

Parte 3

Artículos adicionales

Js

Ilya Kantor

Hecho el 19 de marzo de 2023

La última versión de este tutorial está en <https://es.javascript.info>.

Trabajamos constantemente para mejorar el tutorial. Si encuentra algún error, por favor escríbanos a [nuestro github](#).

- [Marcos y ventanas](#)
 - [Ventanas emergentes y métodos de ventana](#)
 - [Comunicación entre ventanas](#)
 - [El ataque de secuestro de clics](#)
- [Datos binarios y archivos](#)
 - [ArrayBuffer, arrays binarios](#)
 - [TextDecoder y TextEncoder](#)
 - [Blob](#)
 - [File y FileReader](#)
- [Solicitudes de red](#)
 - [Fetch](#)
 - [FormData](#)
 - [Fetch: Progreso de la descarga](#)
 - [Fetch: Abort](#)
 - [Fetch: Cross-Origin Requests](#)
 - [Fetch API](#)
 - [Objetos URL](#)
 - [XMLHttpRequest](#)
 - [Carga de archivos reanudable](#)
 - [Sondeo largo](#)
 - [WebSocket](#)
 - [Eventos enviados por el servidor](#)
- [Almacenando datos en el navegador](#)
 - [Cookies, document.cookie](#)
 - [LocalStorage, sessionStorage](#)
 - [IndexedDB](#)
- [Animaciones](#)
 - [Curva de Bézier](#)
 - [Animaciones CSS](#)
 - [Animaciones JavaScript](#)
- [Componentes Web](#)
 - [Desde la altura orbital](#)
 - [Elementos personalizados](#)
 - [Shadow DOM](#)
 - [Elemento template](#)
 - [Shadow DOM slots, composición](#)
 - [Estilo Shadow DOM](#)
 - [Shadow DOM y eventos](#)
- [Expresiones Regulares](#)
 - [Patrones y banderas \(flags\)](#)
 - [Clases de caracteres](#)
 - [Unicode: bandera "u" y clase \p{...}](#)
 - [Anclas: inicio ^ y final \\$ de cadena](#)
 - [Modo multilínea de anclas ^ \\$, bandera "m"](#)
 - [Límite de palabra: \b](#)
 - [Escapando, caracteres especiales](#)
 - [Conjuntos y rangos \[...\]](#)
 - [Cuantificadores +, *, ? y {n}](#)
 - [Cuantificadores codiciosos y perezosos](#)

- Grupos de captura
- Referencias inversas en patrones: \N y \k<nombre>
- Alternancia (O) |
- Lookahead y lookbehind (revisar delante/detrás)
- Backtracking catastrófico
- Indicador adhesivo “y”, buscando en una posición.
- Métodos de RegExp y String

Marcos y ventanas

Ventanas emergentes y métodos de ventana

Una ventana emergente (popup window) es uno de los métodos más antiguos para mostrar documentos adicionales al usuario.

Básicamente, solo ejecutas :

```
window.open("https://javascript.info/");
```

...Y eso abrirá una nueva ventana con la URL dada. La mayoría de los navegadores modernos están configurados para abrir pestañas nuevas en vez de ventanas separadas.

Los popups existen desde tiempos realmente antiguos. La idea inicial fue mostrar otro contenido sin cerrar la ventana principal. Ahora hay otras formas de hacerlo: podemos cargar contenido dinámicamente con `fetch` y mostrarlo de forma dinámica con `<div>`. Entonces, los popups no son algo que usamos todos los días.

Además, los popups son complicados en dispositivos móviles, que no muestran varias ventanas simultáneamente.

Aún así, hay tareas donde los popups todavía son usados, por ejemplo para autorización o autenticación (Ingreso con Google/Facebook/...), porque:

1. Un popup es una ventana separada con su propio entorno JavaScript independiente. Por lo tanto es seguro abrir un popup desde un sitio de terceros no confiable.
2. Es muy fácil abrir un popup.
3. Un popup puede navegar (cambiar URL) y enviar mensajes a la ventana que lo abrió.

Bloqueo de ventanas emergentes (Popup)

En el pasado, sitios malvados abusaron mucho de las ventanas emergentes. Una página incorrecta podría abrir toneladas de ventanas emergentes con anuncios. Entonces, la mayoría de los navegadores intentan bloquear las ventanas emergentes y proteger al usuario.

La mayoría de los navegadores bloquean las ventanas emergentes si se llaman fuera de los controladores de eventos activados por el usuario, como `onclick`.

Por ejemplo:

```
// popup blocked
window.open("https://javascript.info");

// popup allowed
button.onclick = () => {
  window.open("https://javascript.info");
};
```

De esta manera, los usuarios están algo protegidos de ventanas emergentes no deseadas, pero la funcionalidad no está totalmente deshabilitada.

`window.open`

La sintaxis para abrir una ventana emergente es: `window.open(url, name, params)`:

`url`

Una URL para cargar en la nueva ventana.

`name`

Un nombre de la nueva ventana. Cada ventana tiene un `window.name`, y aquí podemos especificar cuál ventana usar para la ventana emergente. Si hay una ventana con ese nombre, la URL dada se abre en ella, de lo contrario abre una nueva ventana.

`params`

La cadena de configuración para nueva ventana. Contiene configuraciones, delimitado por una coma. No debe haber espacios en los parámetros, por ejemplo: `width=200, height=100`.

Configuración de `params`:

- Posición:
 - `left/top` (numérico) – coordenadas de la esquina superior izquierda de la ventana en la pantalla. Hay una limitación: no se puede colocar una nueva ventana fuera de la pantalla.
 - `width/height` (numérico) – ancho y alto de una nueva ventana. Hay un límite mínimo de ancho/alto , así que es imposible crear una ventana invisible.
- Características de la ventana:
 - `menubar` (yes/no) – muestra u oculta el menú del navegador en la nueva ventana.
 - `toolbar` (yes/no) – muestra u oculta la barra de navegación del navegador (atrás, adelante, recargar, etc.) en la nueva ventana.
 - `location` (yes/no) – muestra u oculta el campo URL en la nueva ventana. FF e IE no permiten ocultarlo por defecto.
 - `status` (yes/no) – muestra u oculta la barra de estado. De nuevo, la mayoría de los navegadores lo obligan a mostrar.
 - `resizable` (yes/no) – permite deshabilitar el cambio de tamaño para la nueva ventana. No recomendado.
 - `scrollbars` (yes/no) – permite deshabilitar las barras de desplazamiento para la nueva ventana. No recomendado.

También hay una serie de características específicas del navegador menos compatibles, que generalmente no se usan. Revisa [window.open en MDN](#) para ejemplos.

Ejemplo: Una ventana minimalista

Abramos una ventana con un conjunto mínimo de características solo para ver cuál de ellos permite desactivar el navegador:

```
let params = `scrollbars=no,resizable=no,status=no,location=no,toolbar=no,menubar=no,
width=0,height=0,left=-1000,top=-1000`;

open("/", "test", params);
```

Aquí la mayoría de las “características de la ventana” están deshabilitadas y la ventana se coloca fuera de la pantalla. Ejecútelo y vea lo que realmente sucede. La mayoría de los navegadores “arreglan” cosas extrañas como cero `ancho/alto` y fuera de pantalla `Izquierda/superior`. Por ejemplo, Chrome abre una ventana con ancho/alto completo, para que ocupe la pantalla completa.

Agreguemos opciones de posicionamiento normal y coordenadas razonables de `ancho`, `altura`, `izquierda`, `arriba`:

```
let params = `scrollbars=no,resizable=no,status=no,location=no,toolbar=no,menubar=no,
width=600,height=300,left=100,top=100`;

open("/", "test", params);
```

La mayoría de los navegadores muestran el ejemplo anterior según sea necesario.

Reglas para configuraciones omitidas:

- Si no hay un tercer argumento en la llamada a `open` o está vacío, se usan los parámetros de ventana predeterminados.
- Si hay una cadena de `params`, pero se omiten algunas características sí/no (`yes/no`), las características omitidas se asumen con valor `no` . Entonces, si especifica parámetros, asegúrese de establecer explícitamente todas las funciones requeridas en `yes` .
- Si no hay `izquierda/arriba` en `params`, entonces el navegador intenta abrir una nueva ventana cerca de la última ventana abierta.
- Si no hay `ancho/altura` , entonces la nueva ventana tendrá el mismo tamaño que la última abierta.

Acceder a la ventana emergente desde la ventana

La llamada `open` devuelve una referencia a la nueva ventana. Se puede usar para manipular sus propiedades, cambiar de ubicación y aún más.

En este ejemplo, generamos contenido emergente a partir de JavaScript:

```
let newWin = window.open("about:blank", "hello", "width=200,height=200");
newWin.document.write("Hello, world!");
```

Y aquí modificamos el contenido después de la carga:

```
let newWindow = open("/", "example", "width=300,height=300");
newWindow.focus();

alert(newWindow.location.href); // (*) about:blank, loading hasn't started yet

newWindow.onload = function() {
  let html = `<div style="font-size:30px">Welcome!</div>`;
  newWindow.document.body.insertAdjacentHTML("afterbegin", html);
};
```

Por favor, tenga en cuenta: inmediatamente después de `window.open` la nueva ventana no está cargada aún. Esto queda demostrado por el `alert` en la línea (*). Así que esperamos a que `onload` lo modifique. También podríamos usar `DOMContentLoaded` de los manejadores de `newWin.document`.

Política mismo origen

Las ventanas pueden acceder libremente a los contenidos de las demás sólo si provienen del mismo origen (el mismo protocolo://domain:port).

De lo contrario es imposible por razones de seguridad del usuario, por ejemplo si la ventana principal es de `site.com` y la ventana emergente (popup) es de `gmail.com`. Para los detalles, ver capítulo [Comunicación entre ventanas](#).

Acceder a la ventana desde el popup

Un popup también puede acceder la ventana que lo abrió usando la referencia `window.opener`. Es `null` para todas las ventanas excepto los popups.

Si ejecutas el código de abajo, reemplaza el contenido de la ventana del opener (actual) con “Test”:

```
let newWin = window.open("about:blank", "hello", "width=200,height=200");

newWin.document.write(
  "<script>window.opener.document.body.innerHTML = 'Test'</script>"
);
```

Así que la conexión entre las ventanas es bidireccional: la ventana principal y el popup tienen una referencia entre sí.

Cerrar una popup

Para cerrar una ventana: `win.close()`.

Para comprobar si una ventana esta cerrada: `win.closed`.

Técnicamente, el `close()` es un método disponible para cualquier `ventana`, pero `window.close()` es ignorado por la mayoría de los navegadores si `window` no es creada con `window.open()`. Así que solo funcionará en una popup.

El `closed` es una propiedad `true` si la ventana esta cerrada. Eso es usualmente para comprobar la popup (o la ventana principal) está todavía abierta o no. Un usuario puede cerrarla en cualquier momento, y nuestro código debería tener esa posibilidad en cuenta.

