mdn web docs

> Esta página ha sido traducida del inglés por la comunidad. Aprende más y únete a la comunidad de MDN Web Docs.

Usando IndexedDB

IndexedDB es una manera de almacenar datos dentro del navegador del usuario. Debido a que permite la creación de aplicaciones con habilidades de consulta enriquecidas, con independencia de la disponibilidad de la red, sus aplicaciones pueden trabajar tanto en línea como fuera de línea.

Acerca de este documento

Este tutorial es una guía sobre el uso de la API asíncrona de IndexedDB. Si no está familiarizado con IndexedDB, por favor consulte primero Conceptos Básicos Acerca de IndexedDB.

Para la documentación de referencia sobre la API de IndexedDB, vea el artículo IndexedDB y sus subpaginas, que documentan los tipos de objetos usados por IndexedDB, así como los métodos síncronos y asíncronos.

Patrones Básicos

El patrón básico que indexedDB propone es:

1. Abrir una base de datos.
2. Crear un objeto de almacenamiento en la base de datos.
3. Iniciar una transacción y hacer una petición para hacer alguna operación de la base de datos, tal como añadir o recuperar datos.

4. Espere a que se complete la operación por la escucha de la clase correcta de eventos DOM .
5. Hacer algo con el resultado (El cual puede ser encontrado en el objeto de la petición).

Con esos grandes rasgos en mente, seremos más concretos.

Creando y estructurando el almacenamiento

Como las especificaciones están todavía elaborandose, las implementaciones actuales de indexedDB dependen de los navegadores. Hasta que la especificación se haya consolidado, los proveedores de navegadores pueden tener diferentes implementaciones de los estandares de indexedDB. Una vez se alcance el consenso en el estandar, los proveedores implementarán la API sin prefijos. En algunas implementaciones ya fueron removidos los prefijos: Internet Explorer 10, Firefox 16, Chrome 24. Cuando utilizan un prefijo, los navegadores basados en gecko usan el prefijo `moz` , mientras que los navegadores basados en WebKit usan el prefijo `webkit` .

Usando una versión experimental de IndexedDB

En caso que usted quiera probar su código en navegadores que todavía usen un prefijo, puede usar el siguiente codigo:

JS

```
// En la siguiente línea, puede incluir prefijos de implementación que quiera
probar.
window.indexedDB =
  window.indexedDB ||
  window.mozIndexedDB ||
  window.webkitIndexedDB ||
  window.msIndexedDB;
// No use "var indexedDB = ..." Si no está en una función.
// Por otra parte, puedes necesitar referencias a algun objeto window.IDB*:
window.IDBTransaction =
  window.IDBTransaction ||
  window.webkitIDBTransaction ||
  window.msIDBTransaction;
window.IDBKeyRange =
```

```
  window.IDBKeyRange || window.webkitIDBKeyRange || window.msIDBKeyRange;
// (Mozilla nunca ha prefijado estos objetos, por lo tanto no necesitamos
window.mozIDB*)
```

Hay que tener cuidado con las implementaciones que usan un prefijo ya que puede ser inestables, incompletas, o usen una versión antigua de la especificación. En producción se recomienda usar el código sin prefijos. Es preferible no tener soporte para un navegador a decir que lo tiene y fallar en ello :

```
  JS
if (!window.indexedDB) {
  window.alert(
    "Su navegador no soporta una versión estable de indexedDB. Tal y como las
características no serán validas",
  );
}
```

Abriendo una base de datos

Iniciamos todo el proceso así:

```
  JS
// dejamos abierta nuestra base de datos
var request = window.indexedDB.open("MyTestDatabase", 3);
```

¿Lo has visto? Abrir una base de datos es igual que cualquier otra operación — solo tienes que "solicitarla" (request).

La solicitud de apertura no abre la base de datos o inicia la transacción de inmediato. La llamada a la función `open()` retornan unos objetos `IDBOpenDBRequest`, cuyo resultado, correcto o erróneo, se maneja en un evento. Alguna otra función asincrónica en indexedDB hace lo mismo - Devolver un objeto `IDBRequest` (inglés) que disparará un evento con el resultado o el error. El resultado para la función de abrir es una instancia de un `IDBDatabase`.

El segundo parámetro para el método open es la versión de la base de datos. La versión de la base de datos determina el esquema - El almacen de objectos en la base de datos y su estructura. Si la base de datos no existe, es creada y se

dispara un evento `onupgradeneeded` de inmediato, permitiéndote proveer una actualización de la estructura e índices en la función que capture dicho evento. Se verá más adelante en Actualizando la versión de la base de datos.

> **Advertencia: Importante**: El número de versión es un `unsigned long`. Por lo tanto significa que puede ser un entero muy grande. También significa que si usas un flotante será convertido en un entero más cercano y la transacción puede no ser iniciada, el evento `upgradeneeded` no se desencadenará. Por ejemplo no use 2.4 como un número de versión ya que será igual que la 2:
>
> ```
> JS
> var request = indexedDB.open("MyTestDatabase", 2.4); // Esto no se hace, la versión será redondeada a 2
> ```

Generando manipuladores

La primera cosa que usted querrá hacer con la totalidad de las peticiones que usted genera es agregar controladores de éxito y de error:

```
JS
request.onerror = function (event) {
  // Hacer algo con request.errorCode!
};
request.onsuccess = function (event) {
  // Hacer algo con request.result!
};
```

¿Cuál de las dos funciones, onSuccess () o onerror (), se vuelve a llamar? Si todo tiene éxito, un evento de éxito (es decir, un evento DOM cuya propiedad tipo se establece en el "éxito") se dispara con la solicitud como su objetivo. Una vez que se dispara, la función onSuccess () a petición se activa con el evento de éxito como argumento. De lo contrario, si había algún problema, un evento de error (es decir, un evento DOM cuyo tipo de propiedad se establece en "error") se dispara a petición. Esto desencadena la función onerror () con el evento de error como argumento.

La API IndexedDB está diseñada para minimizar la necesidad de control de errores, por lo que no es probable que veamos muchos eventos de error (al menos, no una vez que estás acostumbrado a la API). En el caso de la apertura de una base de datos, sin embargo, hay algunas condiciones comunes que generan eventos de error. El problema más común se produce cuando el usuario ha decidido no dar, a su aplicación web, el permiso para crear una base de datos. Uno de los principales objetivos de diseño de IndexedDB es permitir que grandes cantidades de datos se almacenen para su uso sin conexión a internet. (Para obtener más información sobre la cantidad de almacenamiento que puede tener para cada navegador, consulte Límites de almacenamiento.)

