ceviche.js

Capítulo 4: APIs del navegador

> *Ya conocemos el DOM, con el que podemos modificar elementos y manejar eventos. De paso, vimos algo del CSSOM, y logramos saber cuándo se ejecutan algunas animaciones. Pero el navegador no se limita a ello y ofrece más APIs para crear aplicaciones interactivas y complejas.*

Aplicaciones web y HTML5

Una aplicación web es una herramienta similar a una aplicación de escritorio, pero que es utilizada dentro del navegador, y tiene dos ventajas importantes:

- Es ubicuo: Una aplicación web está disponible en casi cualquier equipo que tenga un navegador web incorporado. Debido a que no existe necesidad de instalar una aplicación, la información del usuario está disponible sin importar el equipo desde donde se acceda a la aplicación.
- Es auto-actualizable: Una aplicación web no reside en el equipo, si no en un servidor web. Esto tiene como ventaja que puede ser actualizada sin necesidad de la interacción del usuario.

Mientras que, como desventajas:

- Su disponibilidad depende de otros factores: De una conexión a Internet, del servidor de la aplicación (tanto para aplicaciones de Internet como intranets), y en situaciones menos comunes, del navegador usado.
- Está limitado al navegador: El navegador por definición está limitado en cuanto a lo que puede acceder del equipo, lo que en términos técnicos se conoce como *sandboxing*. Este tipo de limitaciones, por consiguiente, limitan a las aplicaciones web que se ejecutan dentro de él.

Las aplicaciones web no son de ahora y no implican utilizar solo JavaScript. Existían aplicaciones web antes del llamado *Web 2.0*, que utilizan JavaScript,

Java, Flash, Flex, Silverlight, e incluso algunas solo utilizan HTML y CSS. Sin embargo, el avance que ha tenido JavaScript, desde la *Web 2.0* hasta el HTML5, ha logrado superar en alguna forma las desventajas que tenían las aplicaciones web.

Ahora se pueden realizar peticiones al servidor sin recargar toda la página, tener una comunicación interactiva con el servidor, realizar algunas operaciones sin necesidad de tener una conexión, leer y escribir archivos en el equipo, entre otros.

Algunas de las siguientes APIs del navegador servirán para **La Buena Espina**, pero utilizarlas no significa que estemos creando una aplicación web, ya que una aplicación web no está determinada por las tecnologías que usa.

Web Storage

Empecemos con una API simple de usar pero que soluciona un problema común al trabajar con una aplicación web: El dueño de **La Buena Espina** quiere un formulario de contacto para que los comensales puedan dar sus impresiones sobre el servicio y la comida. Pero, ¿qué pasaría si luego de enviar el formulario se pierde la conexión, el usuario cierra su navegador o el servidor no responde? Los comentarios no llegarán al dueño y se pueden perder buenas críticas con respecto al restaurante.

Las APIs de Web Storage solucionan este problema, al menos en parte, ya que permiten guardar información en el navegador. Esta información es guardada en formato *nombre - valor*, similar a un solo objeto plano en JavaScript, y puede existir en dos formas:

- *Local Storage*: Existiendo uno por cada *origen* (el valor devuelto por `location.origin`). Estará disponible luego de haber cerrado el navegador.
- *Session Storage*: Similar al *Local Storage*, solo está disponible mientras el navegador esté abierto.

Estos valores sobreviven a pestañas cerradas y recargadas, similar a las *cookies*, excepto que estas tienen un tamaño máximo, y otras limitaciones. Sin embargo,

tanto el *Local Storage* como el *Session Storage* tienen un tamaño máximo de 5 megabytes por *origen*.

Un punto importante a resaltar es que ambos *Storages* se comportan como objetos planos globales, y guardan tanto los nombres como los valores en forma de cadenas. Adicionalmente, tienen dos métodos para acceder y asignar valores: `getItem` y `setItem`.

Para el caso descrito en el primer párrafo, podríamos utilizar el *Local Storage* junto a `dom.js`:

```
dom('#contact-form').on('submit', function() {
  window.localStorage.setItem('contact_content', dom('#contact-
comment').value());
});
```

De esta forma, cuando enviemos el formulario, se guardará el contenido del elemento `#contact-comment` en el *Local Storage* bajo el nombre `contact_content`.

> *Soporte para Web Storage*

Geolocation

El dueño de **La Buena Espina** quiere que el sitio web de su restaurante invite al usuario a ir a sus locales, y una forma de lograr eso es indicarle a sus posibles clientes la ubicación exacta de sus locales. Pero con eso no basta, porque también podría indicarle al potencial cliente **cómo llegar** a alguno de sus locales, dependiendo de un dato que ahora es más fácil de conseguir: la *geolocalización*.

