APIs de JavaScript

Antonio Espín Herranz

Relación de APIs

- **Forms**:
  - Centrada en la validación y construcción de validadores de formularios.
- **Canvas**:
  - API gráfica, proporciona una superficie de dibujo gráfico. Permite generar e imprimir gráficos.
  - Canvas genera imágenes a partir de pixeles que se pueden manipular por medio de funciones gráficas.
- **Drag and Drop**:
  - Incorpora la posibilidad de arrastrar y soltar elementos en pantalla.
- **Geolocation**:
  - Establecer la ubicación física del dispositivo usado para acceder a la aplicación.
- **Storage**:
  - Almacenamiento: Web Storage e Indexes Database.

Relación de APIs

- **File**:
  - Operaciones relacionadas con archivos.
- **Sistema de archivos:** *Descatalogado → LocalStorage*
- **Communication***: XMLHttpRequest Descatalogado → FETCH*
- **Web Workers**:
  - Ejecución de tareas en 2º plano.
- **WebSockets**
  - Establecer conexiones entre el navegador y un servidor

Detectar soporte para HTML5

- Podemos utilizar una **librería** (**Modenizr – NO forma parte de la especificación**)
  - http://modernizr.com/download/
- Para comprobar si tenemos soporte para objetos y propiedades en HTML5.

  <script src="modernizr.min.js"></script>

  – En código:

```
if (Modernizr.canvas) {
    // a crear formas!!
} else {
    // no hay soporte para canvas, lo siento:
}
```

API Geolocation

- Proporciona unos métodos para detectar la ubicación física real del usuario.
- Dispone de **3 métodos**:
  - getCurrentPosition(ubicación, error, configuración);
  - watchPosition(ubicación, error, configuración);
  - clearWatch(id);
  - Para llamar a estos métodos los haremos a partir de:
    - **navigator.geolocation**.método()
- Y unos **objetos**:
  - Position.
  - PositionError.
  - Configuración.

getCurrentPosition

- Tiene 3 parámetros sólo el primero es obligatorio.
  - **Ubicación**: Es una **función de JavaScript** que recibirá un objeto posición (tiene dos atributos):
    - **coords**: Establece la ubicación del dispositivo.
      - **latitude**, **longitude**, **altitude**, **accuracy** (exactitud en metros), **altitudeAccuracy** (exactitud de la altitud en metros), **heading** (dirección en grados) y **speed** (velocidad en metros por sg).
    - **timestamp**: en que momento fue adquirida la información.

getCurrentPosition

- Esquema de la llamada:

```
navigator.geolocation.getCurrentPosition(mostrarInfo);

function mostrarInfo(posicion){
    alert(posicion.coords.latitude + " " +
      posicion.coords.longitude);
}
```

getCurrentPosition

- Podemos indicar una función error, si utilizamos el 2º parámetro:
  - getCurrentPosition(ubicación,errores):
    - Ambos parámetros son funciones de Javascript.
  - Este método devuelve el objeto: PositionError si se detecta un error y se envía a la función que hayamos declarado en el 2º parámetro.

```
function errores(error){
    alert("ERROR: " + error.code + " " + error.message);
}
```

PositionError

- Este objeto tiene dos atributos:
  - code:
    - PERMISSION_DENIED:
      - Valor:1, si el usuario no acepta que le localicen.
    - POSITION_UNAVAILABLE:
      - Valor:2, no se puede determinar la ubicación del dispositivo.
    - TIMEOUT:
      - Valor:3, tiempo excedido.
  - message:
    - Mensaje asociado al error.

getCurrentPosition

- Podemos utilizar un objeto de configuración si utilizamos el **3º parámetro:**
  - getCurrentPosition(ubicación, error, **configuración**).
  - El tercer parámetro será un objeto de tipo **Configuration**.

Objeto Configuration

- Este objeto tiene **3 atributos**:
  - **enableHighAccuracy** (boolean):
    - Requerimos la información más exacta que nos pueda proporcionar.
    - OJO, consume muchos recursos sobre todo en móviles.
    - El valor normal suele ser false.
  - **timeout**: (número en milisegundos):
    - Tiempo máximo para que finalice la operación.
  - **maximumAge** (número en milisegundos):
    - Las ubicaciones encontradas se almacenan en una caché, evita consumo de recursos, si la última ubicación es más vieja que este parámetro se solicita una nueva ubicación al sistema.