Obviamente, los navegadores no permitirán que alguna red de publicidad o sitio web malicioso pueda contaminar su ordenador, por ello los navegadores utilizan un diálogo para indicar al usuario la primera vez que cualquier aplicación web determinada intente abrir una IndexedDB para el almacenamiento. El usuario puede optar por permitir o denegar el acceso. Además, el almacenamiento IndexedDB en los modos de privacidad navegadores sólo dura en memoria hasta que la sesión de incógnito haya sido cerrada (modo de navegación privada para el modo de Firefox e Incognito para Chrome, pero en Firefox no está implementado a partir de noviembre 2015 por lo que no puede utilizar IndexedDB en Firefox navegación privada en absoluto).

Ahora, asumiendo que el usuario acepta su solicitud para crear una base de datos, y que ha recibido un evento de éxito para activar la devolución de llamada de éxito; ¿Que sigue? La solicitud aquí se generó con una llamada a indexedDB.open (), por lo request.result es una instancia de IDBDatabase, y que sin duda quieren ahorrar para más adelante. Su código podría ser algo como esto:

JS

```
var db;
var request = indexedDB.open("MyTestDatabase");
request.onerror = function (event) {
  alert("Why didn't you allow my web app to use IndexedDB?!");
};
request.onsuccess = function (event) {
  db = request.result;
};
```

Manejando errores

Como se mencionó anteriormente, los eventos de error de burbujas. Eventos de error se dirigen a la solicitud que generó el error, entonces el evento se propaga a la operación, y finalmente con el objeto de base de datos. Si desea evitar la adición de controladores de errores a cada solicitud, en su lugar puede añadir un solo controlador de errores en el objeto de base de datos, así:

JS

```
db.onerror = function (event) {
  // Generic error handler for all errors targeted at this database's
  // requests!
  alert("Database error: " + event.target.errorCode);
};
```

Uno de los errores más comunes posibles al abrir una base de datos es `VER_ERR`. Indica que la versión de la base de datos almacenada en el disco es mayor que la versión que está intentando abrir. Este es un caso de error que siempre debe ser manejado por el gestor de errores.

Creación o actualización de la versión de la base de datos

Cuando se crea una nueva base de datos o se aumenta el número de versión de una base de datos existente (mediante la especificación de un número de versión más alto de lo que hizo antes, en Cómo abrir una base de datos), el evento onupgradeneeded se activará y un objeto IDBVersionChangeEvent (inglés) será pasado a cualquier controlador de eventos `onversionchange` establecido en `request.result` (es decir, db en el ejemplo). En el controlador para el evento `upgradeneeded`, se debe crear los almacenes de objetos necesarios para esta versión de la base de datos:

JS

```
// Este evento solamente está implementado en navegadores recientes
request.onupgradeneeded = function (event) {
  var db = event.target.result;

  // Crea un almacén de objetos (objectStore) para esta base de datos
  var objectStore = db.createObjectStore("name", { keyPath: "myKey" });
};
```

En este caso, la base de datos ya tendrá los almacenes de objetos de la versión anterior de la base de datos, por lo que no tiene que crear estos almacenes de objetos de nuevo. Sólo es necesario crear nuevos almacenes de objetos, o eliminar las tiendas de objetos de la versión anterior que ya no son necesarios. Si necesita cambiar un almacén de objetos existentes (por ejemplo, para cambiar la ruta de acceso clave `keyPath`), entonces se debe eliminar el antiguo almacén de objetos y crear de nuevo con las nuevas opciones. (Tenga en cuenta que esto borrará la información en el almacén de objetos Si usted necesita guardar esa información, usted debe leerlo y guardarlo en otro lugar antes de actualizar la base de datos.)

Tratar de crear un almacén de objetos con un nombre que ya existe (o tratando de eliminar un almacén de objetos con un nombre que no existe) lanzará un error.

Si el evento `onupgradeneeded` retorna éxito, entonces se activará el manejador `onsuccess` de la solicitud de base de datos abierta.

Blink / Webkit soportan la versión actual de la especificación, tal como fue liberado en Chrome 23+ y Opera 17+ ; IE10+ también lo soporta. Implementaciones mas viejas o distintas no implementan la versión actual de la especificación, y por lo tanto no son compatibles todavía con el `indexedDB.open(nombre, versión).onupgradeneeded`. Para obtener más información sobre cómo actualizar la versión de la base de datos en Webkit/Blink mas viejos, consulte el artículo de referencia IDBDatabase.

Estructuración de la base de datos

Ahora para estructurar la base de datos. IndexedDB usa almacenes de datos (object stores) en lugar de tablas, y una base de datos puede contener cualquier número de almacenes. Cuando un valor es almacenado, se le asocia con una clave. Existen diversas maneras en que una clave pude ser indicada dependiendo de si el almacén usa una ruta de clave o generador.

La siguiente table muetra las distintas formas en que las claves pueden ser indicadas:

Ruta de clave(`keyPath`)	**Generador de clave (`autoIncrement`)**	**Descripción**
No	No	Este almacén puede contener cualquier tipo de valor, incluso valores primitivos como números y cadenas. Se debe indicar un argumento de clave distinto cada vez que se agregue un nuevo valor.
Si	No	Este almacén de objetos solo puede contener objetos de JavaScript. Los objetos deben tener una propiedad con el mismo nombre que la ruta de clave.
No	Si	Este objeto puede contener cualquier tipo de valor. La clave es generada automáticamente, o se puede indicar un argumento de clave distinto si se quiere usar una clave específica.
Si	Si	Este almacén de objetos solo puede contener objetos de JavaScript. Usualmente una clave es generada y dicho valor es almacenado en el objeto en una propiedad con el mismo nombre que la ruta de clave. Sin embargo, si dicha propiedad ya existe en el objeto, el valor de esa propuiedad es usado como clave en lugar de generar una nueva.

También se puede crear índices en cualquer almacén de objetos, siempre y cuando el almacén contenga objets, y no primitivos. Un índice permite buscar valores contenidos en el almacén usando el valor de una propiedad del objeto almacenado, en lugar de la clave del mismo.

Adicionalmente, los índices tienen la habilidad para hacer cumplir restricciones simples en los datos almacendos. Al indicar la bandera `unique` al crear el índice, el índice asegurará que no se puedan almacenar dos objetos que tengan el mismo valor para la clave del índice. Así, por ejemplo si se tiene un almacén de objetos que contiene un set de personas, y se desea asegurar que no existan dos personas con el mismo email, se puede usar un índice con la bandera `unique` activada para forzar esto.