La *Geolocation API* utiliza diferentes formas para conocer la ubicación de un usuario (o, mejor dicho, del equipo que está utilizando un usuario), diiriendo cada forma en la precisión de la ubicación; y, cuando esta API logra encontrar la ubicación del equipo, devuelve un objeto con 2 valores básicos: la latitud y la longitud. Estos valores numéricos permiten ubicar un lugar en la Tierra a partir de un sistema de coordenadas único para todo el mundo, así que podemos tener

la certeza que el valor que devuelva esta API será (relativamente) exacto.

Para trabajar con la *Geolocation API* tenemos que acceder a un objeto dentro de `navigator` llamado `geolocation`, el cual contiene 3 métodos:

- `getCurrentPosition`: Trata de obtener la ubicación del equipo y toma 3 parámetros: Un *callback* que se ejecutará si se logra obtener la ubicación del equipo, un segundo *callback* que se ejecutará si no se logra obtener la ubicación (indicando el motivo del error), y un tercer objeto con configuración de la petición.
- `watchPosition`: Toma los mismos parámetros de `getCurrentPosition` y realiza un monitoreo de la ubicación del equipo, ejecutándose cada vez que el navegador detecte que la ubicación del equipo ha cambiado. Este método devuelve un id, el cual es utilizado por `clearWatch`.
- `clearWatch`: Detiene el monitoreo creado por el método `watchPosition`.

El último parámetro de `getCurrentPosition` y `watchPosition` puede tener los siguientes valores:

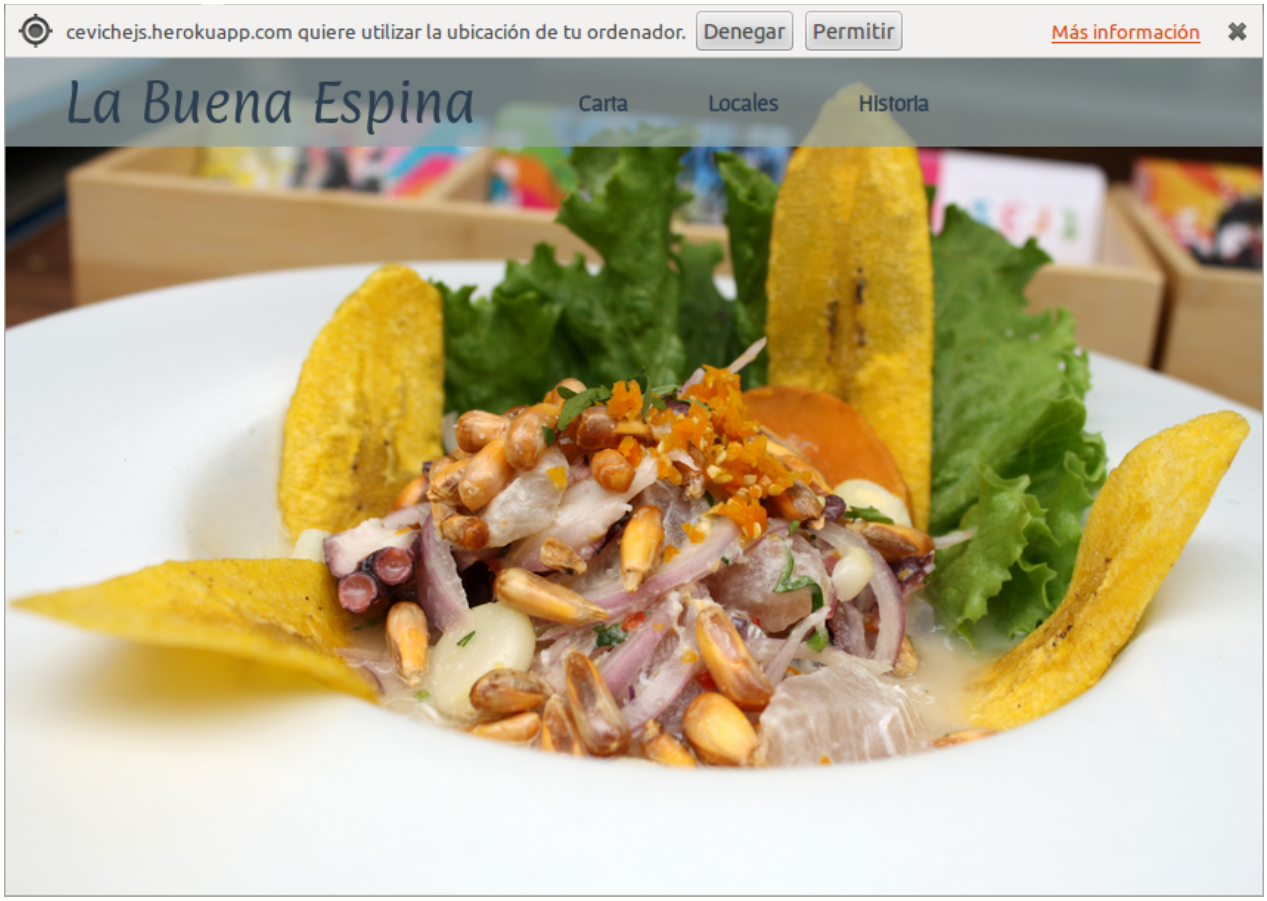
- `enableHighAccuracy`: Define un valor booleano que indica si la API tratará de obtener el valor más exacto para la ubicación.
- `timeout`: Indica el tiempo máximo (en milisegundos) que la API esperará por obtener un resultado, o, en caso contrario, lanzar el *callback* de error (segundo parámetro).
- `maximumAge`: Indica el tiempo máximo (en milisegundos) que el navegador guardará en memoria el valor devuelto por la API.