Ejemplo

```
function obtenerUbicacion(){
    var geoConfig = {
        enabledHighAccuracy: true,
        timeout: 10000,
        maximumAge: 60000
    };
    navigator.geolocation.getCurrentPosition(mostrar,
      errores, geoConfig);
}
```

watchPosition

- Tiene los **mismos parámetros** que la anterior, pero ofrece nuevos datos cuando la ubicación cambia (pensada para **dispositivos móviles**).
- Comportamiento similar a la función setInterval de javascript.
- Devuelve un **identificador** que lo utilizaremos para detenerla, llamando a la función **clearWatch**().

Ejemplo

```
var id =
   navigator.geolocation.watch(mostrar,
   errores, geoConfig);

// Cuando la llamamos indicamos el id.
clearWatch(id);
```

Uso de openStreetMap

```
const status = document.querySelector("#status"); → párrafo
 const mapLink = document.querySelector("#map-link"); → enlace

 mapLink.href = "";
 mapLink.textContent = "";

 mapLink.href =
`https://www.openstreetmap.org/#map=18/${latitud}/${longitud}`;
 mapLink.textContent = `Latitude: ${latitud} °, Longitude: ${longitud} °`;
```

API Canvas

- Necesitamos definir una etiqueta Canvas dentro de la página.

  <canvas id=“miCanvas” height=“200px” height=“200px”></canvas>

- Recuperar el canvas con JS:

  var c=document.getElementById("myCanvas");

- Capturar un contexto en 2D para dibujar gráficos:

  var cxt=c.getContext("2d");

- Pintar un rectángulo relleno de color rojo:

  cxt.fillStyle="#FF0000";

  cxt.fillRect(0,0,150,75);

API Canvas

- Una forma de indicar que el navegador no soporta canvas es:

```
<canvas id="miCanvas" height="200px" height="200px">
    Su navegador no soporta Canvas
</canvas>
```

– De esta forma si el navegador no soporta Canvas mostrará el mensaje.

API Canvas: Referencia

getContext(contexto) Este método crea el contexto para el lienzo. Puede tomar dos valores: `2d` y `3d` para gráficos en 2 y 3 dimensiones.

fillRect(x, y, ancho, alto) Este método dibujará un rectángulo sólido directamente en el lienzo en la posición indicada por `x,y` y el tamaño `ancho,alto`.

strokeRect(x, y, ancho, alto) Este método dibujará un rectángulo vacío (solo el contorno) directamente en el lienzo en la posición indicada por `x,y` y el tamaño `ancho,alto`.

clearRect(x, y, ancho, alto) Este método borra un área en el lienzo usando una figura rectangular declarada por los valores de sus atributos.

createLinearGradient(x1, y1, x2, y2) Este método crea un gradiente lineal para asignarlo a una figura como si fuese un color usando la propiedad `fillStyle`. Sus atributos solo especifican las posiciones de comienzo y final del gradiente (relativas al lienzo). Para declarar los colores involucrados en el gradiente, este método debe ser usado en combinación con `addColorStop()`.

createRadialGradient(x1, y1, r1, x2, y2, r2) Este método crea un gradiente radial para asignarlo a una figura como si fuese un color usando la propiedad `fillStyle`. El gradiente es construido por medio de dos círculos. Los atributos solo especifican la posición y radio de los círculos (relativos al lienzo). Para declarar los colores involucrados en el gradiente, este método debe ser usado en combinación con `addColorStop()`.

addColorStop(posición, color) Este método es usado para declarar los colores para el gradiente. El atributo `posición` es un valor entre 0.0 y 1.0, usado para determinar dónde el color comenzará la degradación.