Este código se carga y luego cierra la ventana:

```
let newWindow = open("/", "example", "width=300,height=300");

newWindow.onload = function () {
  newWindow.close();
  alert(newWindow.closed); // true
};
```

desplazamiento y cambio de tamaño

Hay métodos para mover/redimensionar una ventana:

`win.moveTo(x, y)`

Mueve la ventana en relación con la posición actual `x` píxeles a la derecha y `y` píxeles hacia abajo. Valores negativos están permitidos(para mover a la izquierda/arriba).

`win.resizeBy(width, height)`

Cambiar el tamaño de la ventana según el `width/height` dado en relación con el tamaño actual. Se permiten valores negativos.

`win.resizeTo(width, height)`

Redimensionar la ventana al tamaño dado.

También existe el evento `window.onresize`.

Solo Popup

Para evitar abusos, el navegador suele bloquear estos métodos. Solo funcionan de manera confiable en las ventanas emergentes que abrimos, que no tienen pestañas adicionales.

No minification/maximization

JavaScript no tiene forma de minimizar o maximizar una ventana. Estas funciones de nivel de sistema operativo están ocultas para los desarrolladores de frontend.

Los métodos de movimiento/cambio de tamaño no funcionan para ventanas maximizadas/minimizadas.

desplazando una ventana

Ya hemos hablado sobre el desplazamiento de una ventana en el capítulo [Tamaño de ventana y desplazamiento](#).

`win.scrollBy(x, y)`

Desplaza la ventana `x` píxeles a la derecha y `y` hacia abajo en relación con el actual desplazamiento. Se permiten valores negativos.

`win.scrollTo(x, y)`

Desplaza la ventana a las coordenadas dadas `(x, y)`.

`elem.scrollIntoView(top = true)`

Desplaza la ventana para hacer que `elem` aparezca en la parte superior (la predeterminada) o en la parte inferior para `elem.scrollIntoView(false)`.

También existe el evento `window.onscroll`.

Enfocar/desenfocar una ventana

Teóricamente, están los métodos `window.focus()` y `window.blur()` para poner/sacar el foco de una ventana. Y los eventos `focus/blur` que permiten captar el momento en el que el visitante enfoca una ventana y en el que cambia a otro lugar.

En la práctica estos métodos están severamente limitados, porque en el pasado las páginas malignas abusaban de ellos.

Por ejemplo, mira este código:

```
window.onblur = () => window.focus();
```

Cuando un usuario intenta salir de la ventana (`window.onblur`), lo vuelve a enfocar. La intención es “bloquear” al usuario dentro de la `window`.

Entonces, hay limitaciones que prohíben el código así. Existen muchas limitaciones para proteger al usuario de anuncios y páginas malignas. Ellos dependen del navegador.

Por ejemplo, un navegador móvil generalmente ignora esa llamada por completo. Además, el enfoque no funciona cuando se abre una ventana emergente en una pestaña separada en lugar de en una nueva ventana.

Aún así hay algunas cosas que se pueden hacer.

Por ejemplo:

- Cuando abrimos una popup, puede ser una buena idea ejecutar un `newWindow.focus()` en ella. Solo por si acaso. Para algunas combinaciones de sistema-operativo/navegador, asegura que el usuario ahora esté en la nueva ventana.
- Si queremos saber cuándo un visitante realmente usa nuestra aplicación web, podemos monitorear `window.onfocus/onblur`. Esto nos permite suspender/reanudar las actividades en la página, animaciones etc. Pero tenga en cuenta que el evento `blur` solamente significa que el visitante salió de la ventana. La ventana queda en segundo plano, pero aún puede ser visible.

Resumen

Las ventanas emergentes se utilizan con poca frecuencia, ya que existen alternativas: cargar y mostrar información en la página o en iframe.

Si vamos a abrir una ventana emergente, una buena práctica es informar al usuario al respecto. Un ícono de “ventana que se abre” cerca de un enlace o botón permitiría al visitante sobrevivir al cambio de enfoque y tener en cuenta ambas ventanas.

- Se puede abrir una ventana emergente con la llamada `open(url, name, params)`. Devuelve la referencia a la ventana recién abierta.
- Los navegadores bloquean las llamadas `open` desde el código fuera de las acciones del usuario. Por lo general aparece una notificación para que un usuario pueda permitirlos.
- Los navegadores abren una nueva pestaña de forma predeterminada, pero si se proporcionan tamaños, será una ventana emergente.
- La ventana emergente puede acceder a la ventana que la abre usando la propiedad `window.opener`.
- La ventana principal y la ventana emergente pueden leerse y modificarse libremente entre sí si tienen el mismo origen. De lo contrario, pueden cambiar de ubicación e [intercambiar mensajes](#).

Para cerrar la ventana emergente: use `close()`. Además, el usuario puede cerrarlas (como cualquier otra ventana). El `window.closed` es `true` después de eso.

- Los métodos `focus()` y `blur()` permiten enfocar/desenfocar una ventana. Pero no funcionan todo el tiempo.
- Los eventos `focus` y `blur` permiten rastrear el cambio dentro y fuera de la ventana. Pero tenga en cuenta que una ventana puede seguir siendo visible incluso en el estado de fondo, después de “desenfoque”.

Comunicación entre ventanas

La política de “Mismo origen” (mismo sitio) limita el acceso de ventanas y marcos entre sí.

La idea es que si un usuario tiene dos páginas abiertas: una de `john-smith.com`, y otra es `gmail.com`, entonces no querrán que un script de `john-smith.com` lea nuestro correo de `gmail.com`. Por lo tanto, el propósito de la política de “Mismo origen” es proteger a los usuarios del robo de información.

Mismo origen

Se dice que dos URL tienen el “mismo origen” si tienen el mismo protocolo, dominio y puerto.

Todas estas URL comparten el mismo origen:

- `http://site.com`
- `http://site.com/`
- `http://site.com/my/page.html`

Estas no:

- `http://www.site.com` (otro dominio: `www.` importa)

- `http://site.org` (otro dominio: `.org` importa)
- `https://site.com` (otro protocolo: `https`)
- `http://site.com:8080` (otro puerto: 8080)

La política "Mismo Origen" establece que:

- si tenemos una referencia a otra ventana, por ejemplo, una ventana emergente creada por `window.open` o una ventana dentro de `<iframe>`, y esa ventana viene del mismo origen, entonces tenemos acceso completo a esa ventana.
- en caso contrario, si viene de otro origen, entonces no podemos acceder al contenido de esa ventana: variables, documento, nada. La única excepción es `location`: podemos cambiarla (redirigiendo así al usuario). Pero no podemos leer `location` (por lo que no podemos ver dónde está el usuario ahora, no hay fuga de información).

En acción: iframe

Una etiqueta `<iframe>` aloja una ventana incrustada por separado, con sus propios objetos `document` y `window` separados.

Podemos acceder a ellos usando propiedades:

- `iframe.contentWindow` para obtener la ventana dentro del `<iframe>`.
- `iframe.contentDocument` para obtener el documento dentro del `<iframe>`, abreviatura de `iframe.contentWindow.document`.

Cuando accedemos a algo dentro de la ventana incrustada, el navegador comprueba si el iframe tiene el mismo origen. Si no es así, se niega el acceso (escribir en `location` es una excepción, aún está permitido).

Por ejemplo, intentemos leer y escribir en `<iframe>` desde otro origen:

```
<iframe src="https://example.com" id="iframe"></iframe>

<script>
  iframe.onload = function() {
    // podemos obtener la referencia a la ventana interior
    let iframeWindow = iframe.contentWindow; // OK
    try {
      // ...pero no al documento que contiene
      let doc = iframe.contentDocument; // ERROR
    } catch(e) {
      alert(e); // Error de seguridad (otro origen)
    }

    // tampoco podemos LEER la URL de la página en iframe
    try {
      // No se puede leer la URL del objeto Location
      let href = iframe.contentWindow.location.href; // ERROR
    } catch(e) {
      alert(e); // Error de seguridad
    }

    // ...¡podemos ESCRIBIR en location (y así cargar algo más en el iframe)!
    iframe.contentWindow.location = '/'; // OK

    iframe.onload = null; // borra el controlador para no ejecutarlo después del cambio de ubicación
  };
</script>
```

El código anterior muestra errores para cualquier operación excepto:

- Obtener la referencia a la ventana interna `iframe.contentWindow` – eso está permitido.
- Escribir a `location`.

Por el contrario, si el `<iframe>` tiene el mismo origen, podemos hacer cualquier cosa con él:

```
<!-- iframe from the same site -->
<iframe src="/" id="iframe"></iframe>

<script>
  iframe.onload = function() {
    // solo haz cualquier cosa
    iframe.contentDocument.body.prepend("¡Hola, mundo!");
  };
</script>
```

```
};  
</script>
```

`iframe.onload` vs `iframe.contentWindow.onload`

El evento `iframe.onload` (en la etiqueta `<iframe>`) es esencialmente el mismo que `iframe.contentWindow.onload` (en el objeto de ventana incrustada). Se activa cuando la ventana incrustada se carga completamente con todos los recursos.

... Pero no podemos acceder a `iframe.contentWindow.onload` para un iframe de otro origen, así que usamos `iframe.onload`.

Ventanas en subdominios: `document.domain`

Por definición, dos URL con diferentes dominios tienen diferentes orígenes.

Pero si las ventanas comparten el mismo dominio de segundo nivel, por ejemplo, `john.site.com`, `peter.site.com` y `site.com` (de modo que su dominio de segundo nivel común es `site.com`), podemos hacer que el navegador ignore esa diferencia, de modo que puedan tratarse como si vinieran del “mismo origen” para efecto de la comunicación entre ventanas.

Para que funcione, cada una de estas ventanas debe ejecutar el código:

```
document.domain = 'site.com';
```

Eso es todo. Ahora pueden interactuar sin limitaciones. Nuevamente, eso solo es posible para páginas con el mismo dominio de segundo nivel.

Obsoleto, pero aún funcionando

La propiedad `document.domain` está en proceso de ser removido de la [especificación ↗](#). Los mensajería cross-window (explicado pronto más abajo) es el reemplazo sugerido.

Dicho esto, hasta ahora todos los navegadores lo soportan. Y tal soporte será mantenido en el futuro, para no romper el código existente que se apoya en `document.domain`.

Iframe: trampa del documento incorrecto

Cuando un iframe proviene del mismo origen y podemos acceder a su `document`, existe una trampa. No está relacionado con cross-origin, pero es importante saberlo.

Tras su creación, un iframe tiene inmediatamente un documento. ¡Pero ese documento es diferente del que se carga en él! Entonces, si hacemos algo con el documento de inmediato, probablemente se perderá.

Aquí, mira:

```
<iframe src="/" id="iframe"></iframe>  
  
<script>  
let oldDoc = iframe.contentDocument;  
iframe.onload = function() {  
  let newDoc = iframe.contentDocument;  
  // ¡el documento cargado no es el mismo que el inicial!  
  alert(oldDoc == newDoc); // false  
};  
</script>
```

No deberíamos trabajar con el documento de un iframe aún no cargado, porque ese es el *documento incorrecto*. Si configuramos algún controlador de eventos en él, se ignorarán.

