Capítulo 10
API Web Storage

10.1 Dos sistemas de almacenamiento

La Web fue primero pensada como una forma de mostrar información, solo mostrarla. El procesamiento de información comenzó luego, primero con aplicaciones del lado del servidor y más tarde, de forma bastante ineficiente, a través de pequeños códigos y complementos (plug-ins) ejecutados en el ordenador del usuario. Sin embargo, la esencia de la Web siguió siendo básicamente la misma: la información era preparada en el servidor y luego mostrada a los usuarios. El trabajo duro se desarrollaba casi completamente del lado del servidor porque el sistema no aprovechaba los recursos en los ordenadores de los usuarios.

HTML5 equilibra esta situación. Justificada por las particulares características de los dispositivos móviles, el surgimiento de los sistemas de computación en la nube, y la necesidad de estandarizar tecnologías e innovaciones introducidas por plug-ins a través de los últimos años, la especificación de HTML5 incluye herramientas que hacen posible construir y ejecutar aplicaciones completamente funcionales en el ordenador del usuario, incluso cuando no existe conexión a la red disponible.

Una de las características más necesitadas en cualquier aplicación es la posibilidad de almacenar datos para disponer de ellos cuando sean necesarios, pero no existía aún un mecanismo efectivo para este fin. Las llamadas "Cookies" (archivos de texto almacenados en el ordenador del usuario) fueron usadas por años para preservar información, pero debido a su naturaleza se encontraron siempre limitadas a pequeñas cadenas de texto, lo que las hacía útiles solo en determinadas circunstancias.

La API Web Storage es básicamente una mejora de las Cookies. Esta API nos permite almacenar datos en el disco duro del usuario y utilizarlos luego del mismo modo que lo haría una aplicación de escritorio. El proceso de almacenamiento provisto por esta API puede ser utilizado en dos situaciones particulares: cuando la información tiene que estar disponible solo durante la sesión en uso, y cuando tiene que ser preservada todo el tiempo que el usuario desee. Para hacer estos métodos más claros y comprensibles para los desarrolladores, la API fue dividida en dos partes llamadas **`sessionStorage`** y **`localStorage`**.

sessionStorage Este es un mecanismo de almacenamiento que conservará los datos disponible solo durante la duración de la sesión de una página. De hecho, a diferencia de sesiones reales, la información almacenada a través de este mecanismo es solo accesible desde una única ventana o pestaña y es preservada hasta que la ventana es cerrada. La especificación aún nombra "sesiones" debido a que la información es preservada incluso cuando la ventana es actualizada o una nueva página desde el mismo sitio web es cargada.

localStorage Este mecanismo trabaja de forma similar a un sistema de almacenamiento para aplicaciones de escritorio. Los datos son grabados de forma permanente y se encuentran siempre disponibles para la aplicación que los creó.

Ambos mecanismos trabajan a través de la misma interface, compartiendo los mismos métodos y propiedades. Y ambos son dependientes del origen, lo que quiere decir que la información está disponible solo a través del sitio web o la aplicación que los creó. Cada sitio web tendrá designado su propio espacio de almacenamiento que durará hasta que la ventana es cerrada o será permanente, de acuerdo al mecanismo utilizado.

La API claramente diferencia datos temporarios de permanentes, facilitando la construcción de pequeñas aplicaciones que necesitan preservar solo unas cadenas de texto como referencia temporaria (por ejemplo, carros de compra) o aplicaciones más grandes y complejas que necesitan almacenar documentos completos por todo el tiempo que sea necesario.

IMPORTANTE: Muchos navegadores solo trabajan de forma adecuada con esta API cuando la fuente es un servidor real. Para probar los siguientes códigos, le recomendamos que primero suba los archivos a su servidor.

10.2 La sessionStorage

Esta parte de la API, **`sessionStorage`**, es como un reemplazo para las Cookies de sesión. Las Cookies, así como **`sessionStorage`**, mantienen los datos disponibles durante un período específico de tiempo, pero mientras las Cookies de sesión usan el navegador como referencia, **`sessionStorage`** usa solo una simple ventana o pestaña. Esto significa que las Cookies creadas para una sesión estarán disponibles mientras el navegador continúe abierto, mientras que los datos creados con **`sessionStorage`** estarán solo disponibles mientras la ventana que los creó no es cerrada.

Implementación de un sistema de almacenamiento de datos

Debido a que ambos sistemas, **`sessionStorage`** y **`localStorage`**, trabajan con la misma interface, vamos a necesitar solo un documento HTML y un simple formulario para probar los códigos y experimentar con esta API:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Web Storage API</title>
  <link rel="stylesheet" href="storage.css">
  <script src="storage.js"></script>
</head>
<body>
  <section id="cajaformulario">
    <form name="formulario">
      <p>Clave:<br><input type="text" name="clave" id="clave"></p>
      <p>Valor:<br><textarea name="text" id="texto"></textarea></p>
      <p><input type="button" name="grabar" id="grabar"
                                              value="Grabar"></p>
    </form>
  </section>
  <section id="cajadatos">
    No hay información disponible
  </section>
</body>
</html>
```

Listado 10-1. *Plantilla para la API Storage*

También crearemos un grupo de reglas de estilo simples para dar forma a la página y diferenciar el área del formulario de la caja donde los datos serán mostrados y listados:

```
#cajaformulario{
  float: left;
  padding: 20px;
  border: 1px solid #999999;
}
#cajadatos{
  float: left;
  width: 400px;
  margin-left: 20px;
  padding: 20px;
  border: 1px solid #999999;
}
#clave, #texto{
  width: 200px;
}
#cajadatos > div{
  padding: 5px;
  border-bottom: 1px solid #999999;
}
```

Listado 10-2. *Estilos para nuestra plantilla.*

Hágalo usted mismo: Cree un archivo HTML con el código del Listado 10-1 y un archivo CSS llamado **`storage.css`** con los estilos del Listado 10-2. También necesitará crear un archivo llamado **`storage.js`** para grabar y probar los códigos Javascript presentados a continuación.

Creando datos

Ambos, **`sessionStorage`** y **`localStorage`**, almacenan datos como ítems. Los ítems están formados por un par clave/valor, y cada valor será convertido en una cadena de texto antes de ser almacenado. Piense en ítems como si fueran variables, con un nombre y un valor, que pueden ser creadas, modificadas o eliminadas.

Existen dos nuevos métodos específicos de esta API incluidos para crear y leer un valor en el espacio de almacenamiento:

setItem(clave, valor) Este es el método que tenemos que llamar para crear un ítem. El ítem será creado con una clave y un valor de acuerdo a los atributos especificados. Si ya existe un ítem con la misma clave, será actualizado al nuevo valor, por lo que este método puede utilizarse también para modificar datos previos.

getItem(clave) Para obtener el valor de un ítem, debemos llamar a este método especificando la clave del ítem que queremos leer. La clave en este caso es la misma que declaramos cuando creamos al ítem con **`setItem()`**.

```
function iniciar(){
  var boton=document.getElementById('grabar');
  boton.addEventListener('click', nuevoitem, false);
}
function nuevoitem(){
  var clave=document.getElementById('clave').value;
  var valor=document.getElementById('texto').value;
  sessionStorage.setItem(clave,valor);

  mostrar(clave);
}
function mostrar(clave){
  var cajadatos=document.getElementById('cajadatos');
  var valor=sessionStorage.getItem(clave);
  cajadatos.innerHTML='<div>'+clave+' - '+valor+'</div>';
}
window.addEventListener('load', iniciar, false);
```

Listado 10-3. *Almacenando y leyendo datos.*

El proceso es extremadamente simple. Los métodos son parte de **`sessionStorage`** y son llamados con la sintaxis **`sessionStorage.setItem()`**. En el código del Listado 10-3, la función **`nuevoitem()`** es ejecutada cada vez que el usuario hace clic en el botón del formulario. Esta función crea un ítem con la información insertada en los campos del formulario y luego llama a la función **`mostrar()`**. Esta última función lee el ítem de acuerdo a la clave recibida usando el método **`getItem()`** y muestra su valor en la pantalla.

Además de estos métodos, la API también ofrece una sintaxis abreviada para crear y leer ítems desde el espacio de almacenamiento. Podemos usar la clave del ítem como una propiedad y acceder a su valor de esta manera.

Este método usa en realidad dos tipos de sintaxis diferentes de acuerdo al tipo de información que estamos usando para crear el ítem. Podemos encerrar una variable representando la clave entre corchetes (por ejemplo, **`sessionStorage[clave]=valor`**) o podemos usar directamente el nombre de la propiedad (por ejemplo, **`sessionStorage.miitem=valor)`**.

```
function iniciar(){
  var boton=document.getElementById('grabar');
  boton.addEventListener('click', nuevoitem, false);
}
function nuevoitem(){
  var clave=document.getElementById('clave').value;
```

```
  var valor=document.getElementById('texto').value;
  sessionStorage[clave]=valor;

  mostrar(clave);
}
function mostrar(clave){
  var cajadatos=document.getElementById('cajadatos');
  var valor=sessionStorage[clave];
  cajadatos.innerHTML='<div>'+clave+' - '+valor+'</div>';
}
window.addEventListener('load', iniciar, false);
```

Listado 10-4. *Usando un atajo para trabajar con ítems.*

Leyendo datos

El anterior ejemplo solo lee el último ítem grabado. Vamos a mejorar este código aprovechando más métodos y propiedades provistos por la API con el propósito de manipular ítems:

length Esta propiedad retorna el número de ítems guardados por esta aplicación en el espacio de almacenamiento. Trabaja exactamente como la propiedad **`length`** usada normalmente en Javascript para procesar arrays, y es útil para lecturas secuenciales.

key(índice) Los ítems son almacenados secuencialmente, enumerados con un índice automático que comienzo por 0. Con este método podemos leer un ítem específico o crear un bucle para obtener toda la información almacenada.