API Canvas: Referencia

beginPath() Este método es requerido para comenzar un nuevo trazado.

closePath() Este método puede ser usado al final de un trazado para cerrarlo. Generará una línea recta desde la última posición del lápiz hasta el punto donde el trazado comenzó. No es necesario usar este método cuando el trazado debe permanecer abierto o es dibujado en el lienzo usando `fill()`.

stroke() Este método es usado para dibujar un trazado como una figura vacía (solo el contorno).

fill() Este método es usado para dibujar un trazado como una figura sólida.

clip() Este método es usado para crear una máscara a partir de un trazado. Todo lo que sea enviado al lienzo luego de que este método es declarado será dibujado sólo si cae dentro de la máscara.

moveTo(x, y) Este método mueve el lápiz virtual a una nueva posición para continuar el trazado desde ese punto.

lineTo(x, y) Este método agrega líneas rectas al trazado desde la posición actual del lápiz hasta el punto indicado por los atributos `x` e `y`.

rect(x, y, ancho, alto) Este método agrega un rectángulo al trazado en la posición `x,y` y con un tamaño determinado por `ancho,alto`.

API Canvas: Referencia

arc(x, y, radio, ángulo inicio, ángulo final, dirección) Este método agrega un arco al trazado. El centro del arco es determinado por `x` e `y`, los ángulos son definidos en radianes, y la `dirección` es un valor booleano para determinar si el arco será dibujado en el mismo sentido o el opuesto a las agujas del reloj. Para convertir grados en radianes, use la fórmula: `Math.PI/180xgrados`.

quadraticCurveTo(cpx, cpy, x, y) Este método agrega una curva Bézier cuadrática al trazado. Comienza desde la posición actual del lápiz y termina en el punto `x,y`. Los atributos `cpx` y `cpy` especifican la posición del punto de control que dará forma a la curva.

bezierCurveTo(cp1x, cp1y, cp2x, cp2y, x, y) Este método agrega una curva Bézier cúbica al trazado. Comienza desde la posición actual del lápiz y termina en el punto `x,y`. Los atributos `cp1x`, `cp1y`, `cp2x`, y `cp2y` especifican la posición de los dos puntos de control que darán forma a la curva.

strokeText(texto, x, y, máximo) Este método dibuja un texto vacío (solo el contorno) directamente en el lienzo. El atributo `máximo` es opcional y determina el máximo tamaño del texto en pixeles.

fillText(texto, x, y, máximo) Este método dibuja un texto sólido directamente en el lienzo. El atributo `máximo` es opcional y determina el máximo tamaño del texto en pixeles.

measureText(texto) Este método calcula el tamaño del área que un texto ocupará en el lienzo usando los estilos vigentes. La propiedad `width` es usada para retornar el valor.

translate(x, y) Este método mueve el origen del lienzo al punto `x,y`. La posición inicial del origen (0,0) es la esquina superior izquierda del área generada por el elemento `<canvas>`.

API Canvas: Referencia

translate(x, y) Este método mueve el origen del lienzo al punto `x,y`. La posición inicial del origen (0,0) es la esquina superior izquierda del área generada por el elemento `<canvas>`.

rotate(angle) Este método es usado para rotar el lienzo alrededor del origen. El ángulo debe ser declarado en radianes. Para convertir grados en radianes, use la fórmula: `Math.PI/180xgrados`.

scale(x, y) Este método cambia la escala del lienzo. Los valores por defecto son (1.0, 1.0). Los valores provistos pueden ser negativos.

transform(m1, m2, m3, m4, dx, dy) Este método modifica la matriz de transformación del lienzo. La nueva matriz es calculada sobre la anterior.

setTransform(m1, m2, m3, m4, dx, dy) Este método modifica la matriz de transformación del lienzo. Reinicia los valores anteriores y declara los nuevos.

save() Este método graba el estado del lienzo, incluyendo la matriz de transformación, propiedades de estilo y la máscara.

restore() Este método restaura el último estado del lienzo grabado, incluyendo la matriz de transformación, propiedades de estilo y la máscara.