¿Cómo detectar el momento en que el documento está ahí?

El documento correcto definitivamente está en su lugar cuando se activa `iframe.onload`. Pero solo se activa cuando se carga todo el iframe con todos los recursos.

Podemos intentar capturar el momento anterior usando comprobaciones en `setInterval`:

```
<iframe src="/" id="iframe"></iframe>

<script>
  let oldDoc = iframe.contentDocument;

  // cada 100 ms comprueba si el documento es el nuevo
  let timer = setInterval(() => {
    let newDoc = iframe.contentDocument;
    if (newDoc == oldDoc) return;

    alert("¡El nuevo documento está aquí!");

    clearInterval(timer); // cancelo setInterval, ya no lo necesito
  }, 100);
</script>
```

Colección: `window.frames`

Una forma alternativa de obtener un objeto de ventana para `<iframe>` – es obtenerlo de la colección nombrada `window.frames`:

- Por número: `window.frames[0]` – el objeto de ventana para el primer marco del documento.
- Por nombre: `window.frames iframeName` – el objeto de ventana para el marco con `name="iframeName"`.

Por ejemplo:

```
<iframe src="/" style="height:80px" name="win" id="iframe"></iframe>

<script>
  alert(iframe.contentWindow == frames[0]); // true
  alert(iframe.contentWindow == frames.win); // true
</script>
```

Un `iframe` puede tener otros `iframes` en su interior. Los objetos `window` correspondientes forman una jerarquía.

Los enlaces de navegación son:

- `window.frames` – la colección de ventanas “hijas” (para marcos anidados).
- `window.parent` – la referencia a la ventana “padre” (exterior).
- `window.top` – la referencia a la ventana padre superior.

Por ejemplo:

```
window.frames[0].parent === window; // true
```

Podemos usar la propiedad `top` para verificar si el documento actual está abierto dentro de un marco o no:

```
if (window == top) { // current window == window.top?
  alert('El script está en la ventana superior, no en un marco.');
} else {
  alert('¡El script se ejecuta en un marco!');
}
```

El atributo “sandbox” de `iframe`

El atributo `sandbox` permite la exclusión de ciertas acciones dentro de un `<iframe>` para evitar que ejecute código no confiable. Separa el `iframe` en un “sandbox” tratándolo como si procediera de otro origen y/o aplicando otras limitaciones.

Hay un “conjunto predeterminado” de restricciones aplicadas para `<iframe sandbox src="...>`. Pero se puede relajar si proporcionamos una lista de restricciones separadas por espacios que no deben aplicarse como un valor del atributo, así: `<iframe sandbox="allow-forms allow-popups">`.

En otras palabras, un atributo “sandbox” vacío pone las limitaciones más estrictas posibles, pero podemos poner una lista delimitada por espacios de aquellas que queremos levantar.

Aquí hay una lista de limitaciones:

allow-same-origin

Por defecto, “sandbox” fuerza la política de “origen diferente” para el iframe. En otras palabras, hace que el navegador trate el `iframe` como si viniera de otro origen, incluso si su `src` apunta al mismo sitio. Con todas las restricciones implícitas para los scripts. Esta opción elimina esa característica.

allow-top-navigation

Permite que el `iframe` cambie `parent.location`.

allow-forms

Permite enviar formularios desde `iframe`.

allow-scripts

Permite ejecutar scripts desde el `iframe`.

allow-popups

Permite `window.open` popups desde el `iframe`

Consulta el manual [🔗](#) para obtener más información.

El siguiente ejemplo muestra un iframe dentro de un entorno controlado con el conjunto de restricciones predeterminado: `<iframe sandbox src="...>`. Tiene algo de JavaScript y un formulario.

Tenga en cuenta que nada funciona. Entonces, el conjunto predeterminado es realmente duro:

<https://plnkr.co/edit/Ph8oZDpoxUg24AGu?p=preview> [🔗](#)

Por favor tome nota:

El propósito del atributo “sandbox” es solo agregar más restricciones. No puede eliminarlas. En particular, no puede relajar las restricciones del mismo origen si el iframe proviene de otro origen.

Mensajería entre ventanas

La interfaz `postMessage` permite que las ventanas se comuniquen entre sí sin importar de qué origen sean.

Por lo tanto, es una forma de evitar la política del “mismo origen”. Permite a una ventana de `john-smith.com` hablar con `gmail.com` e intercambiar información, pero solo si ambos están de acuerdo y llaman a las funciones de JavaScript correspondientes. Eso lo hace seguro para los usuarios.

La interfaz tiene dos partes.

`postMessage`

La ventana que quiere enviar un mensaje llama al método [postMessage](#) de la ventana receptora. En otras palabras, si queremos enviar el mensaje a `win`, debemos llamar a `win.postMessage(data, targetOrigin)`.

Argumentos:

`data`

Los datos a enviar. Puede ser cualquier objeto, los datos se clonian mediante el “algoritmo de clonación estructurada”. IE solo admite strings, por lo que debemos usar `JSON.stringify` en objetos complejos para admitir ese navegador.

`targetOrigin`

Especifica el origen de la ventana de destino, de modo que solo una ventana del origen dado recibirá el mensaje.

El argumento “targetOrigin” es una medida de seguridad. Recuerde que si la ventana de destino proviene de otro origen, no podemos leer su `location` en la ventana del remitente. Por lo tanto, no podemos estar seguros qué sitio está abierto en la ventana deseada en este momento: el usuario podría navegar fuera del sitio y la ventana del remitente no tener idea de ello.

Especificando `targetOrigin` asegura que la ventana solo reciba los datos si todavía está en el sitio correcto. Importante cuando los datos son sensibles.

Por ejemplo, aquí `win` solo recibirá el mensaje si tiene un documento del origen `http://example.com`:

```
<iframe src="http://example.com" name="example">

<script>
  let win = window.frames.example;

  win.postMessage("message", "http://example.com");
</script>
```

Si no queremos esa comprobación, podemos establecer `targetOrigin` en `*`.

```
<iframe src="http://example.com" name="example">

<script>
  let win = window.frames.example;

  win.postMessage("message", "*");
</script>
```

onmessage

Para recibir un mensaje, la ventana destino debe tener un controlador en el evento `message`. Se activa cuando se llama a `postMessage` (y la comprobación de `targetOrigin` es correcta).

El objeto de evento tiene propiedades especiales:

data

Los datos de `postMessage`.

origin

El origen del remitente, por ejemplo, `http://javascript.info`.

source

La referencia a la ventana del remitente. Podemos llamar inmediatamente `source.postMessage(...)` de regreso si queremos.

Para asignar ese controlador, debemos usar `addEventListener`, una sintaxis corta `window.onmessage` no funciona.

He aquí un ejemplo:

```
window.addEventListener("message", function(event) {
  if (event.origin != 'http://javascript.info') {
    // algo de un dominio desconocido, ignorémoslo
    return;
  }

  alert( "Recibi: " + event.data );

  // puedes enviar un mensaje usando event.source.postMessage(...)
});
```

El ejemplo completo:

[https://plnkr.co/edit/eDoxBvzrEN3SjcaB?p=preview ↗](https://plnkr.co/edit/eDoxBvzrEN3SjcaB?p=preview)

Resumen

Para llamar a métodos y acceder al contenido de otra ventana, primero debemos tener una referencia a ella.

Para las ventanas emergentes tenemos estas referencias:

- Desde la ventana de apertura: `window.open` – abre una nueva ventana y devuelve una referencia a ella,
- Desde la ventana emergente: `window.opener` – es una referencia a la ventana de apertura desde una ventana emergente.

Para iframes, podemos acceder a las ventanas padres o hijas usando:

- `window.frames` – una colección de objetos de ventana anidados,
- `window.parent`, `window.top` son las referencias a las ventanas principales y superiores,
- `iframe.contentWindow` es la ventana dentro de una etiqueta `<iframe>`.

Si las ventanas comparten el mismo origen (host, puerto, protocolo), las ventanas pueden hacer lo que quieran entre sí.

En caso contrario, las únicas acciones posibles son:

- Cambiar `location` en otra ventana (acceso de solo escritura).
- Enviarle un mensaje.

Las excepciones son:

- Ventanas que comparten el mismo dominio de segundo nivel: `a.site.com` y `b.site.com`. Luego, configurar `document.domain='site.com'` en ambos, los coloca en el estado de "mismo origen".
- Si un iframe tiene un atributo `sandbox`, se coloca forzosamente en el estado de "origen diferente", a menos que se especifique `allow-same-origin` en el valor del atributo. Eso se puede usar para ejecutar código que no es de confianza en iframes desde el mismo sitio.

La interfaz `postMessage` permite que dos ventanas con cualquier origen hablen:

1. El remitente llama a `targetWin.postMessage(data, targetOrigin)`.
2. Si `targetOrigin` no es `'*'`, entonces el navegador comprueba si la ventana targetWin tiene el origen targetOrigin.`
3. Si es así, entonces `targetWin` activa el evento `message` con propiedades especiales:
 - `origin` – el origen de la ventana del remitente (como `http://my.site.com`)
 - `source` – la referencia a la ventana del remitente.
 - `data` – los datos, cualquier objeto en todas partes excepto IE que solo admite cadenas.

Deberíamos usar `addEventListener` para configurar el controlador para este evento dentro de la ventana de destino.

El ataque de secuestro de clics

El ataque "secuestro de clics" permite que una página maligna haga clic en un "sitio víctima" * en nombre del visitante *.

Muchos sitios fueron pirateados de esta manera, incluidos Twitter, Facebook, Paypal y otros sitios. Todos han sido arreglados, por supuesto.

La idea

La idea es muy simple.

Así es como se hizo el secuestro de clics con Facebook:

1. Un visitante es atraído a la página maligna. No importa cómo.
2. La página tiene un enlace de apariencia inofensiva (como "hazte rico ahora" o "haz clic aquí, muy divertido").
3. Sobre ese enlace, la página maligna coloca un `<iframe>` transparente con `src` de facebook.com, de tal manera que el botón "Me gusta" está justo encima de ese enlace. Por lo general, eso se hace con `z-index`.
4. Al intentar hacer clic en el enlace, el visitante de hecho hace clic en el botón.

La demostración

Así es como se ve la página malvada. Para aclarar las cosas, el `<iframe>` es semitransparente (en las páginas realmente malvadas es completamente transparente):

```
<style>
iframe { /* iframe del sitio de la víctima */
  width: 400px;
  height: 100px;
  position: absolute;
  top:0; left:-20px;
  opacity: 0.5; /* realmente opacity:0 */
```

```

    z-index: 1;
}
</style>

<div>Haga clic para hacerse rico ahora:</div>

<!-- La URL del sitio de la víctima -->
<iframe src="/clickjacking/facebook.html"></iframe>

<button>¡Haga clic aquí!</button>

<div>...Y eres genial (en realidad soy un pirata informático genial)!</div>

```

La demostración completa del ataque:

[https://plnkr.co/edit/b7ebSHKUARbAlqdo?p=preview ↗](https://plnkr.co/edit/b7ebSHKUARbAlqdo?p=preview)

Aquí tenemos un `<iframe src="facebook.html">` semitransparente, y en el ejemplo podemos verlo flotando sobre el botón. Un clic en el botón realmente hace clic en el iframe, pero eso no es visible para el usuario, porque el iframe es transparente.