Sabiendo esto, podemos empezar a trabajar con la *Geolocation API*:

```
navigator.geolocation.getCurrentPosition(function(position) {
  console.log(position.coords);
}, function(error) {
  console.log(error.message, error.code);
}, {
  enableHighAccuracy: true,
  timeout: 2500,
```

```
    maximumAge: 0
  });
```

Cuando se ejecute este código se mostrará una ventana o un mensaje (dependiendo del navegador) pidiendo permiso al usuario para poder realizar la geolocalización. Es importante resaltar este punto ya que no es posible obtener la ubicación de un equipo sin previo permiso del usuario.



Permisos para geolocalización

Si denegamos el permiso de geolocalización al navegador, la consola nos mostrará este mensaje:

```
"User denied Geolocation"  1
```

Mientras que el primer valor devuelve un mensaje entendible para el usuario, el

segundo valor es un código de error devuelto por la API, el cual puede tener 3 valores:

Valor	Descripción
1	El usuario no dio permiso al navegador
2	No se pudo encontrar la ubicación
3	Pasó más del tiempo permitido en el *timeout* definido por el tercer parámetro

Y si le damos el permiso, nos devolverá el siguiente objeto (cuyos valores pueden cambiar de acuerdo al equipo y al tipo de conexión):

```
{
  accuracy: 75,
  altitude: null,
  altitudeAccuracy: null,
  heading: null,
  latitude: -12.1042457,
  longitude: -76.9628362,
  speed: null
}
```

Cuando ya tenemos estos valores podemos utilizar algún servicio de mapas, como Google Maps, Mapbox o Leaflet para mostrar la ubicación de forma visual en un mapa.

> *Soporte para Geolocation*

Application Cache

Para el caso de aplicaciones web suele ser de vital importancia el poder acceder a ellas de manera *offline*, sobre todo si la conexión a Internet solo se hace necesaria para respaldar información en un servidor externo. En este tipo de aplicaciones donde se debería poder acceder a los archivos "estáticos" de la aplicación independientemente del estado de conexión que tenga el equipo, y es

aquí donde el *Application Cache* entra en acción.

Esta API permite definir un archivo *manifiesto* que indicará cuáles son los archivos que se desean descargar cuando el navegador se conecta a la aplicación cuando está *online*, y que luego utilizará cuando no exista una conexión a Internet disponible.

Un manifiesto básico sigue el siguiente formato:

```
CACHE MANIFEST

/
/images/logo.png
/images/sprites.png
/styles/layout.css
/javascript/libraries/dom.js
/javascript/app.js
```

En este caso, el manifiesto le indica al navegador que debe descargar y guardar en caché todos los archivos ubicados en esas rutas. Sin embargo, las capacidades de este manifiesto no se reducen a indicar la lista de archivos a guardar en caché, si no que permite indicar cuáles deben ser obtenidos siempre desde el servidor, así como indicar archivos que se utilizarán cuando la conexión falle.

```
CACHE MANIFEST

CACHE:
/
images/logo.png
images/sprites.png
styles/layout.css
scripts/libraries/dom.js
scripts/app.js

NETWORK:
```

```
*

FALLBACK:
/ /offline.html
```

Este nuevo manifiesto indica explícitamente cuáles son los archivos que deben ser guardados en caché (similar al primer manifiesto), así como los archivos que deben obtenerse del servidor (por defecto, todos los que no están definidos debajo de la línea `CACHE`), y define el archivo que el navegador debe usar en caso algún archivo no pueda ser obtenido.

Para que el navegador sepa dónde encontrar este archivo, debe ser incluido como atributo dentro de la etiqueta `<html>`:

```
<html manifest="manifest.appcache">
  ...
</html>
```

> *Soporte para Application Cache*

File

Con esta API podemos leer archivos que cargamos desde el navegador, mediante la etiqueta `<input type="file">`, así como al realizar operaciones *drag and drop* de manera nativa. De esta forma, podemos previsualizar imágenes antes de subirlas a un servidor o realizar operaciones con los archivos aunque la aplicación esté *offline*.