Esto puede sonar confuso, pero un ejemplo simple debe ilustrar el concepto. Primero, definiremos alguna información de clientes para usar en nuestro ejemplo:

JS

```
// Así se ve nuestra información de clientes.
const customerData = [
  { ssn: "444-44-4444", name: "Bill", age: 35, email: "bill@company.com" },
  { ssn: "555-55-5555", name: "Donna", age: 32, email: "donna@home.org" },
];
```

Ahora, creemos una IndexedDB para almacenar los datos:

JS

```
const dbName = "the_name";

var request = indexedDB.open(dbName, 2);

request.onerror = function (event) {
  // Manejar errores.
};
request.onupgradeneeded = function (event) {
  var db = event.target.result;

  // Se crea un almacén para contener la información de nuestros cliente
  // Se usará "ssn" como clave ya que es garantizado que es única
  var objectStore = db.createObjectStore("customers", { keyPath: "ssn" });

  // Se crea un índice para buscar clientes por nombre. Se podrían tener duplicados
  // por lo que no se puede usar un índice único.
  objectStore.createIndex("name", "name", { unique: false });

  // Se crea un índice para buscar clientespor email. Se quiere asegurar que
  // no puedan haberdos clientes con el mismo email, asi que se usa un índice único.
  objectStore.createIndex("email", "email", { unique: true });

  // Se usa transaction.oncomplete para asegurarse que la creación del almacén
  // haya finalizado antes de añadir los datos en el.
  objectStore.transaction.oncomplete = function (event) {
    // Guarda los datos en el almacén recién creado.
    var customerObjectStore = db
      .transaction("customers", "readwrite")
      .objectStore("customers");
```

```
    for (var i in customerData) {
      customerObjectStore.add(customerData[i]);
    }
  };
};
```

Como se indicó previamente, `onupgradeneeded` es el único lugar donde se puede alterar la estructura de la base de datos. En el, se puede crear y borrar almacenes de objetos y construir y remover índices.

Los almacenes de datos son creados con una llamada a `createObjectStore()`. El método toma como parámetros el nombre del almacén y un objeto. A pesar de que el segundo parámetro es opcional, es muy importante, porque permite definir propiedades opcionales importantes y refinar el tipo de almacén que se desea crear. En este caso, se pregunta por un almacén llamado "customers" y se define la clave, que es la propiedad que indica que un objeto en el almacén es único. La propiedad en este ejemplo es "ssn" (Social Security Number) ya que los números de seguridad social está garantizado que sea único. "ssn" debe estar presente en cada objeto que se guarda al almacén.

También se solicitó crear un índice llamado "name" que se fija en la propiedad `name` de los objetos almacenados. Así como con `createObjectStore()`, `createIndex()` toma un objeto opcional `options` que refina el tipo de índice que se desea crear. Agregar objetos que no tengan una propiedad `name` funcionará, pero los objetos no aparecerán en el índice "name"

Ahora se pueden obtener los clientes almacenados usando su `ssn` directamente del almacen, o usando su nombre a través del índice. Para aprender como hacer esto, vea la sección El uso de un índice

El uso de un generador de claves

Indicar la bandera `autoIncrement` cuando se crea el almacén habilitará el generador de claves para dicho almacén. Por defecto esta bandera no está marcada.

Con el generador de claves, la clave será generada automáticamente a medida que se agreguen valores al almacén. El número actual de un generador de claves siempre se establece en 1 cuando se creal el almacén por primera vez. Básicamente la nueva clave autogenerada es incrementada en 1 basada en la llave anterior. El numero actual para un generador de claves nunca disminuye, salvo como resultado de operaciones de base de datos que sean revertidos, por ejemplo, cuando la transacción de base de datos es abortada. Por lo tanto borrar un registro o incluso borrar todos los registros de un almacén nunca afecta al generador de claves

Se puede crear otro almacén de objetos con generador de claves como se muestra abajo:

JS

```
// Abrir la indexedDB.
var request = indexedDB.open(dbName, 3);

request.onupgradeneeded = function (event) {
  var db = event.target.result;

  // Create another object store called "names" with the autoIncrement flag set as
true.
  var objStore = db.createObjectStore("names", { autoIncrement: true });

  // Because the "names" object store has the key generator, the key for the name
value is generated automatically.
  // The added records would be like:
  // key : 1 => value : "Bill"
  // key : 2 => value : "Donna"
  for (var i in customerData) {
    objStore.add(customerData[i].name);
  }
};
```

Para más detalles acerca del generador de claves, por favor ver "W3C Key Generators" .

Añadir, recuperación y eliminación de datos

Antes que haga algo con su nueva base de datos , necesita comenzar una transacción. Transactions come from the database object, and you have to specify which object stores you want the transaction to span. Once you are inside the transaction, you can access the object stores that hold your data and make your requests. Next, you need to decide if you're going to make changes to the database or if you just need to read from it. Transactions have three available modes: `readonly` , `readwrite` , and `versionchange` .

To change the "schema" or structure of the database—which involves creating or deleting object stores or indexes—the transaction must be in `versionchange` mode. This transaction is opened by calling the `IDBFactory.open` (inglés) method with a `version` specified. (In WebKit browsers, which have not implemented the latest specifcation, the `IDBFactory.open` (inglés) method takes only one parameter, the `name` of the database; then you must call `IDBVersionChangeRequest.setVersion` to establish the `versionchange` transaction.)

To read the records of an existing object store, the transaction can either be in `readonly` or `readwrite` mode. To make changes to an existing object store, the transaction must be in `readwrite` mode. You open such transactions with `IDBDatabase.transaction` . The method accepts two parameters: the `storeNames` (the scope, defined as an array of object stores that you want to access) and the `mode` (`readonly` or `readwrite`) for the transaction. The method returns a transaction object containing the `IDBIndex.objectStore` (inglés) method, which you can use to access your object store. By default, where no mode is specified, transactions open in `readonly` mode.

You can speed up data access by using the right scope and mode in the transaction. Here are a couple of tips:

- When defining the scope, specify only the object stores you need. This way, you can run multiple transactions with non-overlapping scopes concurrently.
- Only specify a `readwrite` transaction mode when necessary. You can concurrently run multiple `readonly` transactions with overlapping scopes, but you can have only one `readwrite` transaction for an object store. To learn more, see the definition for *transactions* (inglés) in the Basic Concepts (inglés) article.

Agregar datos a la base de datos

If you've just created a database, then you probably want to write to it. Here's what that looks like:

```
JS
var transaction = db.transaction(["customers"], "readwrite");
// Note: Older experimental implementations use the deprecated constant
IDBTransaction.READ_WRITE instead of "readwrite".
// In case you want to support such an implementation, you can write:
// var transaction = db.transaction(["customers"], IDBTransaction.READ_WRITE);
```

The `transaction()` function takes two arguments (though one is optional) and returns a transaction object. The first argument is a list of object stores that the transaction will span. You can pass an empty array if you want the transaction to span all object stores, but don't do it because the spec says an empty array should generate an InvalidAccessError. If you don't specify anything for the second argument, you get a read-only transaction. Since you want to write to it here you need to pass the `"readwrite"` flag.