```
function iniciar(){
  var boton=document.getElementById('grabar');
  boton.addEventListener('click', nuevoitem, false);
  mostrar();
}
function nuevoitem(){
  var clave=document.getElementById('clave').value;
  var valor=document.getElementById('texto').value;

  sessionStorage.setItem(clave,valor);
  mostrar();
  document.getElementById('clave').value='';
  document.getElementById('texto').value='';
}
function mostrar(){
  var cajadatos=document.getElementById('cajadatos');
  cajadatos.innerHTML='';
  for(var f=0;f<sessionStorage.length;f++){
    var clave=sessionStorage.key(f);
    var valor=sessionStorage.getItem(clave);
    cajadatos.innerHTML+='<div>'+clave+' - '+valor+'</div>';
  }
}
window.addEventListener('load', iniciar, false);
```

Listado 10-5. *Lstando ítems.*

El propósito del código en el Listado 10-5 es mostrar un listado completo de los ítems en la caja derecha de la pantalla. La función **`mostrar()`** fue mejorada usando la propiedad **`length`** y el método **`key()`**. Creamos un bucle **`for`** que va desde 0 al número de ítems que existen en el espacio de almacenamiento. Dentro del bucle, el método **`key()`** retornará la clave que nosotros definimos para cada ítem. Por ejemplo, si el ítem en la posición 0 del espacio de almacenamiento fue creado con la clave “miitem”, el código **`sessionStorage.key(0)`** retornará el valor “miitem”. Llamando a este método desde un bucle podemos listar todos los ítems en la pantalla con sus correspondientes claves y valores.

La función **`mostrar()`** es llamada desde la función **`iniciar()`** tan pronto como la aplicación es ejecutada. De este

modo podremos ver desde el comienzo los ítems que fueron grabados previamente en el espacio de almacenamiento.

Hágalo usted mismo: Aproveche los conceptos estudiados con la API Forms en el Capítulo 6 para controlar la validez de los campos del formulario y no permitir la inserción de ítems vacíos o inválidos.

Eliminando datos

Los ítems pueden ser creados, leídos y, por supuesto, eliminados. Es hora de ver cómo eliminar un ítem. La API ofrece dos métodos para este propósito:

removeItem(clave) Este método eliminará un ítem individual. La clave para identificar el ítem es la misma declarada cuando el ítem fue creado con el método **`setItem()`**.

clear() Este método vaciará el espacio de almacenamiento. Todos los ítems serán eliminados.

```
function iniciar(){
  var boton=document.getElementById('grabar');
  boton.addEventListener('click', nuevoitem, false);
  mostrar();
}
function nuevoitem(){
  var clave=document.getElementById('clave').value;
  var valor=document.getElementById('texto').value;

  sessionStorage.setItem(clave,valor);
  mostrar();
  document.getElementById('clave').value='';
  document.getElementById('texto').value='';
}
function mostrar(){
  var cajadatos=document.getElementById('cajadatos');
    cajadatos.innerHTML='<div><button
       onclick="eliminarTodo()">Eliminar Todo</button></div>';
  for(var f=0;f<sessionStorage.length;f++){
    var clave=sessionStorage.key(f);
    var valor=sessionStorage.getItem(clave);
    cajadatos.innerHTML+='<div>'+clave+' - '+valor+'<br><button
      onclick="eliminar(\''+clave+'\')">Eliminar</button></div>';
  }
}
function eliminar(clave){
  if(confirm('Está Seguro?')){
    sessionStorage.removeItem(clave);
    mostrar();
  }
}
function eliminarTodo(){
  if(confirm('Está Seguro?')){
    sessionStorage.clear();
    mostrar();
  }
}
window.addEventListener('load', iniciar, false);
```

Listado 10-6. *Eliminando ítems.*

Las funciones **`iniciar()`** y **`nuevoitem()`** en el Listado 10-6 son las mismas de códigos previos. Solo la función **`mostrar()`** cambia para incorporar el manejador de eventos **`onclick`** y llamar a las funciones que eliminarán un ítem individual o vaciarán el espacio de almacenamiento. La lista de ítems presentada en pantalla es construida de la misma manera que antes, pero esta vez un botón “Eliminar” es agregado junto a cada ítem para poder eliminarlo. Un botón para eliminar todos los ítems juntos también fue agregado en la parte superior.

Las funciones **`eliminar()`** y **`eliminarTodo()`** se encargan de eliminar el ítem seleccionado o limpiar el espacio de almacenamiento, respectivamente. Cada función llama a la función **`mostrar()`** al final para actualizar la lista de ítems en pantalla.

Hágalo usted mismo: Con el código del Listado 10-6, podrá estudiar cómo la información es procesada por **`sessionStorage`**. Abra la plantilla del Listado 10-1 en su navegador, cree nuevos ítems y luego abra la plantilla en una nueva ventana. La información en cada ventana es diferente. La vieja ventana mantendrá su información disponible y el espacio de almacenamiento de la nueva ventana estará vacío. A diferencia de otros sistemas (como Cookies de sesiones), para **`sessionStorage`** cada ventana es considerada una instancia diferente de la aplicación y la información de la sesión no se propaga entre ellas.

El sistema **`sessionStorage`** preserva los datos creados en una ventana solo hasta que esa ventana es cerrada. Es útil para controlar carros de compra o cualquier otra aplicación que requiere acceso a datos por períodos cortos de tiempo.

10.3 La localStorage

Disponer de un sistema confiable para almacenar datos durante la sesión de una ventana puede ser extremadamente útil en algunas circunstancias, pero cuando intentamos simular poderosas aplicaciones de escritorio en la web, un sistema de almacenamiento temporario no es suficiente.

Para cubrir este aspecto, Storage API ofrece un segundo sistema que reservará un espacio de almacenamiento para cada aplicación (cada origen) y mantendrá la información disponible permanentemente. Con **`localStorage`**, finalmente podemos grabar largas cantidades de datos y dejar que el usuario decida si la información es útil y debe ser conservada o no.

El sistema usa la misma interface que **`sessionStorage`**, debido a esto cada método y propiedad estudiado hasta el momento en este capítulo son también disponibles para **`localStorage`**. Solo la substitución del prefijo **`session`** por **`local`** es requerida para preparar los códigos.

```
   function iniciar(){
     var boton=document.getElementById('grabar');
     boton.addEventListener('click', nuevoitem, false);
     mostrar();
   }

function nuevoitem(){
     var clave=document.getElementById('clave').value;
     var valor=document.getElementById('texto').value;

     localStorage.setItem(clave,valor);
     mostrar();
     document.getElementById('clave').value='';
     document.getElementById('texto').value='';
   }
   function mostrar(){
     var cajadatos=document.getElementById('cajadatos');
     cajadatos.innerHTML='';
     for(var f=0;f<localStorage.length;f++){
       var clave=localStorage.key(f);
       var valor=localStorage.getItem(clave);
       cajadatos.innerHTML+='<div>'+clave+' - '+valor+'</div>';
     }
   }
   window.addEventListener('load', iniciar, false);
```

Listado 10-7. *Usando* `localStorage`.

En el Listado 10-7, simplemente reemplazamos **`sessionStorage`** por **`localStorage`** en el código de uno de los ejemplos anteriores. Ahora, cada ítem creado será preservado a través de diferentes ventanas e incluso luego de que todas las ventanas del navegador son cerradas.

> **Hágalo usted mismo:** Usando la plantilla del Listado 10-1, pruebe el código del Listado 10-7. Este código creará un nuevo ítem con la información insertada en el formulario y automáticamente listará todos los ítems disponibles en el espacio de almacenamiento reservado para esta aplicación. Cierre el navegador y abra el archivo HTML nuevamente. La información es preservada, por lo que podrá ver aún en pantalla todos los ítems ingresados previamente.

Evento storage

Debido a que **`localStorage`** hace que la información esté disponible en cada ventana donde la aplicación fue cargada, surgen al menos dos problemas: debemos resolver cómo estas ventanas se comunicarán entre sí y cómo haremos para mantener la información actualizada en cada una de ellas. En respuesta a ambos problemas, la especificación incluye el evento **`storage`**.

> **storage** Este evento será disparado por la ventana cada vez que un cambio ocurra en el espacio de almacenamiento. Puede ser usado para informar a cada ventana abierta con la misma aplicación que los datos han cambiado en el espacio de almacenamiento y que se debe hacer algo al respecto.