Como resultado, si el visitante está autorizado en Facebook (“recordarme” generalmente está activado), entonces agrega un “Me gusta”. En Twitter sería un botón “Seguir”.

Este es el mismo ejemplo, pero más cercano a la realidad, con `opacity:0` para `<iframe>`:

[https://plnkr.co/edit/UtEq0WPu7HKCx7Fj?p=preview ↗](https://plnkr.co/edit/UtEq0WPu7HKCx7Fj?p=preview)

Todo lo que necesitamos para atacar es colocar el `<iframe>` en la página maligna de tal manera que el botón esté justo sobre el enlace. De modo que cuando un usuario hace clic en el enlace, en realidad hace clic en el botón. Eso suele ser posible con CSS.

Clickjacking es para clics, no para teclado

El ataque solo afecta las acciones del mouse (o similares, como los toques en el móvil).

La entrada del teclado es muy difícil de redirigir. Técnicamente, si tenemos un campo de texto para piratear, entonces podemos colocar un iframe de tal manera que los campos de texto se superpongan entre sí. Entonces, cuando un visitante intenta concentrarse en la entrada que ve en la página, en realidad se enfoca en la entrada dentro del iframe.

Pero luego hay un problema. Todo lo que escriba el visitante estará oculto, porque el iframe no es visible.

Las personas generalmente dejarán de escribir cuando no puedan ver sus nuevos caracteres impresos en la pantalla.

Defensas de la vieja escuela (débiles)

La defensa más antigua es un poco de JavaScript que prohíbe abrir la página en un marco (el llamado “framebusting”).

Eso se ve así:

```

if (top != window) {
  top.location = window.location;
}

```

Es decir: si la ventana descubre que no está en la parte superior, automáticamente se convierte en la parte superior.

Esta no es una defensa confiable, porque hay muchas formas de esquivarla. Cubramos algunas.

Bloquear la navegación superior

Podemos bloquear la transición causada por cambiar `top.location` en el controlador de eventos `beforeunload`.

La página superior (adjuntando una, que pertenece al pirata informático) establece un controlador de prevención, como este:

```

window.onbeforeunload = function() {
  return false;
};

```

Cuando el `iframe` intenta cambiar `top.location`, el visitante recibe un mensaje preguntándole si quiere irse.

En la mayoría de los casos, el visitante respondería negativamente porque no conocen el iframe; todo lo que pueden ver es la página superior, no hay razón para irse. ¡Así que `top.location` no cambiará!

En acción:

<https://plnkr.co/edit/0CHIKO3ehOKtgrLM?p=preview>

Atributo Sandbox

Una de las cosas restringidas por el atributo `sandbox` es la navegación. Un iframe de espacio aislado no puede cambiar `top.location`.

Entonces podemos agregar el iframe con `sandbox="allow-scripts allow-forms"`. Eso relajaría las restricciones, permitiendo guiones y formularios. Pero omitimos `allow-top-navigation` para que se prohíba cambiar `top.location`.

Aquí está el código:

```
<iframe sandbox "allow-scripts allow-forms" src="facebook.html"></iframe>
```

También hay otras formas de evitar esa simple protección.

X-Frame-Options

El encabezado del lado del servidor `X-Frame-Options` puede permitir o prohibir mostrar la página dentro de un marco.

Debe enviarse exactamente como encabezado HTTP: el navegador lo ignorará si se encuentra en la etiqueta HTML `<meta>`. Entonces, `<meta http-equiv="X-Frame-Options" ...>` no hará nada.

El encabezado puede tener 3 valores:

DENY

Nunca muestra la página dentro de un marco.

SAMEORIGIN

Permitir dentro de un marco si el documento principal proviene del mismo origen.

ALLOW-FROM domain

Permitir dentro de un marco si el documento principal es del dominio dado.

Por ejemplo, Twitter usa `X-Frame-Options: SAMEORIGIN`.

Mostrando con funcionalidad deshabilitada

El encabezado `X-Frame-Options` tiene un efecto secundario. Otros sitios no podrán mostrar nuestra página en un marco, incluso si tienen buenas razones para hacerlo.

Así que hay otras soluciones... Por ejemplo, podemos "cubrir" la página con un `<div>` con estilos `height: 100%; width: 100%`, de modo que interceptará todos los clics. Ese `<div>` debe eliminarse si `window == top` o si descubrimos que no necesitamos la protección.

Algo como esto:

```
<style>
  #protector {
    height: 100%;
    width: 100%;
    position: absolute;
    left: 0;
    top: 0;
    z-index: 99999999;
  }
</style>

<div id="protector">
  <a href="/" target="_blank">Ir al sitio</a>
</div>

<script>
```

```
// habrá un error si la ventana superior es de un origen diferente
// pero está bien aquí
if (top.document.domain == document.domain) {
    protector.remove();
}
</script>
```

La demostración:

<https://plnkr.co/edit/8oe0eRfrOuCPeBr3?p=preview>

Atributo Samesite cookie

El atributo `samesite` cookie también puede prevenir ataques de secuestro de clics.

Una cookie con dicho atributo solo se envía a un sitio web si se abre directamente, no a través de un marco o de otra manera. Más información en el capítulo [Cookies, document.cookie](#).

Si el sitio, como Facebook, tenía el atributo `samesite` en su cookie de autenticación, así:

```
Set-Cookie: authorization=secret; samesite
```

...Entonces dicha cookie no se enviaría cuando Facebook esté abierto en iframe desde otro sitio. Entonces el ataque fracasaría.

El atributo `samesite` cookie no tendrá efecto cuando no se utilicen cookies. Esto puede permitir que otros sitios web muestren fácilmente nuestras páginas públicas no autenticadas en iframes.

Sin embargo, esto también puede permitir que los ataques de secuestro de clics funcionen en algunos casos limitados. Un sitio web de sondeo anónimo que evita la duplicación de votaciones al verificar las direcciones IP, por ejemplo, aún sería vulnerable al secuestro de clics porque no autentica a los usuarios que usan cookies.

Resumen

El secuestro de clics es una forma de “engañosar” a los usuarios para que hagan clic en el sitio de una víctima sin saber qué está sucediendo. Eso es peligroso si hay acciones importantes activadas por clic.

Un pirata informático puede publicar un enlace a su página maligna en un mensaje o atraer visitantes a su página por otros medios. Hay muchas variaciones.

Desde una perspectiva, el ataque “no es profundo”: todo lo que hace un pirata informático es interceptar un solo clic. Pero desde otra perspectiva, si el pirata informático sabe que después del clic aparecerá otro control, entonces pueden usar mensajes astutos para obligar al usuario a hacer clic en ellos también.

El ataque es bastante peligroso, porque cuando diseñamos la interfaz de usuario generalmente no anticipamos que un pirata informático pueda hacer clic en nombre del visitante. Entonces, las vulnerabilidades se pueden encontrar en lugares totalmente inesperados.

- Se recomienda utilizar `X-Frame-Options: SAMEORIGIN` en páginas (o sitios web completos) que no están destinados a verse dentro de marcos.
- Usa una cubierta `<div>` si queremos permitir que nuestras páginas se muestren en iframes, pero aún así permanecer seguras.

Datos binarios y archivos

Trabajando con datos binarios y archivos en JavaScript.

ArrayBuffer, arrays binarios

En el desarrollo web nos encontramos con datos binarios sobre todo al tratar con archivos (crear, cargar, descargar). Otro caso de uso típico es el procesamiento de imágenes.

Todo esto es posible en JavaScript y las operaciones binarias son de alto rendimiento.

Aunque hay un poco de confusión porque hay muchas clases. Por nombrar algunas:

- `ArrayBuffer`, `Uint8Array`, `DataView`, `Blob`, `File`, etc.

Los datos binarios en JavaScript se implementan de una manera no estándar en comparación con otros lenguajes. Pero cuando ordenamos las cosas, todo se vuelve bastante sencillo.

El objeto binario básico es `ArrayBuffer` – una referencia a un área de memoria contigua de longitud fija.

Lo creamos así:

```
let buffer = new ArrayBuffer(16); // crea un buffer de longitud 16
alert(buffer.byteLength); // 16
```

Esto asigna un área de memoria contigua de 16 bytes y la rellena previamente con ceros.

`ArrayBuffer` no es un array de algo

Eliminemos una posible fuente de confusión. `ArrayBuffer` no tiene nada en común con `Array`:

- Tiene una longitud fija, no podemos aumentarla ni disminuirla.
- Ocupa exactamente ese espacio en la memoria.
- Para acceder a bytes individuales, se necesita otro objeto “vista”, no `buffer[índice]`.

`ArrayBuffer` es un área de memoria. ¿Qué se almacena en ella? No tiene ninguna pista. Sólo una secuencia cruda de bytes.

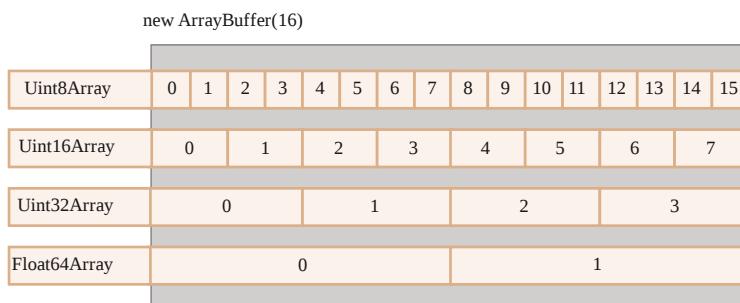
Para manipular un `ArrayBuffer`, necesitamos utilizar un objeto “vista”.

Un objeto vista no almacena nada por sí mismo. Son “gafas” que le dan una interpretación a los bytes almacenados en el `ArrayBuffer`.

Por ejemplo:

- `Uint8Array` : trata cada byte del `ArrayBuffer` como un número separado, con valores posibles de 0 a 255 (un byte es de 8 bits, por lo que sólo puede contener esa cantidad). Este valor se denomina “entero sin signo de 8 bits”.
- `Uint16Array` : trata cada 2 bytes como un entero, con valores posibles de 0 a 65535. Es lo que se llama un “entero sin signo de 16 bits”.
- `Uint32Array` : trata cada 4 bytes como un entero, con valores posibles de 0 a 4294967295. Eso se llama “entero sin signo de 32 bits”.
- `Float64Array` : trata cada 8 bytes como un número de punto flotante con valores posibles desde $5 \cdot 10^{-324}$ hasta $1.8 \cdot 10^{308}$.