Now that you have a transaction you need to understand its lifetime. Transactions are tied very closely to the event loop. If you make a transaction and return to the event loop without using it then the transaction will become inactive. The only way to keep the transaction active is to make a request on it. When the request is finished you'll get a DOM event and, assuming that the request succeeded, you'll have another opportunity to extend the transaction during that callback. If you return to the event loop without extending the transaction then it will become inactive, and so on. As long as there are pending requests the transaction remains active. Transaction lifetimes are really very simple but it might take a little time to get used to. A few more examples will help, too. If you start seeing `TRANSACTION_INACTIVE_ERR` error codes then you've messed something up.

Transactions can receive DOM events of three different types: `error`, `abort`, and `complete`. We've talked about the way that `error` events bubble, so a transaction receives error events from any requests that are generated from it. A more subtle point here is that the default behavior of an error is to abort the transaction in which it occurred. Unless you handle the error by first calling `preventDefault()` on

the error event then doing something else, the entire transaction is rolled back. This design forces you to think about and handle errors, but you can always add a catchall error handler to the database if fine-grained error handling is too cumbersome. If you don't handle an error event or if you call `abort()` on the transaction, then the transaction is rolled back and an `abort` event is fired on the transaction. Otherwise, after all pending requests have completed, you'll get a `complete` event. If you're doing lots of database operations, then tracking the transaction rather than individual requests can certainly aid your sanity.

Now that you have a transaction, you'll need to get the object store from it. Transactions only let you have an object store that you specified when creating the transaction. Then you can add all the data you need.

JS

```
// Do something when all the data is added to the database.
transaction.oncomplete = function (event) {
  alert("All done!");
};

transaction.onerror = function (event) {
  // Don't forget to handle errors!
};

var objectStore = transaction.objectStore("customers");
for (var i in customerData) {
  var request = objectStore.add(customerData[i]);
  request.onsuccess = function (event) {
    // event.target.result == customerData[i].ssn;
  };
}
```

The `result` of a request generated from a call to `add()` is the key of the value that was added. So in this case, it should equal the `ssn` property of the object that was added, since the object store uses the `ssn` property for the key path. Note that the `add()` function requires that no object already be in the database with the same key. If you're trying to modify an existing entry, or you don't care if one exists already, you can use the `put()` function, as shown below in the Updating an entry in the database section.

Extracción de datos de la base de datos

Removing data is very similar:

JS

```
var request = db
  .transaction(["customers"], "readwrite")
  .objectStore("customers")
  .delete("444-44-4444");
request.onsuccess = function (event) {
  // It's gone!
};
```

Obtener datos de la base de datos

Now that the database has some info in it, you can retrieve it in several ways. First, the simple `get()`. You need to provide the key to retrieve the value, like so:

JS

```
var transaction = db.transaction(["customers"]);
var objectStore = transaction.objectStore("customers");
var request = objectStore.get("444-44-4444");
request.onerror = function (event) {
  // Handle errors!
};
request.onsuccess = function (event) {
  // Do something with the request.result!
  alert("Name for SSN 444-44-4444 is " + request.result.name);
};
```

That's a lot of code for a "simple" retrieval. Here's how you can shorten it up a bit, assuming that you handle errors at the database level:

JS

```
db
  .transaction("customers")
  .objectStore("customers")
  .get("444-44-4444").onsuccess = function (event) {
  alert("Name for SSN 444-44-4444 is " + event.target.result.name);
};
```

See how this works? Since there's only one object store, you can avoid passing a list of object stores you need in your transaction and just pass the name as a string. Also, you're only reading from the database, so you don't need a `"readwrite"` transaction. Calling `transaction()` with no mode specified gives you a `"readonly"` transaction. Another subtlety here is that you don't actually save the request object to a variable. Since the DOM event has the request as its target you can use the event to get to the `result` property.

> **Nota:** You can speed up data access by limiting the scope and mode in the transaction. Here are a couple of tips:
>
> - When defining the scope, specify only the object stores you need. This way, you can run multiple transactions with non-overlapping scopes concurrently.
> - Only specify a `readwrite` transaction mode when necessary. You can concurrently run multiple `readonly` transactions with overlapping scopes, but you can have only one `readwrite` transaction for an object store. To learn more, see the definition for *transactions* in the Basic Concepts article (inglés).

Actualización de una entrada en la base de datos

Now we've retrieved some data, updating it and inserting it back into the IndexedDB is pretty simple. Let's update the previous example somewhat:

JS

```
var objectStore = db
  .transaction(["customers"], "readwrite")
  .objectStore("customers");
var request = objectStore.get("444-44-4444");
request.onerror = function (event) {
  // Handle errors!
};
request.onsuccess = function (event) {
  // Get the old value that we want to update
  var data = request.result;

  // update the value(s) in the object that you want to change
```

```
  data.age = 42;

  // Put this updated object back into the database.
  var requestUpdate = objectStore.put(data);
  requestUpdate.onerror = function (event) {
    // Do something with the error
  };
  requestUpdate.onsuccess = function (event) {
    // Success - the data is updated!
  };
};
```

So here we're creating an `objectStore` and requesting a customer record out of it, identified by its ssn value (`444-44-4444`). We then put the result of that request in a variable (`data`), update the `age` property of this object, then create a second request (`requestUpdate`) to put the customer record back into the `objectStore` , overwriting the previous value.

> **Nota:** That in this case we've had to specify a `readwrite` transaction because we want to write to the database, not just read out of it.

El uso de un cursor

Using `get()` requires that you know which key you want to retrieve. If you want to step through all the values in your object store, then you can use a cursor. Here's what it looks like:

JS

```
var objectStore = db.transaction("customers").objectStore("customers");

objectStore.openCursor().onsuccess = function (event) {
  var cursor = event.target.result;
  if (cursor) {
    alert("Name for SSN " + cursor.key + " is " + cursor.value.name);
    cursor.continue();
  } else {
    alert("No more entries!");
  }
};
```

The `openCursor()` function takes several arguments. First, you can limit the range of items that are retrieved by using a key range object that we'll get to in a minute. Second, you can specify the direction that you want to iterate. In the above example, we're iterating over all objects in ascending order. The success callback for cursors is a little special. The cursor object itself is the `result` of the request (above we're using the shorthand, so it's `event.target.result`). Then the actual key and value can be found on the `key` and `value` properties of the cursor object. If you want to keep going, then you have to call `continue()` on the cursor. When you've reached the end of the data (or if there were no entries that matched your `openCursor()` request) you still get a success callback, but the `result` property is `undefined`.