```
function iniciar(){
     var boton=document.getElementById('grabar');
     boton.addEventListener('click', nuevoitem, false);
     window.addEventListener("storage", mostrar, false);

     mostrar();
   }
   function nuevoitem(){
     var clave=document.getElementById('clave').value;
     var valor=document.getElementById('texto').value;

     localStorage.setItem(clave,valor);
     mostrar();
     document.getElementById('clave').value='';
     document.getElementById('texto').value='';
   }
   function mostrar(){
     var cajadatos=document.getElementById('cajadatos');
     cajadatos.innerHTML='';
     for(var f=0;f<localStorage.length;f++){
       var clave=localStorage.key(f);
       var valor=localStorage.getItem(clave);
       cajadatos.innerHTML+='<div>'+clave+' - '+valor+'</div>';
     }
   }
   window.addEventListener('load', iniciar, false);
```

Listado 10-8. *Escuchando al evento* `storage` *para mantener la lista de ítems actualizada.*

Solo necesitamos comenzar a escuchar al evento **`storage`** en la función **`iniciar()`** del código 10-8 para ejecutar la función **`mostrar()`** en cada ventana siempre que un ítem es creado, modificado o eliminado. Ahora, si algo cambia en una ventana, el cambio será mostrado automáticamente en el resto de las ventanas que están ejecutando la misma aplicación.

Espacio de almacenamiento

La información almacenada por **`localStorage`** será permanente a menos que el usuario decida que ya no la necesita. Esto significa que el espacio físico en el disco duro ocupado por esta información probablemente crecerá cada vez que la aplicación sea usada. Hasta este momento, la especificación de HTML5 recomienda a los fabricantes de navegadores que reserven un mínimo de 5 megabytes para cada origen (cada sitio web o aplicación).

Esta es solo una recomendación que probablemente cambiará dramáticamente en los próximos años. Algunos navegadores están consultando al usuario si expandir o no el espacio disponible cuando la aplicación lo necesita, pero usted debería ser consciente de esta limitación y tenerla en cuenta a la hora de desarrollar sus aplicaciones.

> **IMPORTANTE:** Muchos navegadores solo trabajan de forma adecuada con esta API cuando la fuente es un servidor real. Para probar los siguientes códigos, le recomendamos que primero suba los archivos a su servidor.

10.4 Referencia rápida

Con la ayuda de la API Web Storage, ahora las aplicaciones web pueden ofrecer un espacio de almacenamiento. Usando un par clave/valor, la información es almacenada en el ordenador del usuario para un rápido acceso o para trabajar desconectado de la red.

Tipo de almacenamiento

Dos mecanismos diferentes son ofrecidos para almacenar datos:

sessionStorage Este mecanismo mantiene la información almacenada solo disponible para una simple ventana y solo hasta que la ventana es cerrada.

localStorage Este mecanismo almacena datos de forma permanente. Estos datos son compartidos por todas las ventanas que están ejecutando la misma aplicación y estarán disponibles a menos que el usuario decida que ya no los necesita.

Métodos

La API incluye una interface común para cada mecanismo que cuenta con nuevos métodos, propiedades y eventos:

setItem(clave, valor) Este método crea un nuevo ítem que es almacenado en el espacio de almacenamiento reservado para la aplicación. El ítem está compuesto por un par clave/valor creado a partir de los atributos **`clave`** y **`valor`**.

getItem(clave) Este método lee el contenido de un ítem identificado por la clave especificada por el atributo **`clave`**. El valor de esta clave debe ser el mismo usado cuando el ítem fue creado con el método **`setItem()`**.

key(índice) Este método retorna la clave del ítem encontrado en la posición especificada por el atributo **`índice`** dentro del espacio de almacenamiento.

removeItem(clave) Este método elimina un ítem con la clave especificada por el atributo **`clave`**. El valor de esta clave debe ser el mismo usado cuando el ítem fue creado por el método **`setItem()`**.

clear() - Este método elimina todos los ítems en el espacio de almacenamiento reservado para la aplicación.

Propiedades

length Esta propiedad retorna el número de ítems disponibles en el espacio de almacenamiento reservado para la aplicación.

Eventos

storage Este evento es disparado cada vez que un cambio se produce en el espacio de almacenamiento reservado para la aplicación.

Capítulo 11
API IndexedDB

11.1 Una API de bajo nivel

La API estudiada en el capítulo anterior es útil para almacenamiento de pequeñas cantidades de datos, pero cuando se trata de grandes cantidades de datos estructurados debemos recurrir a un sistema de base de datos. La API IndexedDB es la solución provista por HTML5 a este respecto.

IndexedDB es un sistema de base de datos destinado a almacenar información indexada en el ordenador del usuario. Fue desarrollada como una API de bajo nivel con la intención de permitir un amplio espectro de usos. Esto la convierte en una de las API más poderosa de todas, pero también una de las más complejas. El objetivo fue proveer la estructura más básica posible para permitir a los desarrolladores construir librerías y crear interfaces de alto nivel para satisfacer necesidades específicas. En una API de bajo nivel como esta tenemos que hacernos cargo de cada aspecto y controlar las condiciones de cada proceso en toda operación realizada. El resultado es una API a la que la mayoría de los desarrolladores tardará en acostumbrarse y probablemente utilizará de forma indirecta a través de otras librerías populares construidas sobre ella que seguramente surgirán en un futuro cercano.

La estructura propuesta por IndexedDB es también diferente de SQL u otros sistemas de base de datos populares. La información es almacenada en la base de datos como objetos (registros) dentro de lo que es llamado Almacenes de Objetos (tablas). Los Almacenes de Objetos no tienen una estructura específica, solo un nombre e índices para poder encontrar los objetos en su interior. Estos objetos tampoco tienen una estructura definida, pueden ser diferentes unos de otros y tan complejos como necesitemos. La única condición para los objetos es que contengan al menos una propiedad declarada como índice para que sea posible encontrarlos en el Almacén de Objetos.

Base de datos

La base de datos misma es simple. Debido a que cada base de datos es asociada con un ordenador y un sitio web o aplicación, no existen usuarios para agregar o restricciones de acceso que debamos considerar. Solo necesitamos especificar el nombre y la versión, y la base de datos estará lista.

La interface declarada en la especificación para esta API provee el atributo **`indexedDB`** y el método **`open()`** para crear la base de datos. Este método retorna un objeto sobre el cual dos eventos serán disparados para indicar el éxito o los errores surgidos durante el proceso de creación de la base de datos.

El segundo aspecto que debemos considerar para crear o abrir una base de datos es la versión. La API requiere que una versión sea asignada a la base de datos. Esto es para preparar al sistema para futuras migraciones. Cuando tenemos que actualizar la estructura de una base de datos en el lado del servidor para agregar más tablas o índices, normalmente detenemos el servidor, migramos la información hacia la nueva estructura y luego encendemos el servidor nuevamente. Sin embargo, en este caso la información es contenida del lado del cliente y, por supuesto, el ordenador del usuario no puede ser apagado. Como resultado, la versión de la base de datos debe ser cambiada y luego la información migrada desde la vieja a la nueva versión.

Para trabajar con versiones de bases de datos, la API ofrece la propiedad **`version`** y el método **`setVersion()`**. La propiedad retorna el valor de la versión actual y el método asigna un nuevo valor de versión a la base de datos en uso. Este valor puede ser numérico o cualquier cadena de texto que se nos ocurra.

Objetos y Almacenes de Objetos

Lo que solemos llamar registros, en IndexedDB son llamados objetos. Estos objetos incluyen propiedades para almacenar e identificar valores. La cantidad de propiedades y cómo los objetos son estructurados es irrelevante. Solo deben incluir al menos una propiedad declarada como índice para poder encontrarlos dentro del Almacén de Objetos.

Los Almacenes de Objetos (tablas) tampoco tienen una estructura específica. Solo el nombre y uno o más índices deben ser declarados en el momento de su creación para poder encontrar objetos en su interior más tarde.

Figura 11-1. *Objetos con diferentes propiedades almacenados en un Almacén de Objetos.*

Como podemos ver en la Figura 11-1, un Almacén de Objetos contiene diversos objetos con diferentes propiedades. Algunos objetos tienen una propiedad **`DVD`**, otros tienen una propiedad **`Libro`**, etc… Cada uno tiene su propia estructura, pero todos deberán tener al menos una propiedad declarada como índice para poder ser encontrados. En el ejemplo de la Figura 11-1, este índice podría ser la propiedad **`Id`**.

Para trabajar con objetos y Almacenes de Objetos solo necesitamos crear el Almacén de Objetos, declarar las propiedades que serán usadas como índices y luego comenzar a almacenar objetos en este almacén. No necesitamos pensar acerca de la estructura o el contenido de los objetos en este momento, solo considerar los índices que vamos a utilizar para encontrarlos más adelante en el almacén.

La API provee varios métodos para manipular Almacenes de Objetos:

createObjectStore(nombre, clave, autoIncremento) Este método crea un nuevo Almacén de Objetos con el nombre y la configuración declarada por sus atributos. El atributo **`nombre`** es obligatorio. El atributo **`clave`** declarará un índice común para todos los objetos. Y el atributo **`autoIncremento`** es un valor booleano que determina si el Almacén de Objetos tendrá un generador de claves automático.

objectStore(nombre) Para acceder a los objetos en un Almacén de Objetos, una transacción debe ser iniciada y el Almacén de Objetos debe ser abierto para esa transacción. Este método abre el Almacén de Objetos con el nombre declarado por el atributo **`nombre`**.

deleteObjectStore(nombre) Este método destruye un Almacén de Objetos con el nombre declarado por el atributo **`nombre`**.

Los métodos **`createObjectStore()`** y **`deleteObjectStore()`**, así como otros métodos responsables de la configuración de la base de datos, pueden solo ser aplicados cuando la base de datos es creada o mejorada en una nueva versión.

Índices

Para encontrar objetos en un Almacén de Objetos necesitamos declarar algunas propiedades de estos objetos como índices. Una forma fácil de hacerlo es declarar el atributo **`clave`** en el método **`createObjectStore()`**. La propiedad declarada como **`clave`** será un índice común para cada objeto almacenado en ese Almacén de Objetos particular. Cuando declaramos el atributo **`clave,`** esta propiedad debe estar presente en todos los objetos.

Además del atributo **`clave`** podemos declarar todos los índices que necesitemos para un Almacén de Objetos usando métodos especiales provistos para este propósito:

createIndex(nombre, propiedad, único) Este método crea un índice para un Almacén de Objetos específico. El atributo **`nombre`** es un nombre con el que identificar al índice, el atributo **`propiedad`** es la propiedad que será usada como índice, y **`unique`** es un valor booleano que indica si existe la posibilidad de que dos o más objetos compartan el mismo valor para este índice.

index(nombre) Para usar un índice primero tenemos que crear una referencia al índice y luego asignar esta referencia a la transacción. El método **`index()`** crea esta referencia para el índice declarado en el atributo **`nombre`**.

deleteIndex(nombre) Si ya no necesitamos un índice podemos eliminarlo usando este método.

Transacciones

Un sistema de base de datos trabajando en un navegador debe contemplar algunas circunstancias únicas que no están presentes en otras plataformas. El navegador puede fallar, puede ser cerrado abruptamente, el proceso puede ser detenido por el usuario, o simplemente otro sitio web puede ser cargado en la misma ventana, por ejemplo. Existen muchas situaciones en las que trabajar directamente con la base de datos puede causar mal funcionamiento o incluso corrupción de datos. Para prevenir que algo así suceda, cada acción es realizada por medio de transacciones.

El método que genera una transacción se llama **`transaction().`** Para declarar el tipo de transacción, contamos con los siguientes tres atributos:

READ_ONLY Este atributo genera una transacción de solo lectura. Modificaciones no son permitidas.

READ_WRITE Usando este tipo de transacción podemos leer y escribir. Modificaciones son permitidas.

`VERSION_CHANGE` Este tipo de transacción es solo utilizada para actualizar la versión de la base de datos.

Las más comunes son las transacciones de lectura y escritura. Sin embargo, para prevenir un uso inadecuado, el tipo **READ_ONLY** (solo lectura) es configurado por defecto. Por este motivo, cuando solo necesitamos obtener información de la base de datos, lo único que debemos hacer es declarar el destino de la transacción (normalmente el nombre del Almacén de Objetos de donde vamos a leer esta información).

Métodos de Almacenes de Objetos

Para interactuar con Almacenes de Objetos, leer y almacenar información, la API provee varios métodos:

add(objeto) Este método recibe un par clave/valor o un objeto conteniendo varios pares clave/valor, y con los datos provistos genera un objeto que es agregado al Almacén de Objetos seleccionado. Si un objeto con el mismo valor de índice ya existe, el método **`add()`** retorna un error.

put(objeto) Este método es similar al anterior, excepto que si ya existe un objeto con el mismo valor de índice lo sobrescribe. Es útil para modificar un objeto ya almacenado en el Almacén de Objetos seleccionado.

get(clave) Podemos leer un objeto específico del Almacén de Objetos usando este método. El atributo **`clave`** es el valor del índice del objeto que queremos leer.

delete(clave) Para eliminar un objeto del Almacén de Objetos seleccionado solo tenemos que llamar a este método con el valor del índice del objeto a eliminar.

11.2 Implementando IndexedDB

¡Suficiente con la teoría! Es momento de crear nuestra primera base de datos y aplicar algunos de los métodos ya mencionados en este capítulo. Vamos a simular una aplicación para almacenar información sobre películas. Puede agregar a la base los datos que usted desee, pero para referencia, de aquí en adelante vamos a mencionar los siguientes:

id: tt0068646 **nombre:** El Padrino **fecha:** 1972

id: tt0086567 **nombre:** Juegos de Guerra **fecha:** 1983

id: tt0111161 **nombre:** Cadena Perpetua **fecha:** 1994

id: tt1285016 **nombre:** La Red Social **fecha:** 2010

IMPORTANTE: Los nombres de las propiedades (**`id`**, **`nombre`** y **`fecha`**) son los que vamos a utilizar para nuestros ejemplos en el resto del capítulo. La información fue recolectada del sitio web www.imdb.com, pero usted puede utilizar su propia lista o información al azar para probar los códigos.

Plantilla

Como siempre, necesitamos un documento HTML y algunos estilos CSS para crear las cajas en pantalla que contendrán el formulario apropiado y la información retornada. El formulario nos permitirá insertar nuevas películas dentro de la base de datos solicitándonos una clave, el título y el año en el que la película fue realizada.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>IndexedDB API</title>
  <link rel="stylesheet" href="indexed.css">
  <script src="indexed.js"></script>
</head>
<body>
  <section id="cajaformulario">
    <form name="formulario">
      <p>Clave:<br><input type="text" name="clave" id="clave"></p>
      <p>Título:<br><input type="text" name="texto" id="texto"></p>
      <p>Año:<br><input type="text" name="fecha" id="fecha"></p>
      <p><input type="button" name="grabar" id="grabar"
                                          value="Grabar"></p>
    </form>
  </section>
  <section id="cajadatos">
    No hay información disponible
  </section>
</body>
</html>
```