Así, los datos binarios de un `ArrayBuffer` de 16 bytes pueden interpretarse como 16 “números diminutos”, u 8 números más grandes (2 bytes cada uno), o 4 aún más grandes (4 bytes cada uno), o 2 valores de punto flotante con alta precisión (8 bytes cada uno).



`ArrayBuffer` es el objeto central, la raíz de todo, los datos binarios en bruto.

Pero si vamos a escribir en él, o iterar sobre él (básicamente, para casi cualquier operación), debemos utilizar una vista. Por ejemplo:

```
let buffer = new ArrayBuffer(16); // crea un buffer de longitud 16
```

```

let view = new Uint32Array(buffer); // trata el buffer como una secuencia de enteros de 32 bits

alert(Uint32Array.BYTES_PER_ELEMENT); // 4 bytes por entero

alert(view.length); // 4, almacena esa cantidad de enteros
alert(view.byteLength); // 16, el tamaño en bytes

// escribamos un valor
view[0] = 123456;

// iteración sobre los valores
for(let num of view) {
  alert(num); // 123456, luego 0, 0, 0 (4 valores en total)
}

```

TypedArray

El término común para todas estas vistas (`Uint8Array`, `Uint32Array`, etc) es [TypedArray](#). Comparten el mismo conjunto de métodos y propiedades.

Por favor ten en cuenta que no hay ningún constructor llamado `TypedArray`, es sólo un término “paraguas” común para representar una de las vistas sobre `ArrayBuffer`: `Int8Array`, `Uint8Array` y así sucesivamente, la lista completa seguirá pronto.

Cuando veas algo como `new TypedArray`, significa cualquiera de `new Int8Array`, `new Uint8Array`, etc.

Las matrices tipificadas se comportan como las matrices normales: tienen índices y son iterables.

Un constructor de array tipado (ya sea `Int8Array` o `Float64Array`) se comporta de forma diferente dependiendo del tipo de argumento.

Hay 5 variantes de argumentos:

```

new TypedArray(buffer, [byteOffset], [length]);
new TypedArray(object);
new TypedArray(typedArray);
new TypedArray(length);
new TypedArray();

```

1. Si se suministra un argumento `ArrayBuffer`, la vista se crea sobre él. Ya usamos esa sintaxis.

Opcionalmente podemos proporcionar `byteOffset` para empezar (0 por defecto) y la longitud o `length` (hasta el final del buffer por defecto), entonces la vista cubrirá sólo una parte del `buffer`.

2. Si se da un `Array`, o cualquier objeto tipo array, se crea un array tipado de la misma longitud y se copia el contenido.

Podemos usarlo para pre-llenar el array con los datos:

```

let arr = new Uint8Array([0, 1, 2, 3]);
alert( arr.length ); // 4, creó una matriz binaria de la misma longitud
alert( arr[1] ); // 1, llenado con 4 bytes (enteros de 8 bits sin signo) con valores dados

```

3. Si se suministra otro `TypedArray` hace lo mismo: crea un array tipado de la misma longitud y copia los valores. Los valores se convierten al nuevo tipo en el proceso, si es necesario.

```

let arr16 = new Uint16Array([1, 1000]);
let arr8 = new Uint8Array(arr16);
alert( arr8[0] ); // 1
alert( arr8[1] ); // 232, trató de copiar 1000, pero no puede encajar 1000 en 8 bits (explicaciones a continuación)

```

4. Para un argumento numérico `length`: crea el array tipado para contener ese número de elementos. Su longitud en bytes será `length` multiplicada por el número de bytes de un solo elemento `TypedArray.BYTES_PER_ELEMENT`:

```

let arr = new Uint16Array(4); // crea un array tipado para 4 enteros
alert( Uint16Array.BYTES_PER_ELEMENT ); // 2 bytes por entero
alert( arr.byteLength ); // 8 (tamaño en bytes)

```

5. Sin argumentos crea un array tipado de longitud cero.

Podemos crear un `TypedArray` directamente sin mencionar `ArrayBuffer`. Pero una vista no puede existir sin un `ArrayBuffer` subyacente, por lo que se crea automáticamente en todos estos casos excepto en el primero (cuando se proporciona).

Para acceder al `ArrayBuffer` subyacente, en `TypedArray` existen las propiedades:

- `buffer` : hace referencia al `ArrayBuffer`.
- `byteLength` : la longitud del `ArrayBuffer`.

De esta forma siempre podemos pasar de una vista a otra:

```
let arr8 = new Uint8Array([0, 1, 2, 3]);  
  
// otra vista sobre los mismos datos  
let arr16 = new Uint16Array(arr8.buffer);
```

Esta es la lista de arrays tipados:

- `Uint8Array`, `Uint16Array`, `Uint32Array` : para números enteros de 8, 16 y 32 bits.
 - `Uint8ClampedArray` : para números enteros de 8 bits, los “sujeta” en la asignación (ver más abajo).
- `Int8Array`, `Int16Array`, `Int32Array` : para números enteros con signo (pueden ser negativos).
- `Float32Array`, `Float64Array` : para números de punto flotante con signo de 32 y 64 bits.

No existe `int8` o tipos de valor único similares

Ten en cuenta que a pesar de los nombres como `Int8Array`, no hay ningún tipo de valor único como `int` o `int8` en JavaScript.

Esto es lógico ya que `Int8Array` no es un array de estos valores individuales sino una vista sobre `ArrayBuffer`.

Comportamiento fuera de los límites

¿Qué pasa si intentamos escribir un valor fuera de límites en un array tipado? No habrá ningún error. Pero los bits extra se cortan.

Por ejemplo, intentemos poner 256 en `Uint8Array`. En forma binaria 256 es `100000000` (9 bits), pero `Uint8Array` sólo proporciona 8 bits por valor, lo que hace que el rango disponible sea de 0 a 255.

Para los números más grandes, sólo se almacenan los 8 bits más a la derecha (menos significativos), y el resto se corta:

8-bit integer
100000000 256

Así que obtendremos un cero.

Para el 257 la forma binaria es `100000001` (9 bits), los 8 más a la derecha se almacenan, por lo que tendremos `1` en el array:

8-bit integer
100000001 257

Es decir, se guarda el número módulo 2^8 .

Esta es la demo:

```
let uint8array = new Uint8Array(16);
```

```

let num = 256;
alert(num.toString(2)); // 100000000 (representación binaria)

uint8array[0] = 256;
uint8array[1] = 257;

alert(uint8array[0]); // 0
alert(uint8array[1]); // 1

```

`Uint8ClampedArray` es especial en este aspecto y su comportamiento es diferente. Guarda 255 para cualquier número que sea mayor que 255, y 0 para cualquier número negativo. Este comportamiento es útil para el procesamiento de imágenes.

Métodos TypedArray

`TypedArray` tiene los métodos regulares de `Array`, con notables excepciones.

Podemos iterar, `map`, `slice`, `find`, `reduce` etc.

Sin embargo, hay algunas cosas que no podemos hacer:

- No hay `splice`: no podemos “borrar” un valor, porque los arrays tipados son vistas en un buffer y estos son áreas fijas y contiguas de memoria. Todo lo que podemos hacer es asignar un cero.
- No hay método `concat`.

Hay dos métodos adicionales:

- `arr.set(fromArr, [offset])` copia todos los elementos de `fromArr` al `arr`, empezando en la posición `offset` (0 por defecto).
- `arr.subarray([begin, end])` crea una nueva vista del mismo tipo desde `begin` hasta `end` (excluyéndolo). Es similar al método `slice` (que también está soportado), pero no copia nada, sólo crea una nueva vista para operar sobre el trozo de datos dado.

Estos métodos nos permiten copiar arrays tipados, mezclarlos, crear nuevos arrays a partir de los existentes, etc.

DataView

`DataView` ↗ es una vista especial superflexible “no tipada” sobre `ArrayBuffer`. Permite acceder a los datos en cualquier desplazamiento en cualquier formato.

- En el caso de los arrays tipados, el constructor dicta cuál es el formato. Se supone que todo el array es uniforme. El número `i` es `arr[i]`.
- Con `DataView` accedemos a los datos con métodos como `.getUint8(i)` o `.getUint16(i)`. Elegimos el formato en el momento de la llamada al método en lugar de en el momento de la construcción.

La sintaxis:

```
new DataView(buffer, [byteOffset], [byteLength])
```

- `buffer`: el `ArrayBuffer` subyacente. A diferencia de los arrays tipados, `DataView` no crea un buffer por sí mismo. Necesitamos tenerlo preparado.
- `byteOffset`: la posición inicial en bytes de la vista (por defecto 0).
- `byteLength`: la longitud en bytes de la vista (por defecto hasta el final del `buffer`).

Por ejemplo, aquí extraemos números en diferentes formatos del mismo buffer:

```

// arreglo binario de 4 bytes, todos tienen el valor máximo 255
let buffer = new Uint8Array([255, 255, 255, 255]).buffer;

let dataView = new DataView(buffer);

// get 8-bit number at offset 0
alert( dataView.getUint8(0) ); // 255

// ahora obtenemos un número de 16 bits en el offset 0, que consta de 2 bytes, que juntos se interpretan como 65535

```

```

alert( dataView.getUint16(0) ); // 65535 (mayor entero sin signo de 16 bits)

// obtener un número de 32 bits en el offset 0
alert( dataView.getInt32(0) ); // 4294967295 (mayor entero de 32 bits sin signo)

dataView.setInt32(0, 0); // poner a cero el número de 4 bytes, poniendo así todos los bytes a 0

```

DataView es genial cuando almacenamos datos de formato mixto en el mismo buffer. Por ejemplo, cuando almacenamos una secuencia de pares (entero de 16 bits, flotante de 32 bits), DataView permite acceder a ellos fácilmente.

Resumen

ArrayBuffer es el objeto central, una referencia al área de memoria contigua de longitud fija.

Para hacer casi cualquier operación sobre ArrayBuffer, necesitamos una vista.

- Puede ser un TypedArray :
 - Uint8Array, Uint16Array, Uint32Array : para enteros sin signo de 8, 16 y 32 bits.
 - Uint8ClampedArray : para enteros de 8 bits, los “sujeta” en la asignación.
 - Int8Array, Int16Array, Int32Array : para números enteros con signo (pueden ser negativos).
 - Float32Array, Float64Array : para números de punto flotante con signo de 32 y 64 bits.
- O una DataView : la vista que utiliza métodos para especificar un formato, por ejemplo getUint8(offset).