drawImage() Esta método dibujará una imagen en el lienzo. Existen tres sintaxis posibles. La sintaxis `drawImage(imagen,x,y)` dibuja la imagen en la posición `x,y`. La sintaxis `drawImage(imagen,x,y,ancho,alto)` dibuja la imagen en la posición `x,y` con un nuevo tamaño declarado por `ancho,alto`. Y la sintaxis `drawImage(imagen, x1, y1, ancho1, alto1, x2, y2, ancho2, alto2)` toma una porción de la imagen original determinada por `x1,y1,ancho1,alto1` y la dibuja en el lienzo en la posición `x2,y2` y el nuevo tamaño `ancho2,alto2`.

getImageData(x, y, ancho, alto) Este método toma una porción del lienzo y la graba como datos en un objeto. Los valores del objeto son accesibles a través de las propiedades `width`, `height` y `data`. Las primeras dos propiedades retornan el tamaño de la

API Canvas: Referencia

porción de la imagen tomada, y `data` retorna la información como un array con valores representando los colores de cada pixel. Este valor puede ser accedido usando la fórmula `(anchox4xy)+(xx4)`.

putImageData(datosImagen, x, y) Este método dibuja en el lienzo la imagen representada por la información en `datosImagen`.

createImageData(ancho, alto) Este método crea una nueva imagen en formato de datos. Todos los pixeles son inicializados en color negro transparente. Puede tomar datos de imagen como atributo en lugar de `ancho` y `alto`. En este caso la nueva imagen tendrá el tamaño determinado por los datos provistos.

createPattern(imagen, tipo) Este método crea un patrón desde una imagen que luego podrá ser asignado a una figura usando la propiedad `fillStyle`. Los valores posibles para el atributo `tipo` son `repeat`, `repeat-x`, `repeat-y` y `no-repeat`.

API Canvas: Referencia (propiedades)

strokeStyle Esta propiedad declara el color para las líneas de las figuras. Puede recibir cualquier valor CSS, incluidas funciones como `rgb()` y `rgba()`.

fillStyle Esta propiedad declara el color para el interior de figuras sólidas. Puede recibir cualquier valor CSS, incluidas funciones como `rgb()` y `rgba()`. Es también usada para asignar gradientes y patrones a figuras (estos estilos son primero asignados a una variable y luego esa variable es declarada como el valor de esta propiedad).

globalAlpha Esta propiedad es usada para determinar el nivel de transparencia de las figuras. Recibe valores entre 0.0 (completamente opaco) y 1.0 (completamente transparente).

lineWidth Esta propiedad especifica el grosor de la línea. Por defecto el valor es 1.0.

lineCap - Esta propiedad determina la forma de la terminación de las líneas. Se pueden utilizar tres valores: `butt` (terminación normal), `round` (termina la línea con un semicírculo) y `square` (termina la línea con un cuadrado).

lineJoin Esta propiedad determina la forma de la conexión entre líneas. Se pueden utilizar tres valores: `round` (la unión es redondeada), `bevel` (la unión es cortada) y `miter` (la unión es extendida hasta que ambas líneas alcanzan un punto en común).

miterLimit Esta propiedad determina cuánto se extenderán las líneas cuando la propiedad `lineJoin` es declarada como `miter`.

font Esta propiedad es similar a la propiedad `font` de CSS y utiliza la misma sintaxis para declarar los estilos del texto.

textAlign Esta propiedad determina cómo el texto será alineado. Los posibles valores son `start`, `end`, `left`, `right` y `center`.

textBaseline Esta propiedad determina el alineamiento vertical para el texto. Los posibles valores son: `top`, `hanging`, `middle`, `alphabetic`, `ideographic` y `bottom`.

shadowColor Esta propiedad establece el color para la sombra. Utiliza valores CSS.

shadowOffsetX Esta propiedad declara la distancia horizontal entre la sombra y el objeto.

API Canvas: Referencia (propiedades)

shadowOffsetY Esta propiedad declara la distancia vertical entre la sombra y el objeto.

shadowBlur Esta propiedad recibe un valor numérico para generar un efecto de difuminación para la sombra.

globalCompositeOperation Esta propiedad determina cómo las nuevas figuras serán dibujadas en el lienzo considerando las figuras ya existentes. Puede recibir varios valores: `source-over`, `source-in`, `source-out`, `source-atop`, `lighter`, `xor`, `destination-over`, `destination-in`, `destination-out`, `destination-atop`, `darker` y `copy`. El valor por defecto es `source-over`, lo que significa que las nuevas formas son dibujadas sobre las anteriores.