Listado 11-1. *Plantilla para IndexedDB API.*

Los estilos CSS definen las cajas para el formulario y cómo mostrar la información en pantalla:

```
#cajaformulario{
  float: left;
  padding: 20px;
  border: 1px solid #999999;
}
#cajadatos{
  float: left;
  width: 400px;
  margin-left: 20px;
  padding: 20px;
```

```
  border: 1px solid #999999;
}
#clave, #texto{
  width: 200px;
}
#cajadatos > div{
  padding: 5px;
  border-bottom: 1px solid #999999;
}
```

Listado 11-2. *Estilos para las cajas.*

> **Hágalo usted mismo:** Necesitará un archivo HTML para la plantilla del Listado 11-1, un archivo CSS llamado **`indexed.css`** para los estilos del Listado 11-2 y un archivo Javascript llamado **`indexed.js`** para introducir todos los códigos estudiados a continuación.

Abriendo la base de datos

Lo primero que debemos hacer en el código Javascript es abrir la base de datos. El atributo **`indexedDB`** y el método **`open()`** abren la base con el nombre declarado o crean una nueva si no existe:

```
function iniciar(){
  cajadatos=document.getElementById('cajadatos');
  var boton=document.getElementById('grabar');
  boton.addEventListener('click', agregarobjeto, false);

  if('webkitIndexedDB' in window){
    window.indexedDB=window.webkitIndexedDB;
    window.IDBTransaction=window.webkitIDBTransaction;
    window.IDBKeyRange=window.webkitIDBKeyRange;
    window.IDBCursor=window.webkitIDBCursor;
  }else if('mozIndexedDB' in window){
    window.indexedDB=window.mozIndexedDB;
  }

  var solicitud=indexedDB.open('mibase');
  solicitud.addEventListener('error', errores, false);
  solicitud.addEventListener('success', crear, false);
}
```

Listado 11-3. *Abriendo la base de datos.*

La función **`iniciar()`** del Listado 11-3 prepara los elementos de la plantilla y abre la base de datos. La instrucción **`indexedDB.open()`** intenta abrir la base de datos con el nombre **`mibase`** y retorna el objeto **`solicitud`** con el resultado de la operación. Los eventos **`error`** o **`success`** son disparados sobre este objeto en caso de error o éxito, respectivamente.

> **IMPORTANTE:** Al momento de escribir estas líneas la API se encuentra en estado experimental. Algunos atributos, incluyendo **`indexedDB`**, necesitan el prefijo del navegador para trabajar apropiadamente. Antes de abrir la base de datos en la función **`iniciar()`** detectamos la existencia de **`webkitIndexedDB`** o **`mozIndexedDB`** y preparamos los atributos para cada uno de estos navegadores específicos (Google Chrome o Firefox). Luego de que este período experimental termine podremos eliminar el condicional **`if`** al comienzo del código del Listado 11-3 y utilizar los métodos reales.

Los eventos son una parte importante de esta API. IndexedDB es una API síncrona y asíncrona. La parte síncrona está siendo desarrollada en estos momentos y está destinada a trabajar con la API Web Workers. En cambio, la parte asíncrona está destinada a un uso web normal y ya se encuentra disponible. Un sistema asíncrono realiza tareas detrás de escena y retorna los resultados posteriormente. Con este propósito, esta API dispara diferentes eventos en cada operación. Cada acción sobre la base de datos y su contenido es procesada detrás de escena (mientras el sistema ejecuta otros códigos) y los eventos correspondientes son disparados luego para informar los resultados obtenidos.

Luego de que la API procesa la solicitud para la base de datos, un evento **`error`** o **`success`** es disparado de

acuerdo al resultado y una de las funciones **errores()** o **crear()** es ejecutada para controlar los errores o continuar con la definición de la base de datos, respectivamente.