One common pattern with cursors is to retrieve all objects in an object store and add them to an array, like this:

JS

```
var customers = [];

objectStore.openCursor().onsuccess = function (event) {
  var cursor = event.target.result;
  if (cursor) {
    customers.push(cursor.value);
    cursor.continue();
  } else {
    alert("Got all customers: " + customers);
  }
};
```

> **Nota:** Note: Mozilla has also implemented `getAll()` to handle this case (and `getAllKeys()`, which is currently hidden behind the `dom.indexedDB.experimental` preference in about:config). these aren't part of the IndexedDB standard, so may disappear in the future. We've included them because we think they're useful. The following code does precisely the same thing as above:
>
> JS
>
> ```
> objectStore.getAll().onsuccess = function (event) {
>   alert("Got all customers: " + event.target.result);
> };
> ```

> There is a performance cost associated with looking at the `value` property of a cursor, because the object is created lazily. When you use `getAll()` for example, Gecko must create all the objects at once. If you're just interested in looking at each of the keys, for instance, it is much more efficient to use a cursor than to use `getAll()`. If you're trying to get an array of all the objects in an object store, though, use `getAll()`.

El uso de un índice

Storing customer data using the SSN as a key is logical since the SSN uniquely identifies an individual. (Whether this is a good idea for privacy is a different question, and outside the scope of this article.) If you need to look up a customer by name, however, you'll need to iterate over every SSN in the database until you find the right one. Searching in this fashion would be very slow, so instead you can use an index.

JS

```
var index = objectStore.index("name");
index.get("Donna").onsuccess = function (event) {
  alert("Donna's SSN is " + event.target.result.ssn);
};
```

The "name" cursor isn't unique, so there could be more than one entry with the `name` set to `"Donna"`. In that case you always get the one with the lowest key value.

If you need to access all the entries with a given `name` you can use a cursor. You can open two different types of cursors on indexes. A normal cursor maps the index property to the object in the object store. A key cursor maps the index property to the key used to store the object in the object store. The differences are illustrated here:

JS

```
// Using a normal cursor to grab whole customer record objects
index.openCursor().onsuccess = function (event) {
  var cursor = event.target.result;
  if (cursor) {
```

```
    // cursor.key is a name, like "Bill", and cursor.value is the whole object.
    alert(
      `Name: ${cursor.key}, SSN: ${cursor.value.ssn}, email: ${cursor.value.email}`,
    );
    cursor.continue();
  }
};

// Using a key cursor to grab customer record object keys
index.openKeyCursor().onsuccess = function (event) {
  var cursor = event.target.result;
  if (cursor) {
    // cursor.key is a name, like "Bill", and cursor.value is the SSN.
    // No way to directly get the rest of the stored object.
    alert("Name: " + cursor.key + ", SSN: " + cursor.value);
    cursor.continue();
  }
};
```

Especificación de la gama y la dirección de los cursores

If you would like to limit the range of values you see in a cursor, you can use an `IDBKeyRange` object and pass it as the first argument to `openCursor()` or `openKeyCursor()`. You can make a key range that only allows a single key, or one that has a lower or upper bound, or one that has both a lower and upper bound. The bound may be "closed" (i.e., the key range includes the given value(s)) or "open" (i.e., the key range does not include the given value(s)). Here's how it works:

JS

```
// Only match "Donna"
var singleKeyRange = IDBKeyRange.only("Donna");

// Match anything past "Bill", including "Bill"
var lowerBoundKeyRange = IDBKeyRange.lowerBound("Bill");

// Match anything past "Bill", but don't include "Bill"
var lowerBoundOpenKeyRange = IDBKeyRange.lowerBound("Bill", true);

// Match anything up to, but not including, "Donna"
var upperBoundOpenKeyRange = IDBKeyRange.upperBound("Donna", true);
```

```
// Match anything between "Bill" and "Donna", but not including "Donna"
var boundKeyRange = IDBKeyRange.bound("Bill", "Donna", false, true);

// To use one of the key ranges, pass it in as the first argument of
openCursor()/openKeyCursor()
index.openCursor(boundKeyRange).onsuccess = function (event) {
  var cursor = event.target.result;
  if (cursor) {
    // Do something with the matches.
    cursor.continue();
  }
};
```

Sometimes you may want to iterate in descending order rather than in ascending order (the default direction for all cursors). Switching direction is accomplished by passing `prev` to the `openCursor()` function as the second argument:

JS

```
objectStore.openCursor(boundKeyRange, "prev").onsuccess = function (event) {
  var cursor = event.target.result;
  if (cursor) {
    // Do something with the entries.
    cursor.continue();
  }
};
```

If you just want to specify a change of direction but not constrain the results shown, you can just pass in null as the first argument:

JS

```
objectStore.openCursor(null, "prev").onsuccess = function (event) {
  var cursor = event.target.result;
  if (cursor) {
    // Do something with the entries.
    cursor.continue();
  }
};
```

Since the "name" index isn't unique, there might be multiple entries where `name` is the same. Note that such a situation cannot occur with object stores since the key

must always be unique. If you wish to filter out duplicates during cursor iteration over indexes, you can pass `nextunique` (or `prevunique` if you're going backwards) as the direction parameter. When `nextunique` or `prevunique` is used, the entry with the lowest key is always the one returned.

JS

```
index.openKeyCursor(null, "nextunique").onsuccess = function (event) {
  var cursor = event.target.result;
  if (cursor) {
    // Do something with the entries.
    cursor.continue();
  }
};
```

Please see "IDBCursor Constants" for the valid direction arguments.

Cambios Versión mientras que una aplicación web está abierto en otra pestaña

When your web app changes in such a way that a version change is required for your database, you need to consider what happens if the user has the old version of your app open in one tab and then loads the new version of your app in another. When you call `open()` with a greater version than the actual version of the database, all other open databases must explicitly acknowledge the request before you can start making changes to the database (an `onblocked` event is fired until tey are closed or reloaded). Here's how it works:

JS

```
var openReq = mozIndexedDB.open("MyTestDatabase", 2);

openReq.onblocked = function (event) {
  // If some other tab is loaded with the database, then it needs to be closed
  // before we can proceed.
  alert("Please close all other tabs with this site open!");
};

openReq.onupgradeneeded = function (event) {
  // All other databases have been closed. Set everything up.
  db.createObjectStore(/* ... */);
  useDatabase(db);
```

```
};

openReq.onsuccess = function (event) {
  var db = event.target.result;
  useDatabase(db);
  return;
};

function useDatabase(db) {
  // Make sure to add a handler to be notified if another page requests a version
  // change. We must close the database. This allows the other page to upgrade the
database.
  // If you don't do this then the upgrade won't happen until the user closes the
tab.
  db.onversionchange = function (event) {
    db.close();
    alert("A new version of this page is ready. Please reload!");
  };

  // Do stuff with the database.
}
```

Seguridad

IndexedDB uses the same-origin principle, which means that it ties the store to the origin of the site that creates it (typically, this is the site domain or subdomain), so it cannot be accessed by any other origin.