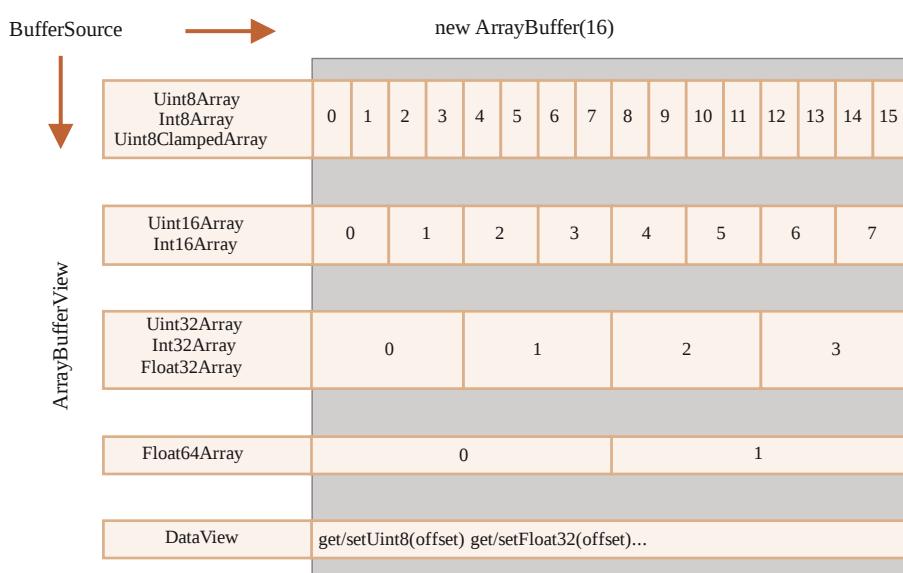
En la mayoría de los casos creamos y operamos directamente sobre arrays tipados, dejando el ArrayBuffer a cubierto, como “denominador común”. Podemos acceder a él como .buffer y hacer otra vista si es necesario.

También hay dos términos adicionales, que se utilizan en las descripciones de los métodos que operan con datos binarios:

- ArrayBufferView es un término paraguas para todos estos tipos de vistas.
- El término BufferSource es un término general para ArrayBuffer o ArrayBufferView.

Veremos estos términos en los próximos capítulos. El término BufferSource es uno de los más comunes, ya que significa “cualquier tipo de datos binarios” : un ArrayBuffer o una vista sobre él.

Aquí tienes la hoja de ruta:



✓ Tareas

Concatenar arrays tipados

Dado un array de `Uint8Array`, escribir una función `concat(arrays)` que devuelva la concatenación de ellos en un único array.

[Abrir en entorno controlado con pruebas.](#)

A solución

TextDecoder y TextEncoder

¿Qué pasa si los datos binarios son en realidad un string? Por ejemplo, recibimos un archivo con datos textuales.

El objeto incorporado [TextDecoder](#) nos permite leer el valor y convertirlo en un string de JavaScript, dados el búfer y la codificación.

Primero necesitamos crearlo:

```
let decoder = new TextDecoder([label], [options]);
```

- `label` – la codificación, `utf-8` por defecto, pero `big5`, `windows-1251` y muchos otros también son soportados.
- `options` – objeto opcional:
 - `fatal` – booleano, si es `true` arroja una excepción por caracteres inválidos (no-decodificable), de otra manera (por defecto) son reemplazados con el carácter `\uFFFD`.
 - `ignoreBOM` – booleano, si es `true` entonces ignora BOM (una marca Unicode de orden de bytes opcional), raramente es necesario.

...Y luego decodificar:

```
let str = decoder.decode([input], [options]);
```

- `input` – `BufferSource` para decodificar.
- `options` – objeto opcional:
 - `stream` – true para decodificación de secuencias, cuando el `decoder` es usado repetidamente para fragmentos de datos entrantes. En ese caso, un carácter de varios bytes puede ocasionalmente dividirse entre fragmentos. Esta opción le dice al `TextDecoder` que memorice caracteres “incompletos” y que los decodifique cuando venga el siguiente fragmento.

Por ejemplo:

```
let uint8Array = new Uint8Array([72, 111, 108, 97]);  
alert( new TextDecoder().decode(uint8Array) ); // Hola
```

```
let uint8Array = new Uint8Array([228, 189, 160, 229, 165, 189]);  
alert( new TextDecoder().decode(uint8Array) ); // 你好
```

Podemos decodificar una parte del búfer al crear una vista de sub arreglo para ello:

```
let uint8Array = new Uint8Array([0, 72, 111, 108, 97, 0]);  
  
// El string esta en medio  
// crear una nueva vista sobre el string, sin copiar nada  
let binaryString = uint8Array.subarray(1, -1);  
  
alert( new TextDecoder().decode(binaryString) ); // Hola
```

TextEncoder

[TextEncoder](#) hace lo contrario: convierte un string en bytes.

La sintaxis es:

```
let encoder = new TextEncoder();
```

La única codificación que soporta es “utf-8”.

Tiene dos métodos:

- `encode(str)` – regresa un dato de tipo `Uint8Array` de un string.
- `encodeInto(str, destination)` – codifica un `str` en `destination`, este último debe ser de tipo `Uint8Array`.

```
let encoder = new TextEncoder();

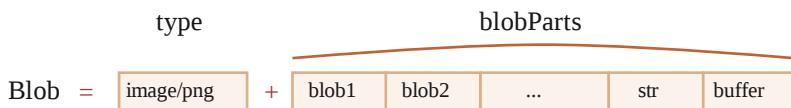
let uint8Array = encoder.encode("Hola");
alert(uint8Array); // 72,111,108,97
```

Blob

Los `ArrayBuffer` y las vistas son parte del estándar ECMA, una parte de JavaScript.

En el navegador, hay objetos de alto nivel adicionales, descritas en la [API de Archivo ↗](#), en particular `Blob`.

`Blob` consta de un tipo especial de cadena (usualmente de tipo MIME), más partes `Blob`: una secuencia de otros objetos `Blob`, cadenas y `BufferSource`.



La sintaxis del constructor es:

```
new Blob(blobParts, opciones);
```

- `blobParts` es un array de valores `Blob/BufferSource/String`.
- `opciones` objeto opcional:
 - `tipo` – `Blob`, usualmente un tipo MIME, por ej. `image/png`,
 - `endings` – para transformar los finales de línea para hacer que el `Blob` coincida con los caracteres de nueva línea del Sistema Operativo actual (`\r\n` or `\n`). Por omisión es `"transparent"` (no hacer nada), pero también puede ser `"native"` (transformar).

Por ejemplo:

```
// crear un Blob a partir de una cadena
let blob = new Blob(["<html>...</html>"], {type: 'text/html'});
// observación: el primer argumento debe ser un array [...]

// crear un Blob a partir de un array tipado y cadenas
let hello = new Uint8Array([72, 101, 108, 108, 111]); // "Hello" en formato binario

let blob = new Blob([hello, ' ', 'world'], {type: 'text/plain'});
```

Podemos extraer porciones del `Blob` con:

```
blob.slice([byteStart], [byteEnd], [contentType]);
```

- `byteStart` – el byte inicial, por omisión es 0.
- `byteEnd` – el último byte (exclusivo, por omisión es el final).
- `contentType` – el `tipo` del nuevo blob, por omisión es el mismo que la fuente.

Los argumentos son similares a `array.slice`, los números negativos también son permitidos.

los objetos `Blob` son inmutables

No podemos cambiar datos directamente en un `Blob`, pero podemos obtener partes de un `Blob`, crear nuevos objetos `Blob` a partir de ellos, mezclarlos en un nuevo `Blob` y así por el estilo.

Este comportamiento es similar a las cadenas de JavaScript: no podemos cambiar un carácter en una cadena, pero podemos hacer una nueva, corregida.

Blob como URL

Un Blob puede ser utilizado fácilmente como una URL para `<a>`, `` u otras etiquetas, para mostrar su contenido.

Gracias al `tipo`, también podemos descargar/cargar objetos `Blob`, y el `tipo` se convierte naturalmente en `Content-Type` en solicitudes de red.

Empecemos con un ejemplo simple. Al hacer click en un link, descargas un `Blob` dinámicamente generado con contenido `hello world` en forma de archivo:

```
<!-- descargar atributos fuerza al navegador a descargar en lugar de navegar -->
<a download="hello.txt" href="#" id="link">Descargar</a>

<script>
let blob = new Blob(["Hello, world!"], {type: 'text/plain'});

link.href = URL.createObjectURL(blob);
</script>
```

También podemos crear un link dinámicamente en JavaScript y simular un click con `link.click()`, y la descarga inicia automáticamente.

Este es un código similar que permite al usuario descargar el `Blob` creado dinámicamente, sin HTML:

```
let link = document.createElement('a');
link.download = 'hello.txt';

let blob = new Blob(['Hello, world!'], {type: 'text/plain'});

link.href = URL.createObjectURL(blob);

link.click();

URL.revokeObjectURL(link.href);
```

`URL.createObjectURL` toma un `Blob` y crea una URL única para él, con la forma `blob:<origin>/<uuid>`.

Así es como se ve el valor de `link.href`:

```
blob:https://javascript.info/1e67e00e-860d-40a5-89ae-6ab0cbee6273
```

Por cada URL generada por `URL.createObjectURL` el navegador almacena un URL → `Blob` mapeado internamente. Así que las URLs son cortas, pero permiten acceder al `Blob`.

Una URL generada (y por lo tanto su enlace) solo es válida en el documento actual, mientras está abierto. Y este permite referenciar al `Blob` en ``, `<a>`, básicamente cualquier otro objeto que espera un URL.

También hay efectos secundarios. Mientras haya un mapeado para un `Blob`, el `Blob` en sí mismo se guarda en la memoria. El navegador no puede liberarlo.

El mapeado se limpia automáticamente al vaciar un documento, así los objetos `Blob` son liberados. Pero si una aplicación es de larga vida, entonces eso no va a pasar pronto.

Entonces, si creamos una URL, este `Blob` se mantendrá en la memoria, incluso si ya no se necesita.

`URL.revokeObjectURL(url)` elimina la referencia el mapeo interno, además de permitir que el `Blob` sea borrado (si ya no hay otras referencias), y que la memoria sea liberada.

En el último ejemplo, intentamos que el `Blob` sea utilizado una sola vez, para descargas instantáneas, así llamamos `URL.revokeObjectURL(link.href)` inmediatamente.

En el ejemplo anterior con el link HTML cliqueable, no llamamos `URL.revokeObjectURL(link.href)`, porque eso puede hacer la URL del `Blob` inválido. Después de la revocación, como el mapeo es eliminado, la URL ya no volverá a funcionar.

Blob a base64

Una alternativa a `URL.createObjectURL` es convertir un `Blob` en una cadena codificada en base64.

Esa codificación representa datos binarios como una cadena ultra segura de caracteres “legibles” con códigos ASCII desde el 0 al 64. Y lo que es más importante, podemos utilizar codificación en las “URLs de datos”.

Un [URL de datos ↗](#) tiene la forma `data:[<mediatype>][;base64],<data>`. Podemos usar suficientes URLs por doquier, junto a URLs “regulares”.