Versión de la base de datos

Antes de comenzar a trabajar en el contenido de la base de datos, debemos seguir algunos pasos para completar su definición. Como dijimos anteriormente, las bases de datos IndexedDB usan versiones. Cuando la base de datos es creada, un valor **null** (nulo) es asignado a su versión. Por lo tanto, controlando este valor podremos saber si la base de datos es nueva o no:

```
function errores(e){
  alert('Error: '+e.code+' '+e.message);
}
function crear(e){
  bd=e.result || e.target.result;
  if(bd.version==''){
    var solicitud=bd.setVersion('1.0');
    solicitud.addEventListener('error', errores, false);
    solicitud.addEventListener('success', crearbd, false);
  }
}
```

Listado 11-4. Declarando la versión y respondiendo a eventos.

Nuestra función **errores()** es simple (no necesitamos procesar errores para esta aplicación de muestra). En este ejemplo solo usamos los atributos **code** y **message** de la interface IDBErrorEvent para generar un mensaje de alerta en caso de error. La función **crear()**, por otro lado, sigue los pasos correctos para detectar la versión de la base de datos y proveer un valor de versión en caso de que sea la primera vez que la aplicación es ejecutada en este ordenador. La función asigna el objeto **result** creado por el evento a la variable **bd** y usa esta variable para representar la base de datos (esta variable se definió como global para referenciar la base de datos en el resto del código).

> **IMPORTANTE:** En este momento algunos navegadores envían el objeto **result** a través del evento y otros a través del elemento que disparó el evento. Para seleccionar la referencia correcta de forma automática, usamos la lógica **e.result || e.target.result**. Seguramente usted deberá usar solo uno de estos valores en sus aplicaciones cuando las implementaciones estén listas.

La interface IDBDatabase provee la propiedad **version** para informar el valor de la versión actual y también provee el método **setVersion()** para declarar una nueva versión. Lo que hacemos en la función **crear()** en el Listado 11-4 es detectar el valor actual de la versión de la base de datos y declarar uno nuevo si es necesario (en caso de que el valor sea una cadena de texto vacía). Si la base de datos ya existe, el valor de la propiedad **version** será diferente de **null** (nulo) y no tendremos que configurar nada, pero si esta es la primera vez que el usuario utiliza esta aplicación entonces deberemos declarar un nuevo valor para la versión y configurar la base de datos.

El método **setVersion()** recibe una cadena de texto que puede ser un número o cualquier valor que se nos ocurra, solo debemos estar seguros de que siempre usamos el mismo valor en cada código para abrir la versión correcta de la base de datos. Este método es, así como cualquier otro procedimiento en esta API, asíncrono. La versión será establecida detrás de escena y el resultado será informado al código principal a través de eventos. Si ocurre un error en el proceso, llamamos a la función **errores()** como lo hicimos anteriormente, pero si la versión es establecida correctamente entonces la función **crearbd()** es llamada para declarar los Almacenes de Objetos e índices que usaremos en esta nueva versión.

Almacenes de Objetos e índices

A este punto debemos comenzar a pensar sobre la clase de objetos que vamos a almacenar y cómo vamos a leer más adelante la información contenida en los Almacenes de Objetos. Si hacemos algo mal o queremos agregar algo en la configuración de nuestra base de datos en el futuro deberemos declarar una nueva versión y migrar los datos desde la anterior, por lo que es importante tratar de organizar todo lo mejor posible desde el principio. Esto es debido a que la creación de Almacenes de Objetos e índices solo es posible durante una transacción **setVersion**.

```
function crearbd(){
  var almacen=bd.createObjectStore('peliculas',{keyPath:'id'});
  almacen.createIndex('BuscarFecha', 'fecha',{unique: false});
}
```

Listado 11-5. *Declarando Almacenes de Objetos e índices.*

Para nuestro ejemplo solo necesitamos un Almacén de Objetos (para almacenar películas) y dos índices. El primer índice, **id**, es declarado como el atributo **clave** para el método **createObjectStore()** cuando el Almacén de Objetos es creado. El segundo índice es asignado al Almacén de Objetos usando el método **createIndex()**. Este índice fue identificado con el nombre **BuscarFecha** y declarado para la propiedad **fecha** (esta propiedad es ahora un índice). Más adelante vamos a usar este índice para ordenar películas por año.

Agregando Objetos

Por el momento tenemos una base de datos con el nombre **mibase** que tendrá el valor de versión **1.0** y contendrá un Almacén de Objetos llamado **peliculas** con dos índices: **id** y **fecha**. Con esto ya podemos comenzar a agregar objetos en el almacén:

```
function agregarobjeto(){
  var clave=document.getElementById('clave').value;
  var titulo=document.getElementById('texto').value;
  var fecha=document.getElementById('fecha').value;

  var transaccion=bd.transaction(['peliculas'],
                                  IDBTransaction.READ_WRITE);
  var almacen=transaccion.objectStore('peliculas');
  var solicitud=almacen.add({id: clave, nombre: titulo, fecha:
                                                      fecha});
  solicitud.addEventListener('success', function(){
                                    mostrar(clave) }, false);

  document.getElementById('clave').value='';
  document.getElementById('texto').value='';
  document.getElementById('fecha').value='';
}
```

Listado 11-6. *Agregando objetos.*

Al comienzo de la función **iniciar()** habíamos agregado al botón del formulario una escucha para el evento **click**. Esta escucha ejecuta la función **agregarobjeto()** cuando el evento es disparado (el botón es presionado). Esta función toma los valores de los campos del formulario (**clave**, **texto** y **fecha**) y luego genera una transacción para almacenar un nuevo objeto usando esta información.

Para comenzar la transacción, debemos usar el método **transaction()**, especificar el Almacén de Objetos involucrado en la transacción y el tipo de transacción a realizar. En este caso el almacén es **peliculas** y el tipo es declarado como **READ_WRITE**.

El próximo paso es seleccionar el Almacén de Objetos que vamos a usar. Debido a que la transacción puede ser originada para varios Almacenes de Objetos, tenemos que declarar cuál corresponde con la siguiente operación. Usando el método **objectStore()** abrimos el Almacén de Objetos y lo asignamos a la transacción con la siguiente línea: **transaccion.objectStore('peliculas')**.

Es momento de agregar el objeto al Almacén de Objetos. En este ejemplo usamos el método **add()** debido a que queremos crear un nuevo objeto, pero podríamos haber utilizado el método **put()** en su lugar si nuestra intensión hubiese sido modificar un viejo objeto. El método **add()** genera el objeto usando las propiedades **id**, **nombre** y **fecha** y las variables **clave**, **titulo** y **fecha**.

Finalmente, escuchamos al evento disparado por esta solicitud y ejecutamos la función **mostrar()** en caso de éxito. Existe también un evento **error**, por supuesto, pero como la respuesta a los errores depende de lo que usted quiera para su aplicación, no consideramos esa posibilidad en este ejemplo.

Leyendo Objetos

Si el objeto es correctamente almacenado, el evento **success** es disparado y la función **mostrar()** es ejecutada. En el código del Listado 11-6, esta función fue llamada dentro de una función anónima para poder pasar la variable **clave**. Ahora vamos a tomar este valor para leer el objeto previamente almacenado:

```
function mostrar(clave){
  var transaccion=bd.transaction(['peliculas']);
  var almacen=transaccion.objectStore('peliculas');
  var solicitud=almacen.get(clave);
  solicitud.addEventListener('success', mostrarlista, false);
}
function mostrarlista(e){
  var resultado=e.result || e.target.result;
  cajadatos.innerHTML='<div>'+resultado.id+' - '+resultado.nombre+'
                                    - '+resultado.fecha+'</div>';
}
```

Listado 11-7. *Leyendo y mostrando el objeto almacenado.*

El código del Listado 11-7 genera una transacción **READ_ONLY** y usa el método **get()** para leer el objeto con la clave recibida. No tenemos que declarar el tipo de transacción porque **READ_ONLY** es establecido por defecto.

El método **get()** retorna el objeto almacenado con la propiedad **id=clave**. Si, por ejemplo, insertamos la película *El Padrino* de nuestra lista, la variable **clave** tendrá el valor "tt0068646". Este valor es recibido por la función **mostrar()** y usado por el método **get()** para leer la película *El Padrino*. Como puede ver, este código es solo ilustrativo ya que solo puede retornar la misma película que acabamos de almacenar.

Como cada operación es asíncrona, necesitamos dos funciones para mostrar la información. La función **mostrar()** genera la transacción y lee el objeto, y la función **mostrarlista()** muestra los valores de las propiedades del objeto en pantalla. Otra vez, solo estamos escuchando al evento **success**, pero un evento **error** podría ser disparado en caso de que el objeto no pueda ser leído por alguna circunstancia en particular.

La función **mostrarlista()** recibe un objeto. Para acceder a sus propiedades solo tenemos que usar la variable representando al objeto y el nombre de la propiedad, como en **resultado.id** (la variable **resultado** representa el objeto e **id** es una de sus propiedades).

Finalizando el código

Del mismo modo que cualquier código previo, el ejemplo debe ser finalizado agregando una escucha para el evento **load** que ejecute la función **iniciar()** tan pronto como la aplicación es cargada en la ventana del navegador:

```
window.addEventListener('load', iniciar, false);
```

Listado 11-8. *Iniciando la aplicación.*

> **Hágalo usted mismo:** Es momento de probar la aplicación en el navegador. Copie todos los códigos Javascript desde el Listado 11-3 al Listado 11-8 en el archivo **indexed.js** y abra el documento HTML del Listado 11-1. Usando el formulario en la pantalla, inserte información acerca de las películas listadas al comienzo de este capítulo. Cada vez que una nueva película es insertada, la misma información es mostrada en la caja de la derecha.