PRÁCTICA CANVAS

API Drag & Drop

- Esta API soporta arrastrar y soltar como las aplicaciones de Escritorio.
- Eventos:
  - En el elemento **origen**:
    - **dragstart**: El evento salta cuando comienza el arrastre.
    - **drag**: Equivalente a **mousemove**, pero cuando arrastramos.
    - **dragend**: Cuando finaliza la operación de arrastrar y soltar.
  - En el elemento **destino**:
    - **dragenter**: Cuando el puntero del ratón entra dentro del área ocupada por los posibles elementos destino durante una operación de arrastrar y soltar.
    - **dragover**: Equivalente a mousemove, disparado durante una operación de arrastre por posibles elementos destino.
    - **drop**: Cuando se suelta el elemento en destino.
    - **dragleave**: Se dispara cuando el elemento sale del área ocupada por un elemento durante una operación de arrastrar y soltar.

API Drag & Drop

- Tener en cuenta que en las operaciones drag & drop los navegadores realizan acciones por defecto en los eventos:
  - **dragenter**, **dragover** y **drop**.
  - Esto implica llamar al método **preventDefault** sobre el objeto que representa el evento.
    - Cancela el comportamiento por defecto.

dataTransfer

- Representa el objeto que almacena la información de una operación de arrastrar y soltar.
- **setData(tipo,dato)**
  - Este método se usa para declarar datos a ser enviados y su tipo.
- **getData(tipo)**
  - Recupera datos enviados por el origen, pero sólo del tipo especificado.
- **clearData()**
  - Elimina los datos.

Ejemplo

- Arrastrar una imagen a un contenedor:

```
<section id="dropbox">  DESTINO
    Arrastrar y soltar una imagen...
</section>
<section id="picturesbox"> ORIGEN
    <img id="image" src="img_logo.gif">
</section>
```

Ejemplo

```
<script>
var source1, drop;

function initiate(){
  source1 = document.getElementById('image');
  source1.addEventListener('dragstart',  dragged);

  drop = document.getElementById('dropbox');
  drop.addEventListener('dragenter',  function(e){ e.preventDefault(); });
  drop.addEventListener('dragover',  function(e){ e.preventDefault(); });
  drop.addEventListener('drop',  dropped);
}

function dragged(e){
  var code = '<img src="' + source1.getAttribute('src') + '">';
  e.dataTransfer.setData('Text',  code);
}

function dropped(e){
  e.preventDefault();
  drop.innerHTML  = e.dataTransfer.getData('Text');
  source1.parentNode.removeChild(source1);
}

addEventListener('load',  initiate);
</script>
```

API Web Storage

- Este API representa una mejora de las cookies:
  - Ficheros de texto con pares de clave / valor, para almacenar datos en el cliente.
- Almacena datos en el disco duro del usuario.
- **Dos posibilidades**:
  - Datos disponibles para la sesión → **sessionStorage**
  - Datos persistentes → **localStorage**

sessionStorage

- Los datos s**ólo** disponibles para la **sesión en curso** → **TEMPORAL**

- **Almacenar** datos:
  - sessionStorage.**setItem**(keyword, value);
- **Recuperar** datos:
  - var value = sessionStorage.**getItem**(keyword);

sessionStorage

- **Listar** los datos almacenados:

```
for(var f = 0; f < sessionStorage.length; f++){
    var keyword = sessionStorage.key(f);
    var value = sessionStorage.getItem(keyword);
   alert(value);
}
```

- **Borrar** datos almacenados:
  - sessionStorage.**removeItem**(keyword);
- **Borrar todo**:
  - sessionStorage.**clear**();

localStorage

- Los datos se **almacenan de forma permanente** → **PERMANENTE**

- Los mismos métodos, pero **sustituyendo sessionStorage** por **localStorage**.