Cuando trabajamos con elementos `<input type="file">` podemos acceder a los archivos que han sido elegidos mediante la propiedad `files`, la cual es una lista instancia de `FileList`. Cada elemento de esta lista es un objeto instancia de `File` y tiene algunas propiedades:

Propiedad	Descripción
`name`	Nombre del archivo

Propiedad	Descripción
`size`	Tamaño en bytes del archivo
`type`	*MIME type* del archivo
`lastModifiedDate`	Última fecha de modificación del archivo

Sabiendo las propiedades que tienen estos objetos de *File API*, podemos crear un demo simple, empezando con el código HTML básico:

```
<input type="file" name="files" id="files" multiple>

<h4>Imágenes elegidas:</h4>
<div id="preview"></div>
```

Y luego utilizamos la API propiamente dicha:

```
var input = document.getElementById('files'),
    preview = document.getElementById('preview');

input.addEventListener('change', function(e) {
  var files = e.target.files;

  for (var i = 0; i < files.length; i++) {
    (function(file) {
      var reader = new FileReader(),
          img = document.createElement('img');

      img.width = 300;
      preview.appendChild(img);

      reader.addEventListener('load', function(e) {
        img.src = e.target.result;
      });
```

```
        reader.readAsDataURL(file);
      })(files[i]);
    }
  });
```

Dentro de este código de ejemplo utilizamos la función constructora `FileReader`, la cual permite leer las instancias de `File` y convertirlo a una cadena de tipo **Data URI** para, de esta forma, poder cargarlo en un elemento `<img>`.

En el bucle que lee cada imagen obtenida por el input `files` utilizamos una función inmediatamente invocada debido a la naturaleza asíncrona de `FileReader`. Con este tipo de funciones, se obliga al navegador a ejecutar todo el código dentro de la función antes de pasar a la siguiente iteración, lo que nos asegura que se lean los valores correctos para cada iteración.

> *Soporte para File*

File System

Esta API simula un sistema de archivos en el navegador, permitiendo crear, modificar y leer archivos mediante JavaScript. Este sistema de archivos simulado no es el sistema de archivos del sistema operativo, si no que está separado en un entorno controlado (a este tipo de entornos se le llama *sandbox*). Actualmente esta API está en fase experimental y está disponible en Chrome y Opera, por lo que tiene un uso potencial en aplicaciones para Chrome OS o aplicaciones web que funcionan con Chromium.

Al ser un entorno controlado, *File System API* tiene ciertas restricciones:

- **Cada origen tiene su sistema de archivos**: Un origen está formado por el protocolo, dominio y puerto de un documento. Similar a las APIs de Storage, cada origen tiene su propio sistema de archivos y no se pueden acceder entre sí.
- **No se pueden crear o renombrar archivos ejecutables**: Por seguridad, no se pueden crear archivos ejecutables, ya que estos pueden ser aplicaciones maliciosas (virus, troyanos, etc).

- **No se puede salir del** *sandbox*: Igualmente, por seguridad, una aplicación no puede usar la API para acceder a archivos que estén en el sistema de archivos del sistema operativo.
- **No se puede ejecutar desde el protocolo** `file://`: También por seguridad, si se tratar de utilizar esta API en un archivo desde `file://`, el navegador lanzará una excepción y fallará.

Adicionalmente, esta API tiene soporte para trabajar de forma síncrona y asíncrona, recomendando utilizar WebWorkers para el primer caso:

Interfaz	Descripción	Interfaz Síncrona
`LocalFileSystem`	Permite acceder al sistema de archivos controlado	`LocalFileSystemSync`
`FileSystem`	Representa un sistema de archivos	`FileSystemSync`
`Entry`	Representa una entrada en el sistema de archivos, el cual puede ser un archivo o un directorio	`EntrySync`
`DirectoryEntry`	Representa un directorio en el sistema de archivos	`DirectoryEntrySync`
`DirectoryReader`	Permite leer un directorio en el sistema de archivos	`DirectoryReaderSync`
`FileEntry`	Representa un archivo en el sistema de archivos	`FileEntrySync`
`FileError`	Error lanzado cuando falla el acceso al sistema de archivos	`FileException`

Para empezar a trabajar con esta API debemos pedirle al navegador que nos de un sistema de archivos para el origen en el cual estamos trabajando:

```
var requestFileSystem = window.requestFileSystem ||
window.webkitRequestFileSystem;

requestFileSystem(window.TEMPORARY, 1024 * 1024 * 5,
function(fileSystem) {
```

```
    console.log(fileSystem);
  }, function(error) {
    console.log(error);
  });
```

Donde `requestFileSystem` es una variable que guardará una referencia a `window.requestFileSystem` (de existir), o de `window.webkitRequestFileSystem` (en caso `window.requestFileSystem` no exista). Este tipo de asignaciones son comunes cuando se trabaja con APIs que aún no son estándares, ya que primero se busca la implementación *estándar*, y luego la implementación propia del navegador (la cual va acompañada de un prefijo, que puede ser `webkit`, `moz`, `ms` u `o`).