11.3 Listando datos

El método **get()** implementado en el código del Listado 11-7 solo retorna un objeto por vez (el último insertado). En el siguiente ejemplo vamos a usar un cursor para generar una lista incluyendo todas las películas almacenadas en el Almacén de Objetos **peliculas**.

Cursores

Los cursores son una alternativa ofrecida por la API para obtener y navegar a través de un grupo de objetos encontrados en la base de datos. Un cursor obtiene una lista específica de objetos de un Almacén de Objetos e inicia un puntero que apunta a un objeto de la lista a la vez.

Para generar un cursor, la API provee el método **openCursor()**. Este método extrae información del Almacén de Objetos seleccionado y retorna un objeto IDBCursor que tiene sus propios atributos y métodos para manipular el cursor:

continue() Este método mueve el puntero del cursor una posición y el evento **success** del cursor es disparado nuevamente. Cuando el puntero alcanza el final de la lista, el evento **success** es también disparado, pero retorna un objeto vacío. El puntero puede ser movido a una posición específica declarando un valor de índice dentro de los paréntesis.

delete() Este método elimina el objeto en la posición actual del cursor.

update(valor) Este método es similar a **put()** pero modifica el valor del objeto en la posición actual del cursor.

El método **openCursor()** también tiene atributos para especificar el tipo de objetos retornados y su orden. Los valores por defecto retornan todos los objetos disponibles en el Almacén de Objetos seleccionado, organizados en orden ascendente. Estudiaremos este tema más adelante.

```
function iniciar(){
  cajadatos=document.getElementById('cajadatos');
  var boton=document.getElementById('grabar');
  boton.addEventListener('click', agregarobjeto, false);
  if('webkitIndexedDB' in window){
    window.indexedDB=window.webkitIndexedDB;
    window.IDBTransaction=window.webkitIDBTransaction;
    window.IDBKeyRange=window.webkitIDBKeyRange;
    window.IDBCursor=window.webkitIDBCursor;
  }else if('mozIndexedDB' in window){
    window.indexedDB=window.mozIndexedDB;
  }
  var solicitud=indexedDB.open('mibase');
  solicitud.addEventListener('error', errores, false);
  solicitud.addEventListener('success', crear, false);
}
function errores(e){
  alert('Error: '+e.code+' '+e.message);
}
function crear(e){
  bd=e.result || e.target.result;
  if(bd.version==''){
    var solicitud=bd.setVersion('1.0');
    solicitud.addEventListener('error', errores, false);
    solicitud.addEventListener('success', crearbd, false);
  }else{
    mostrar();
  }
}
function crearbd(){
  var almacen=bd.createObjectStore('peliculas',{keyPath: 'id'});
  almacen.createIndex('BuscarFecha', 'fecha',{unique: false});
}
```

```
function agregarobjeto(){
  var clave=document.getElementById('clave').value;
  var titulo=document.getElementById('texto').value;
  var fecha=document.getElementById('fecha').value;
  var transaccion=bd.transaction(['peliculas'],
                                  IDBTransaction.READ_WRITE);
  var almacen=transaccion.objectStore('peliculas');
  var solicitud=almacen.add({id: clave, nombre: titulo, fecha:
                                                      fecha});
  solicitud.addEventListener('success', mostrar, false);
  document.getElementById('clave').value='';
  document.getElementById('texto').value='';
  document.getElementById('fecha').value='';
}
function mostrar(){
  cajadatos.innerHTML='';
  var transaccion=bd.transaction(['peliculas']);
  var almacen=transaccion.objectStore('peliculas');
  var cursor=almacen.openCursor();
  cursor.addEventListener('success', mostrarlista, false);
}
function mostrarlista(e){
  var cursor=e.result || e.target.result;
  if(cursor){
    cajadatos.innerHTML+='<div>'+cursor.value.id+' -
        '+cursor.value.nombre+' - '+cursor.value.fecha+'</div>';
    cursor.continue();
  }
}
window.addEventListener('load', iniciar, false);
```

Listado 11-9. *Lista de objetos.*

El Listado 11-9 muestra el código Javascript completo que necesitamos para este ejemplo. De las funciones usadas para configurar la base de datos, solo **`crear()`** presenta un pequeño cambio. Ahora, cuando la versión de la base de datos sea diferente a **`null`** (lo que significa que la base de datos ya ha sido creada) la función **`mostrar()`** es ejecutada. Esta función ahora se encuentra a cargo de mostrar la lista de objetos almacenados en el Almacén de Objetos, por lo que si la base de datos ya existe veremos una lista de objetos en la caja derecha de la pantalla tan pronto como la página web es cargada.

La mejora introducida en este código se encuentra en las funciones **`mostrar()`** y **`mostrarlista()`**. Aquí es donde trabajamos con cursores por primera vez.

Leer información de la base de datos con un cursor es también una operación que debe hacerse a través de una transacción. Por este motivo, lo primero que hacemos en la función **`mostrar()`** es generar una transacción del tipo **`READ_ONLY`** sobre el Almacén de Objetos **`peliculas`**. Este Almacén de Objetos es seleccionado como el involucrado en la transacción y luego el cursor es abierto sobre este almacén usando el método **`openCursor()`**.

Si la operación es exitosa, un objeto es retornado con toda la información obtenida del Almacén de Objetos, un evento **`success`** es disparado desde este objeto y la función **`mostrarlista()`** es ejecutada.

Para leer la información, el objeto retornado por la operación ofrece varios atributos:

key Este atributo retorna el valor de la clave del objeto en la posición actual del cursor.

value Este atributo retorna el valor de cualquier propiedad del objeto en la posición actual del cursor. El nombre de la propiedad debe ser especificado como una propiedad del atributo (por ejemplo, **`value.fecha`**).

direction Los objetos pueden ser leídos en orden ascendente o descendente (como veremos más adelante); este atributo retorna la condición actual en la cual son leídos.

count Este atributo retorna en número aproximado de objetos en el cursor.

En la función **`mostrarlista()`** del Listado 11-9, usamos el condicional **`if`** para controlar el contenido del cursor. Si ningún objeto es retornado o el puntero alcanza el final de la lista, entonces el objeto estará vacío y el bucle no es continuado. Sin embargo, cuando el puntero apunta a un objeto válido, la información es mostrada en pantalla y el puntero es movido hacia la siguiente posición con **`continue()`**.

Es importante mencionar que no debemos usar un bucle **`while`** aquí debido a que el método **`continue()`**

dispara nuevamente el evento **success** del cursor y la función completa es ejecutada para leer el siguiente objeto retornado.

Hágalo usted mismo: El código del Listado 11-9 reemplaza todos los códigos Javascript previos. Vacíe el archivo **indexed.js** y copie en su interior este nuevo código. Abra la plantilla del Listado 11-1 y, si aún no lo hizo, inserte todas las películas del listado encontrado al comienzo de este capítulo. Verá la lista completa de películas en la caja derecha de la pantalla en orden ascendente, de acuerdo al valor de la propiedad **id**.

Cambio de orden

Hay dos detalles que necesitamos modificar para obtener la lista que queremos. Todas las películas en nuestro ejemplo están listadas en orden ascendente y la propiedad usada para organizar los objetos es **id**. Esta es la propiedad declarada como el atributo **clave** cuando el Almacén de Objetos **peliculas** fue creado, y es por tanto el índice usado por defecto. Pero ésta clase de valores no es lo que a nuestros usuarios normalmente les interesa.

Considerando esta situación, creamos otro índice en la función **crearbd()**. El nombre de este índice adicional fue declarado como **BuscarFecha** y la propiedad asignada al mismo es **fecha**. Este índice nos permitirá ordenar las películas de acuerdo al valor del año en el que fueron filmadas.

```
function mostrar(){
  cajadatos.innerHTML='';
  var transaccion=bd.transaction(['peliculas']);
  var almacen=transaccion.objectStore('peliculas');
  var indice=almacen.index('BuscarFecha');

  var cursor=indice.openCursor(null, IDBCursor.PREV);
  cursor.addEventListener('success', mostrarlista, false);
}
```

Listado 11-10. *Orden descendiente por año.*

La función en el Listado 11-10 reemplaza a la función **mostrar()** del código del Listado 11-9. Esta nueva función genera una transacción, luego asigna el índice **BuscarFecha** al Almacén de Objetos usado en la transacción, y finalmente usa **openCursor()** para obtener los objetos que tienen la propiedad correspondiente al índice (en este caso **fecha**).

Existen dos atributos que podemos especificar en **openCursor()** para seleccionar y ordenar la información obtenida por el cursor. El primer atributo declara el rango dentro del cual los objetos serán seleccionados y el segundo es una de las siguientes constantes:

NEXT (siguiente). El orden de los objetos retornados será ascendente (este es el valor por defecto).

NEXT_NO_DUPLICATE (siguiente no duplicado). El orden de los objetos retornados será ascendente y los objetos duplicados serán ignorados (solo el primer objeto es retornado si varios comparten el mismo valor de índice).

PREV (anterior). El orden de los objetos retornados será descendente.

PREV_NO_DUPLICATE (anterior no duplicado). El orden de los objetos retornados será descendente y los objetos duplicados serán ignorados (solo el primer objeto es retornado si varios comparten el mismo valor de índice).

Con el método **openCursor()** usado en la función **mostrar()** en el Listado 11-10, obtenemos los objetos en orden descendente y declaramos el rango como **null**. Vamos a aprender cómo construir un rango y retornar solo los objetos deseados más adelante en este capítulo.