Por ejemplo, aquí hay una sonrisa:

```
` o arrastrar y soltar u otras interfaces del navegador. En este caso el archivo obtiene la información del Sistema Operativo.

Como `File` (Archivo) hereda de `Blob`, objetos de tipo `File` tienen las mismas propiedades, mas:

- `name` – el nombre del archivo,
- `lastModified` – la marca de tiempo de la última modificación.

Así es como obtenemos un objeto `File` desde `<input type="file">`:

```

<input type="file" onchange="showFile(this)">

<script>
function showFile(input) {
 let file = input.files[0];

 alert(`File name: ${file.name}`); // e.g my.png
 alert(`Last modified: ${file.lastModified}`); // e.g 1552830408824
}
</script>

```

### Por favor tome nota:

El input puede seleccionar varios archivos, por lo que `input.files` es un array de dichos archivos . En este caso tenemos un solo archivo por lo que solo es necesario usar `input.files[0]`.

## FileReader

[FileReader](#) ↗ es un objeto con el único propósito de leer datos desde objetos de tipo `Blob` (por lo tanto `File` también).

El entrega los datos usando eventos debido a que leerlos desde el disco puede tomar un tiempo.

El constructor:

```
let reader = new FileReader(); // sin argumentos
```

Los métodos principales:

- `readAsArrayBuffer(blob)` – lee los datos en formato binario `ArrayBuffer`.
- `readAsText(blob, [codificación])` – lee los datos como una cadena de texto con la codificación dada (por defecto es `utf-8`).
- `readAsDataURL(blob)` – lee los datos binarios y los codifica como [Datos URIs] en base 64 ([https://developer.mozilla.org/es/docs/Web/HTTP/Basics\\_of\\_HTTP/Datos\\_URIs](https://developer.mozilla.org/es/docs/Web/HTTP/Basics_of_HTTP/Datos_URIs)).
- `abort()` – cancela la operación.

La opción del método `read*` depende de qué formato preferimos y cómo vamos a usar los datos.

- `readAsArrayBuffer` – para archivos binarios, en donde se hacen operaciones binarias de bajo nivel. Para operaciones de alto nivel, como slicing, `File` hereda de `Blob` por lo que podemos llamarlas directamente sin tener que leer.
- `readAsText` – para archivos de texto, cuando necesitamos obtener una cadena.
- `readAsDataURL` – cuando necesitamos usar estos datos como valores de `src` en `img` u otras etiquetas html. Hay otra alternativa para leer archivos de ese tipo como discutimos en el capítulo [Blob](#): `URL.createObjectURL(file)`.

Mientras se va realizando la lectura, suceden varios eventos:

- `loadstart` – la carga comenzó.
- `progress` – ocurre mientras se lee.
- `load` – lectura completada, sin errores.
- `abort` – `abort()` ha sido llamado.
- `error` – ha ocurrido un error .
- `loadend` – la lectura finalizó exitosa o no .

Cuando la lectura finaliza, podemos acceder al resultado como:

- `reader.result` el resultado (si fue exitoso)
- `reader.error` el error (si hubo fallo).

Los mas ampliamente usados son seguramente `load` y `error`.

Un ejemplo de como leer un archivo:

```
<input type="file" onchange="readFile(this)">

<script>
function readFile(input) {
 let file = input.files[0];

 let reader = new FileReader();

 reader.readAsText(file);

 reader.onload = function() {
 console.log(reader.result);
 };
}
```

```
reader.onerror = function() {
 console.log(reader.error);
};

}
</script>
```

### **i** `FileReader` para blobs

Como mencionamos en el capítulo [Blob](#), `FileReader` no solo lee archivos sino también cualquier blob.

Podemos usarlo para convertir un blob a otro formato:

- `readAsArrayBuffer(blob)` – a `ArrayBuffer`,
- `readAsText(blob, [encoding])` – a una cadena (una alternativa al `TextDecoder`),
- `readAsDataURL(blob)` – a Datos URI en base 64.

### **i** `FileReaderSync` está disponible dentro de Web Workers

Para los Web Workers también existe una variante síncrona de `FileReader` llamada [FileReaderSync ↗](#).

Sus métodos `read*` no generan eventos sino que devuelven un resultado como las funciones regulares.

Esto es solo dentro de un Web Worker, debido a que demoras en llamadas síncronas mientras se lee el archivo en Web Worker no son tan importantes. No afectan la página.

## Resumen

Los objetos `File` heredan de `Blob`.

Además de los métodos y propiedades de `Blob`, los objetos `File` también tienen las propiedades `name` y `lastModified` mas la habilidad interna de leer del sistema de archivos. Usualmente obtenemos los objetos `File` mediante la entrada del usuario con `<input>` o eventos Drag'n'Drop (`ondragend`).

Los objetos `FileReader` pueden leer desde un archivo o un blob en uno de estos tres formatos:

- String (`readAsText`).
- `ArrayBuffer` (`readAsArrayBuffer`).
- Datos URI codificado en base 64 (`readAsDataURL`).

En muchos casos no necesitamos leer el contenido de un archivo como hicimos con los blobs, podemos crear un enlace corto con `URL.createObjectURL(file)` y asignárselo a un `<a>` o `<img>`. De esta manera el archivo puede ser descargado, mostrado como una imagen o como parte de un canvas, etc.

Y si vamos a mandar un `File` por la red, es fácil utilizando APIs como `XMLHttpRequest` o `fetch` que aceptan nativamente objetos `File`.

## Solicitudes de red

### Fetch

JavaScript puede enviar peticiones de red al servidor y cargar nueva información siempre que se necesite.

Por ejemplo, podemos utilizar una petición de red para:

- Crear una orden,
- Cargar información de usuario,
- Recibir las últimas actualizaciones desde un servidor,
- ...etc.

...Y todo esto sin la necesidad de refrescar la página.

Se utiliza el término global "AJAX" (abreviado Asynchronous JavaScript And XML, en español: "JavaScript y XML Asincrónico") para referirse a las peticiones de red originadas desde JavaScript. Sin embargo, no estamos necesariamente condicionados a utilizar XML dado que el término es antiguo y es por esto que el acrónimo XML se encuentra aquí. Probablemente lo hayáis visto anteriormente.

Existen múltiples maneras de enviar peticiones de red y obtener información de un servidor.

Comenzaremos con el método `fetch()` que es moderno y versátil. Este método no es soportado por navegadores antiguos (sin embargo se puede incluir un polyfill), pero es perfectamente soportado por los navegadores actuales y modernos.

La sintaxis básica es la siguiente:

```
let promise = fetch(url, [options])
```

- `url` – representa la dirección URL a la que deseamos acceder.
- `options` – representa los parámetros opcionales, como puede ser un método o los encabezados de nuestra petición, etc.

Si no especificamos ningún `options`, se ejecutará una simple petición GET, la cual descargará el contenido de lo especificado en el `url`.

El navegador lanzará la petición de inmediato y devolverá una promesa (promise) que luego será utilizada por el código invocado para obtener el resultado.

Por lo general, obtener una respuesta es un proceso de dos pasos.

**Primero, la promesa `promise`, devuelta por `fetch`, resuelve la respuesta con un objeto de la clase incorporada `Response` ↗ tan pronto como el servidor responde con los encabezados de la petición.**

En este paso, podemos chequear el status HTTP para poder ver si nuestra petición ha sido exitosa o no, y chequear los encabezados, pero aún no disponemos del cuerpo de la misma.

La promesa es rechazada si el `fetch` no ha podido establecer la petición HTTP, por ejemplo, por problemas de red o si el sitio especificado en la petición no existe. Estados HTTP anormales, como el 404 o 500 no generan errores.

Podemos visualizar los estados HTTP en las propiedades de la respuesta:

- `status` – código de estado HTTP, por ejemplo: 200.
- `ok` – booleana, `true` si el código de estado HTTP es 200 a 299.

Ejemplo:

```
let response = await fetch(url);

if (response.ok) { // si el HTTP-status es 200-299
 // obtener cuerpo de la respuesta (método debajo)
 let json = await response.json();
} else {
 alert("Error-HTTP: " + response.status);
}
```

**Segundo, para obtener el cuerpo de la respuesta, necesitamos utilizar un método adicional.**

`Response` provee múltiples métodos basados en promesas para acceder al cuerpo de la respuesta en distintos formatos:

- `response.text()` – lee y devuelve la respuesta en formato texto,
- `response.json()` – convierte la respuesta como un JSON,
- `response.formData()` – devuelve la respuesta como un objeto `FormData` (explicado en [el siguiente capítulo](#)),
- `response.blob()` – devuelve la respuesta como `Blob` (datos binarios tipados),
- `response.arrayBuffer()` – devuelve la respuesta como un objeto `ArrayBuffer` (representación binaria de datos de bajo nivel),
- Adicionalmente, `response.body` es un objeto [ReadableStream](#) ↗, el cual nos permite acceder al cuerpo como si fuera un stream y leerlo por partes. Veremos un ejemplo de esto más adelante.

Por ejemplo, si obtenemos un objeto de tipo JSON con los últimos commits de GitHub:

```
let url = 'https://api.github.com/repos/javascript-tutorial/en.javascript.info/commits';
let response = await fetch(url);

let commits = await response.json(); // leer respuesta del cuerpo y devolver como JSON
```

```
alert(commits[0].author.login);
```

O también usando promesas, en lugar de `await`:

```
fetch('https://api.github.com/repos/javascript-tutorial/en.javascript.info/commits')
 .then(response => response.json())
 .then(commits => alert(commits[0].author.login));
```

Para obtener la respuesta como texto, `await response.text()` en lugar de `.json()`:

```
let response = await fetch('https://api.github.com/repos/javascript-tutorial/en.javascript.info/commits');

let text = await response.text(); // leer cuerpo de la respuesta como texto

alert(text.slice(0, 80) + '...');
```

Como demostración de una lectura en formato binario, hagamos un `fetch` y mostremos una imagen del logotipo de "especificación `fetch`" ↗ (ver capítulo [Blob](#) para más detalles acerca de las operaciones con `Blob`):

```
let response = await fetch('/article/fetch/logo-fetch.svg');

let blob = await response.blob(); // download as Blob object

// crear tag para imagen
let img = document.createElement('img');
img.style = 'position:fixed;top:10px;left:10px;width:100px';
document.body.append(img);

// mostrar
img.src = URL.createObjectURL(blob);

setTimeout(() => { // ocultar luego de tres segundos
 img.remove();
 URL.revokeObjectURL(img.src);
}, 3000);
```

### ⚠ Importante:

Podemos elegir un solo método de lectura para el cuerpo de la respuesta.

Si ya obtuvimos la respuesta con `response.text()`, entonces `response.json()` no funcionará, dado que el contenido del cuerpo ya ha sido procesado.