`requestFileSystem` es una función que permite obtener un sistema de archivos dentro del navegador, y tiene 4 parámetros:

- **Tipo de almacenamiento**: el cual puede ser `window.TEMPORARY` (el navegador puede borrar los archivos si necesita espacio) o `window.PERSISTENT` (solo el usuario puede borrar los archivos).
- **Tamaño en bytes**: El tamaño que se quiere asignar al sistema de archivos, el cual puede requerir un permiso explícito del usuario si el tamaño pedido es muy grande.
- **Callback de éxito**: Este callback toma un parámetro, el cual es una instancia de `DOMFileSystem` y tiene dos propiedades: `name` y `root` (instancia de `DirectoryEntry`)
- **Callback de error**: Este callback también toma un solo parámetro, el cual es una instancia de `FileError` y contiene 3 propiedades: el código del error, el nombre del error y un mensaje descriptivo.

> *Otro punto importante es ver cómo una variable guarda una referencia a una función. Recordemos que las funciones son ciudadanos de primera clase en JavaScript, por lo que es posible guardarlas en una variable, o pasarlas como parámetros (como en los dos últimos valores de `requestFileSystem`).*

Si se desea crear un sistema de archivos *persistente* se debe pedir una cuota de espacio al navegador:

```
window.webkitStorageInfo.requestQuota(window.PERSISTENT, 1024 *
1024 * 5, function(bytes) {
  window.webkitRequestFileSystem(window.PERSISTENT, bytes,
function(fileSystem) {
    console.log(fileSystem);
  }, function(error) {
    console.log('Error en requestFileSystem', error);
  });
}, function(error) {
  console.log('Error en requestQuota', error);
});
```

> *Chrome recomienda utilizar* `navigator.webkitTemporaryStorage` *o* `navigator.webkitPersistentStorage` *en vez de* `window.webkitStorageInfo` *para obtener la cuota de espacio. Ambos objetos siguen teniendo el método* `requestQuota`*.*

Escribiendo archivos

Luego de haber obtenido el sistema de archivos, podemos crear un archivo de la siguiente forma:

```
function successCallback(fileSystem) {
```

```
  fileSystem.root.getFile('demo.txt', { create : true },
function(fileEntry) {
    fileEntry.createWriter(function(writer) {
      writer.onwriteend = function(e) {
        console.log('Archivo creado.');
      };

      var blob = new Blob(['Texto', ' de ', 'prueba']);
      writer.write(blob);
    });
  });
}
```

Para crear un archivo tenemos que seguir dos pasos: obtener una referencia al archivo que queremos crear (con `getFile`), y crear una instancia de `FileWriter` (con `createWriter`).

`getFile` acepta 4 parámetros: El nombre del archivo, un objeto de opciones y dos callbacks, uno de éxito y otro de error. Es en el objeto de opciones donde se indica si el archivo se creará o editará (en ambos casos se utiliza `create : true`, pero si solo se quiere crear un archivo y evitar reescribir uno existente, se añade `exclusive : true`).

La instancia de `FileWriter` tiene diferentes *handlers* para manejar los eventos relacionados a la escritura del archivo, pero también tiene a `EventTarget` en su **cadena de prototypes**. Esto quiere decir que podemos utilizar los métodos `addEventListener` y `removeEventListener` para manejar los eventos de esta instancia.

Por último, para poder realizar la escritura del archivo, propiamente dicha, debemos crear una instancia de `Blob`, el cual tiene como primer parámetro un arreglo, el cual contiene las partes del contenido del archivo. Estas partes pueden ser cadenas, u otras instancias de `Blob`. Luego, se debe utilizar el método `write` de la instancia de `FileWriter` para escribir el *blob*.

Leyendo archivos

Para leer archivos también necesitamos obtener una referencia del archivo que deseamos leer; y para esto usamos el método `getFile`, solo que en este caso el segundo parámetro no tendrá ningúna propiedad.

```
function successCallback(fileSystem) {
  fileSystem.root.getFile('demo.txt', {}, function(fileEntry) {
    fileEntry.file(function(file) {
      var reader = new FileReader();

      reader.onloadend = function(e) {
        console.log(this.result);
      };

      reader.readAsText(file);
    });
  });
}
```

Luego de obtener la referencia del archivo debemos obtener el archivo en sí, mediante el método `file`, para luego crear una instancia de `FileReader`. Esta función, que ya ha sido utilizada por la *File API*, permite leer un archivo como texto plano, utilizando el método `readAsText`.