It's important to note that IndexedDB doesn't work for content loaded into a frame from another site (either `<frame>` or `<iframe>`. This is a security and privacy measure and can be considered analogous the blocking of third-party cookies. For more details, see Error 595307 en Firefox .

Warning About Browser Shutdown

When the browser shuts down (e.g., when the user selects Exit or clicks the Close button), any pending IndexedDB transactions are (silently) aborted — they will not complete, and they will not trigger the error handler. Since the user can exit the browser at any time, this means that you cannot rely upon any particular transaction to complete or to know that it did not complete. There are several implications of this behavior.

First, you should take care to always leave your database in a consistent state at the end of every transaction. For example, suppose that you are using IndexedDB to store a list of items that you allow the user to edit. You save the list after the edit by clearing the object store and then writing out the new list. If you clear the object store in one transaction and write the new list in another transaction, there is a danger that the browser will close after the clear but before the write, leaving you with an empty database. To avoid this, you should combine the clear and the write into a single transaction.

Second, you should never tie database transactions to unload events. If the unload event is triggered by the browser closing, any transactions created in the unload event handler will never complete. An intuitive approach to maintaining some information across browser sessions is to read it from the database when the browser (or a particular page) is opened, update it as the user interacts with the browser, and then save it to the database when the browser (or page) closes. However, this will not work. The database transactions will be created in the unload event handler, but because they are asynchronous they will be aborted before they can execute.

In fact, there is no way to guarantee that IndexedDB transactions will complete, even with normal browser shutdown. See Error 870645 en Firefox .

Full IndexedDB example

HTML Content

HTML

```
<script
  type="text/javascript"
  src="https://ajax.googleapis.com/ajax/libs/jquery/1.8.3/jquery.min.js"></script>

<h1>IndexedDB Demo: storing blobs, e-publication example</h1>
<div class="note">
  <p>Works and tested with:</p>
  <div id="compat"></div>
</div>

<div id="msg"></div>
```

```
<form id="register-form">
  <table>
    <tbody>
      <tr>
        <td>
          <label for="pub-title" class="required"> Title: </label>
        </td>
        <td>
          <input type="text" id="pub-title" name="pub-title" />
        </td>
      </tr>
      <tr>
        <td>
          <label for="pub-biblioid" class="required">
            Bibliographic ID:<br />
            <span class="note">(ISBN, ISSN, etc.)</span>
          </label>
        </td>
        <td>
          <input type="text" id="pub-biblioid" name="pub-biblioid" />
        </td>
      </tr>
      <tr>
        <td>
          <label for="pub-year"> Year: </label>
        </td>
        <td>
          <input type="number" id="pub-year" name="pub-year" />
        </td>
      </tr>
    </tbody>
    <tbody>
      <tr>
        <td>
          <label for="pub-file"> File image: </label>
        </td>
        <td>
          <input type="file" id="pub-file" />
        </td>
      </tr>
      <tr>
        <td>
          <label for="pub-file-url">
            Online-file image URL:<br />
```

```
            <span class="note">(same origin URL)</span>
          </label>
        </td>
        <td>
          <input type="text" id="pub-file-url" name="pub-file-url" />
        </td>
      </tr>
    </tbody>
  </table>

  <div class="button-pane">
    <input type="button" id="add-button" value="Add Publication" />
    <input type="reset" id="register-form-reset" />
  </div>
</form>

<form id="delete-form">
  <table>
    <tbody>
      <tr>
        <td>
          <label for="pub-biblioid-to-delete">
            Bibliographic ID:<br />
            <span class="note">(ISBN, ISSN, etc.)</span>
          </label>
        </td>
        <td>
          <input
            type="text"
            id="pub-biblioid-to-delete"
            name="pub-biblioid-to-delete" />
        </td>
      </tr>
      <tr>
        <td>
          <label for="key-to-delete">
            Key:<br />
            <span class="note">(for example 1, 2, 3, etc.)</span>
          </label>
        </td>
        <td>
          <input type="text" id="key-to-delete" name="key-to-delete" />
        </td>
      </tr>
```

```
    </tbody>
  </table>
  <div class="button-pane">
    <input type="button" id="delete-button" value="Delete Publication" />
    <input
      type="button"
      id="clear-store-button"
      value="Clear the whole store"
      class="destructive" />
  </div>
</form>

<form id="search-form">
  <div class="button-pane">
    <input
      type="button"
      id="search-list-button"
      value="List database content" />
  </div>
</form>

<div>
  <div id="pub-msg"></div>
  <div id="pub-viewer"></div>
  <ul id="pub-list"></ul>
</div>
```

CSS Content

CSS

```
body {
  font-size: 0.8em;
  font-family: Sans-Serif;
}

form {
  background-color: #cccccc;
  border-radius: 0.3em;
  display: inline-block;
  margin-bottom: 0.5em;
  padding: 1em;
}

table {
```

```
  border-collapse: collapse;
}

input {
  padding: 0.3em;
  border-color: #cccccc;
  border-radius: 0.3em;
}

.required:after {
  content: "*";
  color: red;
}

.button-pane {
  margin-top: 1em;
}

#pub-viewer {
  float: right;
  width: 48%;
  height: 20em;
  border: solid #d092ff 0.1em;
}
#pub-viewer iframe {
  width: 100%;
  height: 100%;
}

#pub-list {
  width: 46%;
  background-color: #eeeeee;
  border-radius: 0.3em;
}
#pub-list li {
  padding-top: 0.5em;
  padding-bottom: 0.5em;
  padding-right: 0.5em;
}

#msg {
  margin-bottom: 1em;
}
```

```
.action-success {
  padding: 0.5em;
  color: #00d21e;
  background-color: #eeeeee;
  border-radius: 0.2em;
}

.action-failure {
  padding: 0.5em;
  color: #ff1408;
  background-color: #eeeeee;
  border-radius: 0.2em;
}

.note {
  font-size: smaller;
}

.destructive {
  background-color: orange;
}
.destructive:hover {
  background-color: #ff8000;
}
.destructive:active {
  background-color: red;
}
```