API File

- Este API nos permite:
  - Interactuar con archivos locales y procesar su contenido en nuestra aplicación.
  - Proporciona herramientas para trabajar con un pequeño sistema de archivos.
  - Y escribir el contenido de un archivo.

- Los archivos se gestionan a partir de la etiqueta input type=“file”

API File

- Nos permite leer archivos de tipo texto o binarios (por ejemplo, imágenes).
- Obtener propiedades de los archivos como son nombre, tipo, tamaño.
- Este API crea un espacio dentro del disco duro y sólo es accesible a la aplicación que lo creo.

Leer Archivos

- Por un lado, necesitamos el form con la etiqueta file.

```
<form name="form">
    <label for="myfiles">File: </label><br>
    <input type="file" name="myfiles" id="myfiles">
</form>
```

  - Se puede utilizar el atributo multiple para cargar varios archivos.
- A nivel de JavaScript:
  - La interface **FileReader** para leer un archivo.
    - **readAsText**(archivo, codificacion): La codificación UTF-8 (por defecto).
    - **readAsBinaryString**(archivo): para fichero binarios.
    - **readAsDataURL**(archivo): cadena codificada en Base64 que representa el archivo. Para imágenes.

Leer Archivos

```
var databox;

function initiate(){
  databox = document.getElementById('databox');
  var myfiles = document.getElementById('myfiles');
  myfiles.addEventListener('change', process);
}

function process(e){
  var files = e.target.files;                    // Capturar los archivos.
  var myfile = files[0];                         // Capturar el primero.
  var reader = new FileReader();
  reader.addEventListener('load', show);         // El evento load saltará cuando esté leído el archivo.
  reader.readAsText(myfile);
}

function show(e){
  var result = e.target.result;
  databox.innerHTML = result;
}

addEventListener('load', initiate);
```

Propiedades del Archivo

- Del archivo se pueden obtener las siguientes propiedades:
  - **name**: El nombre del archivo.
  - **size**: El tamaño en bytes.
  - **type**: El tipo de archivo. Tipo MIME.

Leer Imágenes

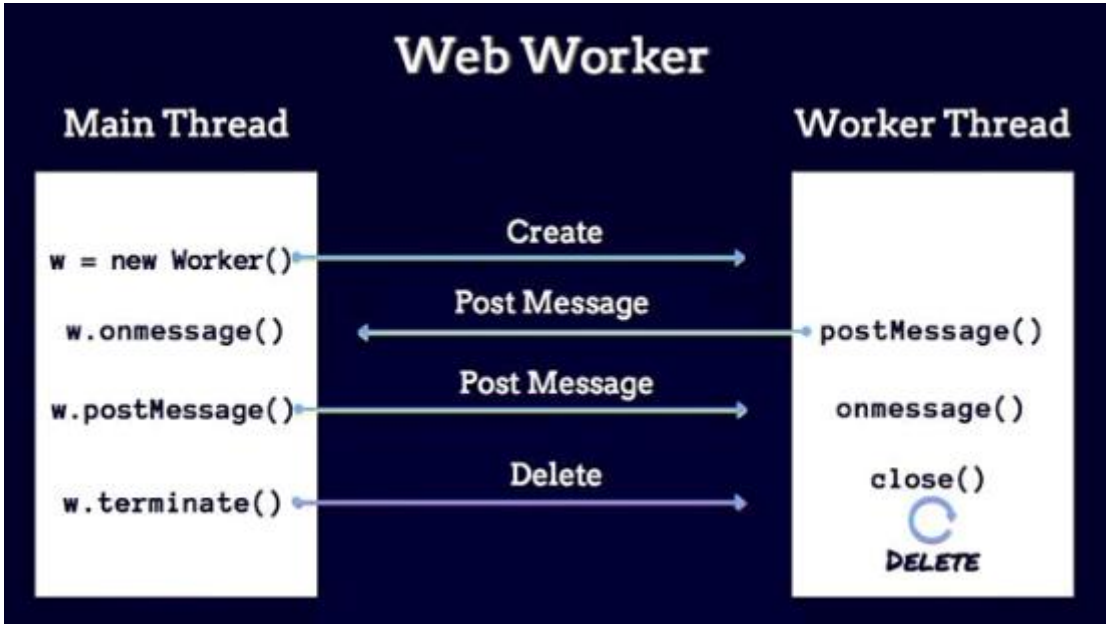
- En este caso tenemos que utilizar el método readAsDataURL y como parámetro le pasaremos el file seleccionado dentro de la etiqueta input type="file" y configuramos un evento change para poder cargar el archivo dentro de la función asociada al evento.