Actualizando archivos

Para actualizar un archivo debemos seguir los mismos que se utilizaron para crear y escribir un archivo nuevo, excepto por un par de cambios:

- El valor de `create` debe ser `false`.
- Mover la posición del cursor de la instancia de `File Writer` al final del archivo, utilizando el método `seek`.

```
function successCallback(fileSystem) {
  fileSystem.root.getFile('demo.txt', { create : false },
```

```
function(fileEntry) {
    fileEntry.createWriter(function(writer) {
      writer.seek(writer.length);

      writer.onwriteend = function(e) {
        console.log('Archivo actualizado.');
      };

      var blob = new Blob(['\n', 'Texto', ' de ', 'prueba']);
      writer.write(blob);
    });
  });
}
```

En este caso, cualquier *blob* que se escriba en el archivo va a sobreescribir el contenido que pueda existir en la posición que el cursor se encuentre (por defecto está en la posición `0`, al inicio del archivo). Es por eso que, en este caso, se pone el cursor al final del archivo.

Creando carpetas

Para crear una carpeta es necesario utilizar el método `getDirectory`, el cual es similar a `getFile` en cuanto a parámetros:

```
function successCallback(fileSystem) {
  fileSystem.root.getDirectory('examples', { create : true },
function(directoryEntry) {
    // directoryEntry
  });
}
```

De esta forma ya tenemos la carpeta creada. `directoryEntry` es una instancia de `DirectoryEntry` (recordemos: `fileSystem.root` también es una instancia de `DirectoryEntry`, por lo que se pueden realizar las mismas

operaciones que hemos visto anteriormente).

Obtener el contenido de una carpeta

Las instancias de `DirectoryEntry` tienen un método llamado `createReader`, el cual crea una instancia de `DirectoryReader`. Las instancias de `DirectoryReader` tienen un método llamado `readEntries`, el cual ejecuta dos callbacks, según si la operación ha sido exitosa o no, y devuelve la lista de *entradas* de una carpeta (una *entrada* puede ser un archivo o una carpeta).

```
function successCallback(fileSystem) {
  var directoryReader = fileSystem.root.createReader();
  directoryReader.readEntries(function(entries) {
    for (var i = 0; i < entries.length; i++) {
      console.log(entries[i]);
    }
  });
}
```

En este caso, es `fileSystem.root` quien crea una instancia de `DirectoryReader`, ya que queremos ver cuáles son los archivos y carpetas que se encuentran dentro de la carpeta raíz. Cabe notar que los elementos de *entries* puede ser instancias de `FileEntry` o de `DirectoryEntry`, dependiendo del tipo de entrada (archivo o carpeta, respectivamente).

Eliminando carpetas

Para eliminar una carpeta tenemos dos métodos de `DirectoryEntry`: `remove` y `removeRecursively`. El primer método solo podrá eliminar una carpeta si esta está vacía, mientras que el segundo método eliminará todo el contenido de la carpeta antes de eliminar la carpeta en sí.

```
function successCallback(fileSystem) {
```

```
    fileSystem.root.getDirectory('examples', {},
  function(directoryEntry) {
      directoryEntry.remove(function() {
        console.log('Carpeta eliminada');
      }, function() {
        console.log('La carpeta no pudo ser elimninada');
      });
    });
  }
```

El método `removeRecursively` funciona exactamente igual:

```
  function successCallback(fileSystem) {
    fileSystem.root.getDirectory('examples', {},
  function(directoryEntry) {
      directoryEntry.removeRecursively(function() {
        console.log('Carpeta eliminada');
      });
    });
  }
```

Cabe destacar que todos los métodos usados dentro de la *FileSystem API* toman dos callbacks: uno de éxito y otro de error, donde este último siempre recibirá un único parámetro con las causas del error. De esta forma, es posible crear una sola función que sirva como callback de error para todos los métodos de la *FileSystem API*, como en este código de ejemplo.

> *Soporte para File System*

History

Con la *History API* podemos simular entradas en el historial del navegador sin necesidad de realizar peticiones al servidor donde la aplicación está alojada (una entrada en el historial es cada página visitada en una pestaña de navegador).

Tradicionalmente, cuando un usuario ingresa a una dirección desde el navegador, o haciendo clic en un enlace, pasa lo siguiente:

1. El navegador realiza una petición al servidor al que apunta la dirección ingresada.
2. El servidor recibe la petición y la procesa, devolviendo una respuesta hacia el navegador.
3. El navegador muestra dicha respuesta al usuario final, lo cual también implica cambiar la dirección de la barra de direcciones del navegador mismo.
4. Se crea una entrada en el historial del navegador para la ventana actual. De esta forma el usuario sabe que está en una nueva página y que tiene la opciónde regresar a la anterior.