JavaScript Content

JS

```
(function () {
  var COMPAT_ENVS = [
    ["Firefox", ">= 16.0"],
    [
      "Google Chrome",
      ">= 24.0 (you may need to get Google Chrome Canary), NO Blob storage support",
    ],
  ];
  var compat = $("#compat");
  compat.empty();
  compat.append('<ul id="compat-list"></ul>');
  COMPAT_ENVS.forEach(function (val, idx, array) {
    $("#compat-list").append("<li>" + val[0] + ": " + val[1] + "</li>");
```

```
});

const DB_NAME = "mdn-demo-indexeddb-epublications";
const DB_VERSION = 1; // Use a long long for this value (don't use a float)
const DB_STORE_NAME = "publications";

var db;

// Used to keep track of which view is displayed to avoid uselessly reloading it
var current_view_pub_key;

function openDb() {
  console.log("openDb ...");
  var req = indexedDB.open(DB_NAME, DB_VERSION);
  req.onsuccess = function (evt) {
    // Better use "this" than "req" to get the result to avoid problems with
    // garbage collection.
    // db = req.result;
    db = this.result;
    console.log("openDb DONE");
  };
  req.onerror = function (evt) {
    console.error("openDb:", evt.target.errorCode);
  };

  req.onupgradeneeded = function (evt) {
    console.log("openDb.onupgradeneeded");
    var store = evt.currentTarget.result.createObjectStore(DB_STORE_NAME, {
      keyPath: "id",
      autoIncrement: true,
    });

    store.createIndex("biblioid", "biblioid", { unique: true });
    store.createIndex("title", "title", { unique: false });
    store.createIndex("year", "year", { unique: false });
  };
}

/**
 * @param {string} store_name
 * @param {string} mode either "readonly" or "readwrite"
 */
function getObjectStore(store_name, mode) {
  var tx = db.transaction(store_name, mode);
```

```
  return tx.objectStore(store_name);
}

function clearObjectStore(store_name) {
  var store = getObjectStore(DB_STORE_NAME, "readwrite");
  var req = store.clear();
  req.onsuccess = function (evt) {
    displayActionSuccess("Store cleared");
    displayPubList(store);
  };
  req.onerror = function (evt) {
    console.error("clearObjectStore:", evt.target.errorCode);
    displayActionFailure(this.error);
  };
}

function getBlob(key, store, success_callback) {
  var req = store.get(key);
  req.onsuccess = function (evt) {
    var value = evt.target.result;
    if (value) success_callback(value.blob);
  };
}

/**
 * @param {IDBObjectStore=} store
 */
function displayPubList(store) {
  console.log("displayPubList");

  if (typeof store == "undefined")
    store = getObjectStore(DB_STORE_NAME, "readonly");

  var pub_msg = $("#pub-msg");
  pub_msg.empty();
  var pub_list = $("#pub-list");
  pub_list.empty();
  // Resetting the iframe so that it doesn't display previous content
  newViewerFrame();

  var req;
  req = store.count();
  // Requests are executed in the order in which they were made against the
  // transaction, and their results are returned in the same order.
```

```
// Thus the count text below will be displayed before the actual pub list
// (not that it is algorithmically important in this case).
req.onsuccess = function (evt) {
  pub_msg.append(
    "<p>There are <strong>" +
      evt.target.result +
      "</strong> record(s) in the object store.</p>",
  );
};
req.onerror = function (evt) {
  console.error("add error", this.error);
  displayActionFailure(this.error);
};

var i = 0;
req = store.openCursor();
req.onsuccess = function (evt) {
  var cursor = evt.target.result;

  // If the cursor is pointing at something, ask for the data
  if (cursor) {
    console.log("displayPubList cursor:", cursor);
    req = store.get(cursor.key);
    req.onsuccess = function (evt) {
      var value = evt.target.result;
      var list_item = $(
        `<li>[${cursor.key}] (biblioid: ${value.biblioid}) ${value.title}</li>`,
      );
      if (value.year != null) list_item.append(" - " + value.year);

      if (
        value.hasOwnProperty("blob") &&
        typeof value.blob != "undefined"
      ) {
        var link = $('<a href="' + cursor.key + '">File</a>');
        link.on("click", function () {
          return false;
        });
        link.on("mouseenter", function (evt) {
          setInViewer(evt.target.getAttribute("href"));
        });
        list_item.append(" / ");
        list_item.append(link);
      } else {
```

```
            list_item.append(" / No attached file");
          }
          pub_list.append(list_item);
        };

        // Move on to the next object in store
        cursor.continue();

        // This counter serves only to create distinct ids
        i++;
      } else {
        console.log("No more entries");
      }
    };
  }

  function newViewerFrame() {
    var viewer = $("#pub-viewer");
    viewer.empty();
    var iframe = $("<iframe />");
    viewer.append(iframe);
    return iframe;
  }

  function setInViewer(key) {
    console.log("setInViewer:", arguments);
    key = Number(key);
    if (key == current_view_pub_key) return;

    current_view_pub_key = key;

    var store = getObjectStore(DB_STORE_NAME, "readonly");
    getBlob(key, store, function (blob) {
      console.log("setInViewer blob:", blob);
      var iframe = newViewerFrame();

      // It is not possible to set a direct link to the
      // blob to provide a mean to directly download it.
      if (blob.type == "text/html") {
        var reader = new FileReader();
        reader.onload = function (evt) {
          var html = evt.target.result;
          iframe.load(function () {
            $(this).contents().find("html").html(html);
```

```
        });
      };
      reader.readAsText(blob);
    } else if (blob.type.indexOf("image/") == 0) {
      iframe.load(function () {
        var img_id = "image-" + key;
        var img = $('<img id="' + img_id + '"/>');
        $(this).contents().find("body").html(img);
        var obj_url = window.URL.createObjectURL(blob);
        $(this)
          .contents()
          .find("#" + img_id)
          .attr("src", obj_url);
        window.URL.revokeObjectURL(obj_url);
      });
    } else if (blob.type == "application/pdf") {
      $("*").css("cursor", "wait");
      var obj_url = window.URL.createObjectURL(blob);
      iframe.load(function () {
        $("*").css("cursor", "auto");
      });
      iframe.attr("src", obj_url);
      window.URL.revokeObjectURL(obj_url);
    } else {
      iframe.load(function () {
        $(this).contents().find("body").html("No view available");
      });
    }
  });
}

/**
 * @param {string} biblioid
 * @param {string} title
 * @param {number} year
 * @param {string} url the URL of the image to download and store in the local
 *   IndexedDB database. The resource behind this URL is subjected to the
 *   "Same origin policy", thus for this method to work, the URL must come from
 *   the same origin as the web site/app this code is deployed on.
 */
function addPublicationFromUrl(biblioid, title, year, url) {
  console.log("addPublicationFromUrl:", arguments);

  var xhr = new XMLHttpRequest();
```

```
    xhr.open("GET", url, true);
    // Setting the wanted responseType to "blob"
    // http://www.w3.org/TR/XMLHttpRequest2/#the-response-attribute
    xhr.responseType = "blob";
    xhr.onload = function (evt) {
      if (xhr.status == 200) {
        console.log("Blob retrieved");
        var blob = xhr.response;
        console.log("Blob:", blob);