Hágalo usted mismo: Tome el código anterior presentado en el Listado 11-9 y reemplace la función **show()** con la nueva función del Listado 11-10. Esta nueva función lista las películas en la pantalla por año y en orden descendente (las más nuevas primero). El resultado debería ser el siguiente:

id: tt1285016 **nombre:** La Red Social **fecha:** 2010

id: tt0111161 **nombre:** Cadena Perpetua **fecha:** 1994

id: tt0086567 **nombre:** Juegos de Guerra **fecha:** 1983

id: tt0068646 **nombre:** El Padrino **fecha:** 1972

11.4 Eliminando datos

Hemos aprendido cómo agregar, leer y listar datos. Es hora de estudiar la posibilidad de eliminar objetos de un Almacén de Objetos. Como mencionamos anteriormente, el método **delete()** provisto por la API recibe un valor y elimina el objeto con la clave correspondiente a ese valor.

El código es sencillo; solo necesitamos crear un botón para cada objeto listado en pantalla y generar una transacción **READ_WRITE** para poder realizar la operación y eliminar el objeto:

```
function mostrarlista(e){
  var cursor=e.result || e.target.result;
  if(cursor){
    cajadatos.innerHTML+='<div>'+cursor.value.id+' -
        '+cursor.value.nombre+' - '+cursor.value.fecha+' <button
            onclick="eliminar(\''+cursor.value.id+'\')">Eliminar
                                                </button></div>';
    cursor.continue();
  }
}
function eliminar(clave){
  if(confirm('Está Seguro?')){
    var transaccion=bd.transaction(['peliculas'],
                                   IDBTransaction.READ_WRITE);
    var almacen=transaccion.objectStore('peliculas');
    var solicitud=almacen.delete(clave);
    solicitud.addEventListener('success', mostrar, false);
  }
}
```

Listado 11-11. *Eliminando objetos.*

El botón agregado a cada película en la función **mostrarlista()** del Listado 11-11 contiene un manejador de eventos en línea. Cada vez que el usuario hace clic en uno de estos botones, la función **eliminar()** es ejecutada con el valor de la propiedad **id** como su atributo. Esta función genera primero una transacción **READ_WRITE** y luego usando la clave recibida procede a eliminar el correspondiente objeto del Almacén de Objetos **peliculas**.

Al final, si la operación fue exitosa, el evento **success** es disparado y la función **mostrar()** es ejecutada para actualizar la lista de películas en pantalla.

> **Hágalo usted mismo:** Tome el código anterior del Listado 11-9, reemplace la función **mostrarlista()** y agregue la función **eliminar()** del código del Listado 11-11. Finalmente, abra el documento HTML del Listado 11-1 para probar la aplicación. Podrá ver el listado de películas pero ahora cada línea incluye un botón para eliminar la película del Almacén de Objetos. Experimente agregando y eliminando películas.

11.5 Buscando datos

Probablemente la operación más importante realizada en un sistema de base de datos es la búsqueda. El propósito absoluto de esta clase de sistemas es indexar la información almacenada para facilitar su posterior búsqueda. Como estudiamos anteriormente es este capítulo, el método **`get()`** es útil para obtener un objeto por vez cuando conocemos el valor de su clave, pero una operación de búsqueda es usualmente más compleja que esto.

Para obtener una lista específica de objetos desde el Almacén de Objetos, tenemos que pasar un rango como argumento para el método **`openCursor()`**. La API incluye la interface IDBKeyRange con varios métodos y propiedades para declarar un rango y limitar los objetos retornados:

only(valor) Solo los objetos con la clave que concuerda con **`valor`** son retornados. Por ejemplo, si buscamos películas por año usando **`only("1972")`**, solo la película *El Padrino* será retornada desde el almacén.

bound(bajo, alto, bajoAbierto, altoAbierto) Para realmente crear un rango, debemos contar con valores que indiquen el comienzo y el final de la lista, y además especificar si esos valores también serán incluidos. El valor del atributo **`bajo`** indica el punto inicial de la lista y el atributo **`alto`** es el punto final. Los atributos **`bajoAbierto`** y **`altoAbierto`** son valores booleanos usados para declarar si los objetos que concuerdan exactamente con los valores de los atributos **`bajo`** y **`alto`** serán ignorados. Por ejemplo, **`bound("1972", "2010", false, true)`** retornará una lista de películas filmadas desde el año 1972 hasta el año 2010, pero no incluirá las realizadas específicamente en el 2010 debido a que el valor es **`true`** (verdadero) para el punto donde el rango termina, lo que indica que el final es abierto y las películas de ese año no son incluidas.

lowerBound(valor, abierto) Este método crea un rango abierto que comenzará por **`valor`** e irá hacia arriba hasta el final de la lista. Por ejemplo, **`lowerBound("1983", true)`** retornará todas las películas hechas luego de 1983 (sin incluir las filmadas en ese año en particular).

upperBound(valor, abierto) Este es el opuesto al anterior. Creará un rango abierto, pero los objetos retornados serán todos los que posean un valor de índice menor a **`valor`**. Por ejemplo, **`upperBound("1983", false)`** retornará todas las películas hechas antes de 1983, incluyendo las realizadas ese mismo año (el atributo **`abierto`** fue declarado como **`false`**).

Preparemos primero una nueva plantilla para presentar un formulario en pantalla con el que se pueda buscar películas:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>IndexedDB API</title>
  <link rel="stylesheet" href="indexed.css">
  <script src="indexed.js"></script>
</head>
<body>
  <section id="cajaformulario">
    <form name="formulario">
      <p>Buscar Película por Año:<br><input type="search"
                                  name="fecha" id="fecha"></p>
      <p><input type="button" name="buscar" id="buscar"
                                           value="Buscar"></p>
    </form>
  </section>
  <section id="cajadatos">
    No hay información disponible
  </section>
</body>
</html>
```

Listado 11-12. *Formulario de búsqueda.*

Este nuevo documento HTML provee un botón y un campo de texto donde ingresar el año para buscar películas de acuerdo a un rango especificado en el siguiente código:

```
function iniciar(){
  cajadatos=document.getElementById('cajadatos');
  var boton=document.getElementById('buscar');
  boton.addEventListener('click', buscarobjetos, false);

  if('webkitIndexedDB' in window){
    window.indexedDB=window.webkitIndexedDB;
    window.IDBTransaction=window.webkitIDBTransaction;
    window.IDBKeyRange=window.webkitIDBKeyRange;
    window.IDBCursor=window.webkitIDBCursor;
  }else if('mozIndexedDB' in window){
    window.indexedDB=window.mozIndexedDB;
  }

  var solicitud=indexedDB.open('mibase');
  solicitud.addEventListener('error', errores, false);
  solicitud.addEventListener('success', crear, false);
}
function errores(e){
  alert('Error: '+e.code+' '+e.message);
}
function crear(e){
  bd=e.result || e.target.result;
  if(bd.version==''){
    var solicitud=bd.setVersion('1.0');
    solicitud.addEventListener('error', errores, false);
    solicitud.addEventListener('success', crearbd, false);
  }
}
function crearbd(){
  var almacen=bd.createObjectStore('peliculas', {keyPath: 'id'});
  almacen.createIndex('BuscarFecha', 'fecha', { unique: false });
}
function buscarobjetos(){
  cajadatos.innerHTML='';
  var buscar=document.getElementById('fecha').value;

  var transaccion=bd.transaction(['peliculas']);
  var almacen=transaccion.objectStore('peliculas');
  var indice=almacen.index('BuscarFecha');
  var rango=IDBKeyRange.only(buscar);