History API hace que estos pasos ya no sean obligatoriamente seguidos, ya que es posible cambiar la dirección de la barra de direcciones del navegador sin necesidad de hacer que el navegador envíe una petición al servidor, de tal forma que del flujo tradicional solo se ejecute el paso 4. Así mismo, ofrece un evento completamente nuevo, el cual se dispara cuando navegamos por las entradas del historial.

Agregando una entrada con `pushState`

El objeto `history` es el encargado de manejar el historial del navegador, y tiene algunos métodos como `back`, `forward` o `go` para navegar a través del historial, y una propiedad `length` que indica el número de entradas en el historial. A su vez, `history` tiene un método llamado `pushState`, el cual agrega una entrada al historial del navegador y cambia la dirección en la barra de direcciones del navegador, pero no realiza ninguna petición al servidor de la nueva dirección.

Suponiendo que los visitantes de **La Buena Espina** utilizan navegadores que soportan la *History API*, podemos hacer que los enlaces de la barra superior utilicen `pushState`, y de esta forma mostrar las secciones manipulando el DOM (para esto, todas las secciones deben estar previamente cargadas en la página):

```
var state = {
  prevURL: '/carta',
  actualURL: '/locales'
};

history.length;
// 1

history.pushState(state, 'Locales', '/locales');

history.length;
// 2
```

El método `pushState` toma 3 parámetros:
1. Un objeto representado el *state* o estado de la nueva entrada en el historial. Sirve para guardar información relacionada a la URL que se está agregando.
2. El nuevo título que tendrá la pestaña del navegador. Este parámetro es ignorado por algunos navegador, por lo que podría no ser útil de momento.
3. La URL que se agregará al historial. Este parámetro reemplazará todo lo que venga después del origen (un origen está conformado por el protocolo, el dominio y el puerto de una dirección).

Cabe notar aquí que ni `pushState` ni `replaceState` pueden poner una URL cuyo origen sea diferente al actual, esto es por un tema de seguridad: Por ejemplo, se podría tener un enlace que modifique la URL actual por la URL de un banco de confianza, pero sin la necesidad de cargar la web de dicho banco.

Así mismo, por seguridad, la *History API* no está disponible en archivos en local (es decir, aquellos que se ejecuten desde el protocolo `file://`).

Reemplazando una entrada con `replaceState`

Si con `pushState` podemos agregar una entrada al historial, con `replaceState` podemos reemplazar la entrada actual (es decir, donde estemos navegando actualmente).

Siguiendo con el ejemplo anterior, tenemos que la dirección actualmente es `/locales`, y queremos que al buscar locales por distrito, cambie la dirección pero no agregue una entrada más en el historial:

```
var state = {
  prevURL: '/carta',
  actualURL: '/locales?buscar_en=Lince',
};

history.length;
// 2

history.replaceState(state, 'Locales', '/locales?buscar_en=Lince');

history.length;
// 2
```

El método `replaceState` toma los mismos parámetros que `pushState` y sirve, principalmente, para actualizar la entrada actual con algunos valores propios de la interacción del usuario con el sitio web.

Evento `popstate`

El evento `popstate` es lanzado cada vez que se *viaja* a través del historial, ya sea con los botones del navegador, o con los métodos `back`, `forward` o `go` de `history`.

Por ejemplo, para conocer el *state* de la entrada del historial a la cual se navegó, se puede utilizar el siguiente código:

```
window.addEventListener('popstate', function(e) {
  console.log(e.state);
});
```

Dado que este evento corresponde a la pestaña (o ventana) actual, es `window` el encargado de *escuchar* el evento.

> *Soporte para History*

WebSocket

Los websockets permiten una comunicación bi-direccional entre el navegador y el servidor, de tal forma que este puede enviarnos datos sin necesidad de hacerle una petición (como ocurre en un modelo tradicional). Así, no solo podemos enviarle información al servidor, si no que podemos estar a la espera de *escuchar* los datos que el servidor pueda mandar por su cuenta.

Para poder utilizar websockets necesitamos tener un servidor de websockets, al cual se accede mediante el protocolo `ws://`, o `wss://` en caso de querer una conexión segura. Existen bibliotecas para crear servidores de websockets en varios lenguajes.

> *A grandes rasgos, el navegador abre conexiones HTTP de petición/respuesta: iniciará enviando una petición hacia el servidor y este enviará una respuesta; y cuando la respuesta ha sido enviada por el servidor, la conexión se cierra. Pero puede existir el caso donde se necesiten enviar peticiones o recibir respuestas sucesivamente (como tener notificaciones en una aplicación web o un chat en tiempo real), y abrir y cerrar conexiones puede demorar mucho. es aquí donde aparece WS: En WS se crea una conexión y se deja abierta hasta que alguna de las dos partes cierre la conexión, no importa cuantos mensajes se manden entre sí.*

