluyendo 30  
const rango3 = IDBKeyRange.bound(20, 30, false, true);

**8. Beneficios del cursor**

* **Eficiencia:** Los cursores cargan registros individualmente, lo que reduce la memoria usada.
* **Flexibilidad:** Puedes realizar operaciones personalizadas sobre cada registro.
* **Control total:** Puedes usar rangos y detener el recorrido en cualquier momento.

**9. Ejemplo completo**

Recorrer todos los registros, imprimir los activos y eliminarlos si no lo son:

const transaction = db.transaction("miAlmacen", "readwrite");  
const objectStore = transaction.objectStore("miAlmacen");  
  
const cursorRequest = objectStore.openCursor();  
  
cursorRequest.onsuccess = (event) => {  
 const cursor = event.target.result;  
  
 if (cursor) {  
 const registro = cursor.value;  
  
 if (registro.activo) {  
 console.log("Registro activo encontrado:", registro);  
 } else {  
 // Borrar registros no activos  
 const deleteRequest = cursor.delete();  
 deleteRequest.onsuccess = () => {  
 console.log("Registro no activo borrado:", registro);  
 };  
 }  
  
 cursor.continue(); // Avanzar al siguiente registro  
 } else {  
 console.log("Todos los registros fueron procesados.");  
 }  
};  
  
cursorRequest.onerror = (event) => {  
 console.error("Error al procesar los registros:", event.target.error);  
};

Con este enfoque, puedes realizar búsquedas, modificaciones o eliminaciones eficientes en **IndexedDB**. ¿Te gustaría un ejemplo más específico?