  var cursor=indice.openCursor(rango);
  cursor.addEventListener('success', mostrarlista, false);
}
function mostrarlista(e){
  var cursor=e.result || e.target.result;
  if(cursor){
    cajadatos.innerHTML+='<div>'+cursor.value.id+' -
       '+cursor.value.nombre+' - '+cursor.value.fecha+'</div>';
    cursor.continue();
  }
}
window.addEventListener('load', iniciar, false);
```

Listado 11-13. *Buscando películas.*

La función **buscarobjetos()** es la más importante del Listado 11-13. En esta función generamos una transacción de solo lectura **READ_ONLY** para el Almacén de Objetos **peliculas**, seleccionamos el índice **BuscarFecha** para usar la propiedad **fecha** como índice, y creamos un rango desde el valor de la variable **buscar** (el año insertado en el formulario por el usuario). El método usado para construir el rango es **only()**, pero puede probar con cualquiera de los métodos estudiados. Este rango es pasado luego como un argumento del método **openCursor()**. Si la operación es exitosa, la función **mostrarlista()** imprimirá en pantalla la lista de películas del año seleccionado.

El método **only()** retorna solo las películas que concuerdan exactamente con el valor de la variable **buscar**.

Para probar otros métodos, puede proveer valores por defecto para completar el rango, por ejemplo **`bound(buscar, "2011", false, true)`**.

El método **`openCursor()`** puede tomar dos posibles atributos al mismo tiempo. Por esta razón, una instrucción como **`openCursor(rango, IDBCursor.PREV)`** es válida y retornará los objetos en el rango especificado y en orden descendente (usando como referencia el mismo índice utilizado para el rango).

> **IMPORTANTE:** La característica de buscar textos se encuentra bajo consideración en este momento, pero aún no ha sido desarrollada o incluso incluida en la especificación oficial. Para obtener más información sobre esta API, visite nuestro sitio web y siga los enlaces correspondientes a este capítulo.

11.6 Referencia rápida

La API IndexedDB tiene una infraestructura de bajo nivel. Los métodos y propiedades estudiados en este capítulo son solo parte de lo que esta API tiene para ofrecer. Con el propósito de simplificar los ejemplos, no seguimos ninguna estructura específica. Sin embargo, esta API, así como otras, está organizada en interfaces. Por ejemplo, existe una interface específica para tratar con la organización de la base de datos, otra para la creación y manipulación de Almacenes de Objetos, etc… Cada interface incluye sus propios métodos y propiedades, por lo que ahora vamos a presentar la información compartida en este capítulo siguiendo esta clasificación oficial.

> **IMPORTANTE:** Las descripciones presentadas en esta referencia rápida solo muestran los aspectos más relevantes de cada interface. Para estudiar la especificación completa, visite nuestro sitio web y siga los enlaces correspondientes a este capítulo.

Interface Environment (IDBEnvironment y IDBFactory)

La interface Environment, o IDBEnvironment, incluye un atributo IDBFactory. Juntas estas interfaces proveen los elementos necesarios para operar con bases de datos:

indexedDB Este atributo provee un mecanismo para acceder al sistema de base de datos indexado.

open(nombre) Este método abre una base de datos con el nombre especificado en su atributo. Si no existe una base de datos previa, una nueva es creada con el nombre provisto.

deleteDatabase(nombre) Este método elimina una base de datos con el nombre especificado en su atributo.

Interface Database (IDBDatabase)

El objeto retornado luego de la apertura o creación de una base de datos es procesado por esta interface. Con este objetivo, la interface provee varios métodos y propiedades:

version Esta propiedad retorna el valor de la versión actual de la base de datos abierta.

name Esta propiedad retorna el nombre de la base de datos abierta.

objectStoreNames Esta propiedad retorna un listado de los nombres de los Almacenes de Objetos contenidos dentro de la base de datos abierta.

setVersion(valor) Este método establece una nueva versión para la base de datos abierta. El atributo **`valor`** puede ser cualquier cadena de texto que deseemos.

createObjectStore(nombre, clave, autoIncremento) Este método crea un nuevo Almacén de Objetos para la base de datos abierta. El atributo **`nombre`** representa el nombre del Almacén de Objetos, **`clave`** es un índice común para todos los objetos almacenados en este almacén, y **`autoIncremento`** es un valor booleano que activa un generador automático de claves.

deleteObjectStore(nombre) Este método elimina un Almacén de Objetos con el nombre especificado en su atributo.

transaction(almacenes, tipo, máximo) Este método inicia una transacción con la base de datos. La transacción puede ser especificada para uno o más Almacenes de Objetos declarados en el atributo **`almacenes`**, y puede ser creada para diferentes tipos de acceso de acuerdo con el atributo **`tipo`**. Puede también recibir un atributo **`máximo`** en milisegundos para especificar el tiempo máximo permitido que la operación puede tardar en realizarse. Para mayor información sobre cómo configurar una transacción, ver Interface Transaction en esta Referencia rápida.

Interface Object Store (IDBObjectStore)

Esta interface incluye todos los métodos y propiedades necesarios para manipular objetos en un Almacén de Objetos (Object Store).

name Esta propiedad retorna el nombre del Almacén de Objetos actualmente en uso.

keyPath Esta propiedad retorna la **`clave`**, si existe, del Almacén de Objetos actualmente en uso (esta es la

clave definida como índice común en el momento en el que el Almacén de Objetos fue creado).

IndexNames Esta propiedad retorna una lista de los nombres de los índices creados para el Almacén de Objetos actualmente en uso.

add(objeto) Este método agrega un objeto al Almacén de Objetos seleccionado con la información provista en su atributo. Si un objeto con el mismo índice ya existe, un error es retornado. El método puede recibir un par clave/valor o un objeto conteniendo varios pares clave/valor como atributo.

put(objeto) Este método agrega un objeto al Almacén de Objetos seleccionado con la información provista en su atributo. Si un objeto con el mismo índice ya existe, el objeto es sobrescrito con la nueva información. El método puede recibir un par clave/valor o un objeto conteniendo varios pares clave/valor como atributo.

get(clave) Este método retorna el objeto con el valor del índice igual a **`clave`**.

delete(clave) Este método elimina el objeto con el valor del índice igual a **`clave`**.

createIndex(nombre, propiedad, único) Este método crea un nuevo índice para el Almacén de Objetos seleccionado. El atributo **`nombre`** especifica el nombre del índice, el atributo **`propiedad`** declara la propiedad de los objetos que será asociada con este índice, y el atributo **`único`** indica si los objetos con un mismo valor de índice serán permitidos o no.

index(nombre) Este método activa el índice con el nombre especificado en su atributo.

deleteIndex(nombre) Este método elimina el índice con el nombre especificado en su atributo.

openCursor(rango, dirección) Este método crea un cursor para el Almacén de Objetos seleccionado. El atributo **`rango`** es un objeto range (definido por la Interface Range) para determinar cuáles objetos son seleccionados. El atributo **`dirección`** establece el orden de estos objetos. Para mayor información sobre cómo configurar y manipular un cursor, ver la Interface Cursors en esta Referencia Rápida. Para mayor información sobre cómo construir un rango con el objeto range, ver la Interface Range en esta Referencia Rápida.

Interface Cursors (IDBCursor)

Esta interface provee valores de configuración para especificar el orden de los objetos seleccionados desde el Almacén de Objetos. Estos valores deben ser declarados como el segundo atributo del método **`openCursor()`**, como en **`openCursor(null, IDBCursor.PREV)`**.

NEXT (siguiente). Esta constante determina un orden ascendente para los objetos apuntados por el cursor (este es el valor por defecto).

NEXT_NO_DUPLICATE (siguiente no duplicado). Esta constante determina un orden ascendente para los objetos apuntados por el cursor e ignora los duplicados.

PREV (anterior). Esta constante determina un orden descendente para los objetos apuntados por el cursor.

PREV_NO_DUPLICATE (anterior no duplicado). Esta constante determina un orden descendente para los objetos apuntados por el cursor e ignora los duplicados.

Esta interface también provee varios métodos y propiedades para manipular los objetos apuntados por el cursor.

continue(clave) Este método mueve el puntero del cursor hacia el siguiente objeto en la lista o hacia el objeto referenciado por el atributo **`clave`**, si es declarado.

delete() Este método elimina el objeto que actualmente se encuentra apuntado por el cursor.

update(valor) Este método modifica el objeto actualmente apuntado por el cursor con el valor provisto por su atributo.

key Esta propiedad retorna el valor del índice del objeto actualmente apuntado por el cursor.

value Esta propiedad retorna el valor de cualquier propiedad del objeto actualmente apuntado por el cursor.

direction Esta propiedad retorna el orden de los objetos obtenidos por el cursor (ascendente o descendente).

Interface Transactions (IDBTransaction)

Esta interface provee valores de configuración para especificar el tipo de transacción que se va a llevar a cabo. Estos valores deben ser declarados como el segundo atributo del método **`transaction()`**, como en **`transaction(almacenes, IDBTransaction.READ_WRITE)`**.

***READ*_ONLY** Esta constante configura la transacción como una transacción de solo lectura (este es el valor por defecto).

READ_WRITE Esta constante configura la transacción como una transacción de lectura-escritura.

VERSION_CHANGE Este tipo de transacción es usado solamente para actualizar el número de versión de la base de datos.

Interface Range (IDBKeyRangeConstructors)

Esta interface provee varios métodos para la construcción de un rango a ser usado con cursores:

only(valor) Este método retorna un rango con los puntos de inicio y final iguales a **`valor`**.

bound(bajo, alto, bajoAbierto, altoAbierto) Este método retorna un rango con el punto de inicio declarado por **`bajo`**, el punto final declarado por **`alto`**, y si estos valores serán excluidos de la lista de objetos o no declarado por los últimos dos atributos.

lowerBound(valor, abierto) Este método retorna un rango comenzando por **`valor`** y terminando al final de la lista de objetos. El atributo **`abierto`** determina si los objetos que concuerdan con **`valor`** serán excluidos o no.

upperBound(valor, abierto) Este método retorna un rango comenzando desde el inicio de la lista de objetos y terminando en **`valor`**. El atributo **`abierto`** determina si los objetos que concuerdan con **`valor`** serán excluidos o no.

Interface Error (IDBDatabaseException)

Los errores retornados por las operaciones en la base de datos son informados a través de esta interface.

code Esta propiedad representa el número de error.

message Esta propiedad retorna un mensaje describiendo el error.

El valor retornado también puede ser comparado con las constantes de la siguiente lista para encontrar el error correspondiente.

UNKNOWN_ERR - valor 0.

NON_TRANSIENT_ERR - valor 1.

NOT_FOUND_ERR - valor 2.

CONSTRAINT_ERR - valor 3.

DATA_ERR - valor 4.

NOT_ALLOWED_ERR - valor 5.

TRANSACTION_INACTIVE_ERR - valor 6.

ABORT_ERR - valor 7.

READ_ONLY_ERR - valor 11.

RECOVERABLE_ERR - valor 21.

TRANSIENT_ERR - valor 31.

TIMEOUT_ERR - valor 32.

DEADLOCK_ERR - valor 33.