En el navegador solo necesitamos crear una instancia de `WebSocket`, pasándole la url del servidor de websockets:

```
var connection = new WebSocket('ws://html5rocks.websocket.org
/echo');

connection.onopen = function(e) {
```

```
  console.log('Connected');
};

connection.onclose = function(e) {
  console.log('Disconnected');
};

connection.onerror = function(e) {
  console.log('An error ocurred');
};

connection.onmessage = function(e) {
  console.log('Message received: ', e.data);
};
```

Estas 4 funciones definen 4 *handlers* para 4 eventos diferentes:

- `open`: Este evento es lanzado cuando se logra abrir una conexión con el servidor de websocket (en este caso con `ws://html5rocks.websocket.org/echo`)
- `close`: Es lanzado cuando se cierra una conexión, ya sea del lado del navegador o del servidor.
- `error`: Este evento es lanzado cuando existe un error en la conexión o en alguno de los lados de la comunicación.
- `message`: Es lanzado cuando llega un mensaje desde el otro lado de la comunicación (en este caso, del servidor). El parámetro que recibe el *handler* tiene una propiedad llamada `data`, el cual contiene el mensaje que el servidor envió.

`WebSocket` tiene a `EventTarget` en su **cadena de prototypes** (que es como se define la *herencia* en JavaScript), por lo que podemos usar `addEventListener` en la variable `connection`:

```
var connection = new WebSocket('ws://html5rocks.websocket.org
/echo');
```

```
connection.addEventListener('open', function(e) {
  console.log('Connected');
});

connection.addEventListener('close', function(e) {
  console.log('Disconnected');
});

connection.addEventListener('error', function(e) {
  console.log('An error ocurred');
});

connection.addEventListener('message', function(e) {
  console.log('Message received: ', e.data);
});
```

Con este código tenemos lo básico para poder escuchar los datos que el servidor mande, pero si queremos enviarle información al servidor, debemos utilizar el método `send`:

```
connection.send('Hi from La Buena Espina');
```

Hay que tener algunas consideraciones al momento de utilizar un servidor de websockets. Por ejemplo, si se usan websockets en una web que usa HTTPS, las conexiones al servidor de websockets deben ser con WSS.

A diferencia de HTTP(S), los protocolos de websockets no cambian automáticamente (un protocolo cambia automáticamente cuando tenemos una imagen cuya url es `//labuenaespina.pe/logo.png` y según la página actual puede cargar `http://labuenaespina.pe/logo.png` o `https://labuenaespina.pe/logo.png`); por lo que se debe cambiar manualmente:

```
var websocketURL = '//html5rocks.websocket.org/echo';
```

```
if (location.protocol === 'http:') {
  websocketURL = 'ws:' + websocketURL;
}
else if (location.protocol === 'https:') {
  websocketURL = 'wss:' + websocketURL;
}

var connection = new WebSocket(websocketURL);
```

Opcionalmente, en el servidor se puede restringir si se aceptan o no conexiones de diferentes orígenes de websockets (mediante el Content Security Policy, específicamente la directiva `connect-src`).

> *Soporte para Websocket*

Server-Sent Event

La *Server-Sent Event API* es una alternativa para los websockets, ya que permite que el navegador esté *escuchando* los datos que un servidor pueda mandar. En este caso, la API utiliza el protocolo HTTP(S), en comparación al protocolo WS que es utilizado por los websockets. Sin embargo, solo se pueden *escuchar* datos, mas no enviar datos al servidor, por lo que puede ser utilizado en casos donde no es necesario o se quiere evitar que el navegador envíe datos al servidor.

Para abrir una conexión al servidor debemos crear una instancia de `EventSource`:

```
var sseConnection = new EventSource('/sse_stream');

sseConnection.addEventListener('open', function(e) {
  console.log('Connected');
});

sseConnection.addEventListener('close', function(e) {
```

```
  console.log('Disconnected');
});

sseConnection.addEventListener('error', function(e) {
  console.log('An error ocurred');
});

sseConnection.addEventListener('message', function(e) {
  console.log('Message received: ', e.data);
});
```

Una de las ventajas que tiene esta API es que podemos *escuchar* eventos propios, los cuales deben ser generados por el servidor mismo. Por ejemplo, podríamos tener un *feed* de eventos en La Buena Espina que indique cuando una mesa ha sido reservada:

```
sseConnection.addEventListener('booked_table', function(e) {
  var tableId = e.data;
  console.log('La mesa ' + tableId + ' ha sido reservada');
});
```

Hay que tener en cuenta que tanto para websockets como para server-sent events la información que recibimos del servidor (o que enviamos, en el caso de websockets) es una cadena, por lo que, de ser el caso, se deben hacer las conversiones necesarias, o utilizar JSON si se trabaja con arreglos u objetos.

> *Soporte para Server-Sent Event*

← 3. Capítulo 3: DOM y CSSOM 5. Capítulo 5: Pruebas →

Acerca de | Disponible en Amazon.com