        addPublication(biblioid, title, year, blob);
      } else {
        console.error(
          "addPublicationFromUrl error:",
          xhr.responseText,
          xhr.status,
        );
      }
    };
    xhr.send();

    // We can't use jQuery here because as of jQuery 1.8.3 the new "blob"
    // responseType is not handled.
    // http://bugs.jquery.com/ticket/11461
    // http://bugs.jquery.com/ticket/7248
    // $.ajax({
    //   url: url,
    //   type: 'GET',
    //   xhrFields: { responseType: 'blob' },
    //   success: function(data, textStatus, jqXHR) {
    //     console.log("Blob retrieved");
    //     console.log("Blob:", data);
    //     // addPublication(biblioid, title, year, data);
    //   },
    //   error: function(jqXHR, textStatus, errorThrown) {
    //     console.error(errorThrown);
    //     displayActionFailure("Error during blob retrieval");
    //   }
    // });
  }

  /**
   * @param {string} biblioid
   * @param {string} title
   * @param {number} year
```

```
 * @param {Blob=} blob
 */
function addPublication(biblioid, title, year, blob) {
  console.log("addPublication arguments:", arguments);
  var obj = { biblioid: biblioid, title: title, year: year };
  if (typeof blob != "undefined") obj.blob = blob;

  var store = getObjectStore(DB_STORE_NAME, "readwrite");
  var req;
  try {
    req = store.add(obj);
  } catch (e) {
    if (e.name == "DataCloneError")
      displayActionFailure(
        "This engine doesn't know how to clone a Blob, " + "use Firefox",
      );
    throw e;
  }
  req.onsuccess = function (evt) {
    console.log("Insertion in DB successful");
    displayActionSuccess();
    displayPubList(store);
  };
  req.onerror = function () {
    console.error("addPublication error", this.error);
    displayActionFailure(this.error);
  };
}

/**
 * @param {string} biblioid
 */
function deletePublicationFromBib(biblioid) {
  console.log("deletePublication:", arguments);
  var store = getObjectStore(DB_STORE_NAME, "readwrite");
  var req = store.index("biblioid");
  req.get(biblioid).onsuccess = function (evt) {
    if (typeof evt.target.result == "undefined") {
      displayActionFailure("No matching record found");
      return;
    }
    deletePublication(evt.target.result.id, store);
  };
  req.onerror = function (evt) {
```

```
      console.error("deletePublicationFromBib:", evt.target.errorCode);
    };
  }

  /**
   * @param {number} key
   * @param {IDBObjectStore=} store
   */
  function deletePublication(key, store) {
    console.log("deletePublication:", arguments);

    if (typeof store == "undefined")
      store = getObjectStore(DB_STORE_NAME, "readwrite");

    // As per spec http://www.w3.org/TR/IndexedDB/#object-store-deletion-operation
    // the result of the Object Store Deletion Operation algorithm is
    // undefined, so it's not possible to know if some records were actually
    // deleted by looking at the request result.
    var req = store.get(key);
    req.onsuccess = function (evt) {
      var record = evt.target.result;
      console.log("record:", record);
      if (typeof record == "undefined") {
        displayActionFailure("No matching record found");
        return;
      }
      // Warning: The exact same key used for creation needs to be passed for
      // the deletion. If the key was a Number for creation, then it needs to
      // be a Number for deletion.
      req = store.delete(key);
      req.onsuccess = function (evt) {
        console.log("evt:", evt);
        console.log("evt.target:", evt.target);
        console.log("evt.target.result:", evt.target.result);
        console.log("delete successful");
        displayActionSuccess("Deletion successful");
        displayPubList(store);
      };
      req.onerror = function (evt) {
        console.error("deletePublication:", evt.target.errorCode);
      };
    };
    req.onerror = function (evt) {
      console.error("deletePublication:", evt.target.errorCode);
```

```
  };
}

function displayActionSuccess(msg) {
  msg = typeof msg != "undefined" ? "Success: " + msg : "Success";
  $("#msg").html('<span class="action-success">' + msg + "</span>");
}
function displayActionFailure(msg) {
  msg = typeof msg != "undefined" ? "Failure: " + msg : "Failure";
  $("#msg").html('<span class="action-failure">' + msg + "</span>");
}
function resetActionStatus() {
  console.log("resetActionStatus ...");
  $("#msg").empty();
  console.log("resetActionStatus DONE");
}

function addEventListeners() {
  console.log("addEventListeners");

  $("#register-form-reset").click(function (evt) {
    resetActionStatus();
  });

  $("#add-button").click(function (evt) {
    console.log("add ...");
    var title = $("#pub-title").val();
    var biblioid = $("#pub-biblioid").val();
    if (!title || !biblioid) {
      displayActionFailure("Required field(s) missing");
      return;
    }
    var year = $("#pub-year").val();
    if (year != "") {
      // Better use Number.isInteger if the engine has EcmaScript 6
      if (isNaN(year)) {
        displayActionFailure("Invalid year");
        return;
      }
      year = Number(year);
    } else {
      year = null;
    }
```

```
    var file_input = $("#pub-file");
    var selected_file = file_input.get(0).files[0];
    console.log("selected_file:", selected_file);
    // Keeping a reference on how to reset the file input in the UI once we
    // have its value, but instead of doing that we rather use a "reset" type
    // input in the HTML form.
    //file_input.val(null);
    var file_url = $("#pub-file-url").val();
    if (selected_file) {
      addPublication(biblioid, title, year, selected_file);
    } else if (file_url) {
      addPublicationFromUrl(biblioid, title, year, file_url);
    } else {
      addPublication(biblioid, title, year);
    }
  });

  $("#delete-button").click(function (evt) {
    console.log("delete ...");
    var biblioid = $("#pub-biblioid-to-delete").val();
    var key = $("#key-to-delete").val();

    if (biblioid != "") {
      deletePublicationFromBib(biblioid);
    } else if (key != "") {
      // Better use Number.isInteger if the engine has EcmaScript 6
      if (key == "" || isNaN(key)) {
        displayActionFailure("Invalid key");
        return;
      }
      key = Number(key);
      deletePublication(key);
    }
  });

  $("#clear-store-button").click(function (evt) {
    clearObjectStore();
  });

  var search_button = $("#search-list-button");
  search_button.click(function (evt) {
    displayPubList();
  });
}
```

```
  openDb();
  addEventListeners();
})(); // Immediately-Invoked Function Expression (IIFE)
```

[Test the online live demo]

Next step

If you want to start tinkering with the API, jump in to the reference documentation and check out the different methods.

See also

Reference

- IndexedDB API Reference
- Indexed Database API Specification
- Using IndexedDB in chrome

Tutorials

- A simple TODO list using HTML5 IndexedDB .

> **Nota:** This tutorial is based on an old version of the specification and does not work on up-to-date browsers - it still uses the removed `setVersion()` method.

- Databinding UI Elements with IndexedDB

Related articles

- IndexedDB — The Store in Your Browser

Firefox

- Mozilla interface files

Help improve MDN

Was this page helpful to you?

Yes No

Learn how to contribute.

This page was last modified on 17 dic 2024 by MDN contributors.