- Tener en cuenta que al mostrar el resultado tenemos que utilizar un tag: img.

PRÁCTICA FILE

WebWorker

- Tareas en segundo plano, en otro distinto del principal:



WebWorker

- Web Workers API es probablemente una de las API más simples.

- Tiene métodos bastante sencillos para crear subprocesos de trabajo y comunicarse con ellos desde el script principal.

WebWorker

- Tipos:
  - worker dedicados y worker compartidos.
  - Los **worker dedicados** pertenecen al mismo contexto de navegación al que pertenece su hilo principal.
  - Los **worker compartidos**, sin embargo, están presentes en un contexto de navegación diferente (por ejemplo: en un iframe) del script principal.

WebWorker

- El constructor **Worker**() crea un hilo de trabajo dedicado y devuelve su objeto de referencia. Luego, usamos este objeto para comunicarnos con ese worker específico.

- El método **postMessage**() se usa en las secuencias de comandos principal y de worker para enviar datos entre sí.
  - Los datos enviados se reciben en el otro lado por el controlador de eventos **onmessage**.

- El método **terminate**() termina un hilo del worker desde el script principal. Esta finalización es inmediata: cualquier ejecución actual de scripts y scripts pendientes se cancelarán.

- El método **close**() hace lo mismo, pero es llamado por el hilo del worker y se cierra.

WebWorker

- En el script del principal (en index.html) se crea el worker.
- Constructor:
  - w = new **Worker**(“path al js del worker”)
  - Añadir un evento: **onmessage** para recibir datos del worker.
    - Recibe un objeto: con la propiedad **data** accedemos al contenido del mensaje.
  - Cuando queremos enviar mensajes al worker utilizamos **postMessage**.

WebWorker

- El worker envía mensajes al hilo principal con **postMessage** y también recibe con **onmessage**.

WebSocket

- El API de sockets nos permite conectar el navegador con un servidor por medio de una conexión persistente.

- Es bueno para servicios que requieren intercambio de información continua.

- Para crearlo:
  - let socket = new WebSocket(“url-servidor");

WebSockets

- Una vez establecida la conexión tenemos que escuchar eventos:
  - open – conexión establecida,
  - message – datos recibidos,
  - error – error en websocket,
  - close – conexión cerrada.

WebSockets

- socket.addEventListener("open", función_open);
- socket.addEventListener("message", función_message);
- socket.addEventListener("error", función_error);
- socket.addEventListener("close", función_close);

WebSockets

- Enviar mensajes:
  - socket.send(“mensaje”);

- Recibir mensajes (con una función asíncrona)
  - **async** function message(evento) {
  - const mensajeRecibido = **await evento.data**;
  - }

Servidor

- Podemos hacer pruebas creando un servidor **HTTP** con **Node**.

- Crear un proyecto con Node.
  - Crear la carpeta para el servidor
  - Crear el proyecto: **npm init**
  - Instalar la dependencia ws:
    - **npm install ws**

Servidor

- Necesitamos importar ws y http
  - const WebSocket = require("ws");
  - const http = require("http");

- Creamos el servidor http y a partir de este el servidor de Web Socket
  - const server = http.createServer();
  - const wss = new WebSocket.Server({ server });

Servidor

- Tenemos que establecer eventos en el servidor de Web Sockets con el método on.

- El primer evento a establecer es “connection” y recibimos como parámetro el socket con el que conecta el cliente.

- A partir de este socket podemos recuperar información de la conexión del cliente como puede ser el puerto y la IP.

Servidor

- **Eventos**:
  - wss.on("**connection**", (socket) => { …});
  - **wss**: Es el servidor de Web Sockets
    - Escucha para recibir conexiones de los clientes.
  - **socket**: Es el socket del cliente que se conecta.
- Después el servidor ejecuta el método **listen** indicando el puerto de escucha.
  - El puerto debe ser > 1024 (Servicios de S.O.)

Servidor

- **Otros eventos** a registrar son: **message** y **close** pero se registran en el socket del cliente y dentro del evento connection:

```
socket.on("message", (data) => {
    // El cliente envía un mensaje al servidor
    // data es el contenido del mensaje: data.toString()
}
socket.on("close", ()=>{
    // El cliente se desconecta del servidor
})
```

Servidor

- Propiedades del cliente que se conecta:
  - La **IP**: socket._socket.**remoteAddress**
  - El **puerto**: socket._socket.**remotePort**

- El servidor tiene acceso a todos los clientes que están conectados: **wss.clients**

- Puede **enviarles** un **mensaje** con el método **send**.

Servidor

- Acceso a los clientes:

```
wss.clients.forEach((cliente) => {
    if (cliente.readyState === WebSocket.OPEN) {
      cliente.send("mensaje")
    }
}
```

  - // Comprobar el estado del cliente: ¿está conectado?

API Forms

- Este API nos permite gestionar la validación de los formularios:
  - Podemos crear mensajes de validación personalizados.
    - Por ejemplo, si no podemos utilizar el atributo **required**. El usuario rellena un formulario con dos teléfonos tiene que rellenar uno de los dos (el que quiera) pero al menos tiene que rellenar uno.
    - Utilizaremos:
      - **setCustomValidity**('El mensaje personalizado');
      - Si lo pasamos en blanco no se mostrará.

API Forms

- setCustomValidity se aplica a un control:
  - var tno = document.getElementById(‘tno’);
  - tno.setCustomValidity(‘no puede ser vacío’);

  - En el evento load tenemos que registrar los eventos input:
    - tno.addEventListener("input", validacion, **false);**

API Forms

- Evento **invalid**:
  - Cuando un usuario envía un formulario, un evento invalid es disparado si un campo inválido es detectado.
  - El evento es disparado por el elemento que causa el error.

```
document.nombre_form.addEventListener("invalid", funcion_js, true);
function funcion_js(e){
    var elemento = e.target;
    elemento.style.background = "#FF0000";
}
```

API Forms

- Evento **invalid:**
  - Cuando queremos gestionar el envío o no del form.
  - No utilizamos un botón submit, si no button, le asociamos un evento y dentro de este podemos comprobar el estado de la validación:

```
if (document.nombreForm.checkValidity()){
    // Enviamos el form:
    document.nombreForm.submit();
}
```

API Forms

- Validación en tiempo Real:
  - Saber si un elemento es válido o no:

```
document.nombreForm.addEventListener("input", controlar, false);

function controlar(e){
    var elemento = e.target;
    if (elemento.validity.valid){
        // Cambiar algún estilo, es un error.
    } else {
        // Dejar el estilo como estaba.
    }
```

API Forms

- Este API también nos permite saber que es lo que ha fallado en la validación.

- Disponemos del objeto ValidateState que tiene 8 posibles valores según el error que se ha producido, por ejemplo:

```
if (elemento.validity.patternMismatch){ … }
```

API Forms

- Valores de **ValidateState**:
  - **valueMissing**: cuando el atributo required estaba activo y el campo está activo.
  - **typeMismatch**: No coincide el tipo introducido, por ejemplo en el email.
  - **patternMismatch**: El valor no se corresponde con el pattern establecido en el campo.
  - **tooLong**: Se declara maxLength y la entrada del campo es mas larga.
  - **rangeUnderflow**: cuando se declara el atributo min y el valor es más bajo.
  - **rangeOverflow**: se declara max y la entrada es más mayor.
  - **stepMismatch**: true cuando se declara el atributo step y su valor no coincide con min, max y step.
  - **customError**: Es true cuando hemos utilizado un error personalizado: setCustomValidity.

API Forms

- Estilos CSS3 para la validación:

```
input:required:invalid, input:focus:invalid {
    /* Estilo cuando es erróneo */
}

input:required:valid {
    /* Estilos cuando es válido */
}
```