```
let text = await response.text(); // cuerpo de respuesta obtenido y procesado
let parsed = await response.json(); // fallo (ya fue procesado)
```

## Encabezados de respuesta

Los encabezados de respuesta están disponibles como un objeto de tipo Map dentro del `response.headers`.

No es exactamente un Map, pero posee métodos similares para obtener de manera individual encabezados por nombre o si quisieramos recorrerlos como un objeto:

```
let response = await fetch('https://api.github.com/repos/javascript-tutorial/en.javascript.info/commits');

// obtenemos un encabezado
alert(response.headers.get('Content-Type')); // application/json; charset=utf-8

// iteramos todos los encabezados
for (let [key, value] of response.headers) {
 alert(`$key = ${value}`);
}
```

## Encabezados de petición

Para especificar un encabezado en nuestro `fetch`, podemos utilizar la opción `headers`. La misma posee un objeto con los encabezados salientes, como se muestra en el siguiente ejemplo:

```
let response = fetch(protectedUrl, {
 headers: {
 Authentication: 'secret'
 }
});
```

...Pero existe una [lista de encabezados ↗](#) que no pueden ser especificados:

- `Accept-Charset`, `Accept-Encoding`
- `Access-Control-Request-Headers`
- `Access-Control-Request-Method`
- `Connection`
- `Content-Length`
- `Cookie`, `Cookie2`
- `Date`
- `DNT`
- `Expect`
- `Host`
- `Keep-Alive`
- `Origin`
- `Referer`
- `TE`
- `Trailer`
- `Transfer-Encoding`
- `Upgrade`
- `Via`
- `Proxy-*`
- `Sec-*`

Estos encabezados nos aseguran que nuestras peticiones HTTP sean controladas exclusivamente por el navegador, de manera correcta y segura.

## Peticiones POST

Para ejecutar una petición `POST`, o cualquier otro método, utilizaremos las opciones de `fetch`:

- `method` – método HTTP, por ej: `POST`,
- `body` – cuerpo de la respuesta, cualquiera de las siguientes:
  - cadena de texto (ej. JSON-encoded),
  - Objeto `FormData`, para enviar información como `multipart/form-data`,
  - `Blob` / `BufferSource` para enviar información en formato binario,
  - [URLSearchParams](#), para enviar información en cifrado `x-www-form-urlencoded` (no utilizado frecuentemente).

El formato JSON es el más utilizado.

Por ejemplo, el código debajo envía la información `user` como un objeto JSON:

```
let user = {
 nombre: 'Juan',
 apellido: 'Perez'
};
```

```

let response = await fetch('/article/fetch/post/user', {
 method: 'POST',
 headers: {
 'Content-Type': 'application/json; charset=utf-8'
 },
 body: JSON.stringify(user)
});

let result = await response.json();
alert(result.message);

```

Tener en cuenta, si la respuesta del `body` es una cadena de texto, entonces el encabezado `Content-Type` será especificado como `text/plain; charset=UTF-8` por defecto.

Pero, como vamos a enviar un objeto JSON, en su lugar utilizaremos la opción `headers` especificada a `application/json`, que es la opción correcta `Content-Type` para información en formato JSON.

## Enviando una imagen

También es posible enviar datos binarios con `fetch`, utilizando los objetos `Blob` o `BufferSource`.

En el siguiente ejemplo, utilizaremos un `<canvas>` donde podremos dibujar utilizando nuestro ratón. Haciendo click en el botón “enviar” enviará la imagen al servidor:

```

<body style="margin:0">
 <canvas id="canvasElem" width="100" height="80" style="border:1px solid"></canvas>

 <input type="button" value="Enviar" onclick="submit()">

 <script>
 canvasElem.onmousemove = function(e) {
 let ctx = canvasElem.getContext('2d');
 ctx.lineTo(e.clientX, e.clientY);
 ctx.stroke();
 };

 async function submit() {
 let blob = await new Promise(resolve => canvasElem.toBlob(resolve, 'image/png'));
 let response = await fetch('/article/fetch/post/image', {
 method: 'POST',
 body: blob
 });

 // el servidor responde con una confirmación y el tamaño de nuestra imagen
 let result = await response.json();
 alert(result.message);
 }
 </script>
</body>

```



Enviar

Una aclaración, aquí no especificamos el `Content-Type` de manera manual, precisamente porque el objeto `Blob` posee un tipo incorporado (en este caso `image/png`, el cual es generado por la función `toBlob`). Para objetos `Blob` ese es el valor por defecto del encabezado `Content-Type`.

Podemos reescribir la función `submit()` sin utilizar `async/await` de la siguiente manera:

```

function submit() {
 canvasElem.toBlob(function(blob) {
 fetch('/article/fetch/post/image', {
 method: 'POST',
 body: blob
 })
 .then(response => response.json())
 .then(result => alert(JSON.stringify(result, null, 2)))
 });
}

```

```
 }, 'image/png');
}
```

## Resumen

Una petición fetch típica está formada por dos llamadas `await`:

```
let response = await fetch(url, options); // resuelve con los encabezados de respuesta
let result = await response.json(); // accede al cuerpo de respuesta como json
```

También se puede acceder sin utilizar `await`:

```
fetch(url, options)
 .then(response => response.json())
 .then(result => /* procesa resultado */)
```

Propiedades de respuesta:

- `response.status` – Código HTTP de la respuesta.
- `response.ok` – Devuelve `true` si el código HTTP es 200-299.
- `response.headers` – Objeto simil-Map que contiene los encabezados HTTP.

Métodos para obtener el cuerpo de la respuesta:

- `response.text()` – lee y devuelve la respuesta en formato texto,
- `response.json()` – convierte la respuesta como un JSON,
- `response.formData()` – devuelve la respuesta como un objeto `FormData` (codificación `multipart/form-data`, explicado en [el siguiente capítulo](#)),
- `response.blob()` – devuelve la respuesta como `Blob` (datos binarios tipados),
- `response.arrayBuffer()` – devuelve la respuesta como un objeto `ArrayBuffer` (datos binarios de bajo nivel)

Opciones de fetch hasta el momento:

- `method` – método HTTP,
- `headers` – un objeto los encabezados de la petición (no todos los encabezados están permitidos),
- `body` – los datos/información a enviar (cuerpo de la petición) como `string`, `FormData`, `BufferSource`, `Blob` u objeto `UrlSearchParams`.

En los próximos capítulos veremos más sobre opciones y casos de uso para `fetch`.

## ✓ Tareas

### Fetch de usuarios de GitHub

Crear una función `async` llamada `getUsers(names)`, que tome como parámetro un arreglo de logins de GitHub, obtenga el listado de usuarios de GitHub indicado y devuelva un arreglo de usuarios de GitHub.

La url de GitHub con la información de usuario específica `USERNAME` es:

`https://api.github.com/users/USERNAME`.

En el ambiente de prueba (sandbox) hay un ejemplo de referencia.

Detalles a tener en cuenta:

1. Debe realizarse una única petición `fetch` por cada usuario.
2. Para que la información esté disponible lo antes posible las peticiones no deben ejecutarse de una por vez.
3. Si alguna de las peticiones fallara o si el usuario no existiese, la función debe devolver `null` en el resultado del arreglo.

[Abrir en entorno controlado con pruebas.](#) ↗

## FormData

Este capítulo trata sobre el envío de formularios HTML: con o sin archivos, con campos adicionales y cosas similares.

Los objetos [FormData](#) pueden ser de ayuda en esta tarea. Tal como habrás supuesto, éste es el objeto encargado de representar los datos de los formularios HTML.

El constructor es:

```
let formData = new FormData([form]);
```

Si se le brinda un elemento HTML `form`, el objeto automáticamente capturará sus campos.

Lo que hace especial al objeto `FormData` es que los métodos de red, tales como `fetch`, pueden aceptar un objeto `FormData` como el cuerpo. Es codificado y enviado como `Content-Type: multipart/form-data`.

Desde el punto de vista del servidor, se ve como una entrega normal.

### Enviando un formulario simple

Enviamos un formulario simple.

Tal como se puede ver, es prácticamente una línea:

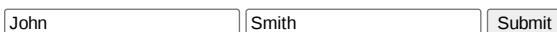
```
<form id="formElem">
 <input type="text" name="name" value="John">
 <input type="text" name="surname" value="Smith">
 <input type="submit">
</form>

<script>
 formElem.onsubmit = async (e) => {
 e.preventDefault();

 let response = await fetch('/article/formdata/post/user', {
 method: 'POST',
 body: new FormData(formElem)
 });

 let result = await response.json();

 alert(result.message);
 };
</script>
```



En este ejemplo, el código del servidor no es representado ya que está fuera de nuestro alcance. El servidor acepta la solicitud POST y responde "Usuario registrado".

### Métodos de FormData

Contamos con métodos para poder modificar los campos del `FormData`:

- `formData.append(name, value)` – agrega un campo al formulario con el nombre `name` y el valor `value`,
- `formData.append(name, blob, fileName)` – agrega un campo tal como si se tratara de un `<input type="file">`, el tercer argumento `fileName` establece el nombre del archivo (no el nombre del campo), tal como si se tratara del nombre del archivo en el sistema de archivos del usuario,
- `formData.delete(name)` – elimina el campo de nombre `name`,
- `formData.get(name)` – obtiene el valor del campo con el nombre `name`,
- `formData.has(name)` – en caso de que exista el campo con el nombre `name`, devuelve `true`, de lo contrario `false`

Un formulario técnicamente tiene permitido contar con muchos campos con el mismo atributo `name`, por lo que múltiples llamadas a `append` agregarán más campos con el mismo nombre.

Por otra parte existe un método `set`, con la misma sintaxis que `append`. La diferencia está en que `.set` remueve todos los campos con el `name` que se le ha pasado, y luego agrega el nuevo campo. De este modo nos aseguramos de que exista solamente un campo con determinado `name`, el resto es tal como en `append`:

- `formData.set(name, value)`,
- `formData.set(name, blob, fileName)`.

También es posible iterar por los campos del objeto `formData` utilizando un bucle `for...of`:

```
let formData = new FormData();
formData.append('key1', 'value1');
formData.append('key2', 'value2');

// Se listan los pares clave/valor
for(let [name, value] of formData) {
 alert(`#${name} = ${value}`); // key1 = value1, luego key2 = value2
}
```

## Enviando un formulario con un archivo

El formulario siempre es enviado como `Content-Type: multipart/form-data`, esta codificación permite enviar archivos. Por lo tanto los campos `<input type="file">` también son enviados, tal como sucede en un envío normal.

Aquí un ejemplo con un formulario de este tipo:

```
<form id="formElem">
 <input type="text" name="firstName" value="John">
 Imagen: <input type="file" name="picture" accept="image/*">
 <input type="submit">
</form>

<script>
 formElem.onsubmit = async (e) => {
 e.preventDefault();

 let response = await fetch('/article/formdata/post/user-avatar', {
 method: 'POST',
 body: new FormData(formElem)
 });

 let result = await response.json();

 alert(result.message);
 };
</script>
```

John Imagen: Choose File No file chosen Submit

## Enviando un formulario con datos Blob

Tal como pudimos ver en el capítulo [Fetch](#), es fácil enviar datos binarios generados dinámicamente (por ejemplo una imagen) como `Blob`. Podemos proporcionarlos directamente en un `fetch` con el parámetro `body`.

De todos modos, en la práctica suele ser conveniente enviar la imagen como parte del formulario junto a otra metadata tal como el nombre y no de forma separada.

Además los servidores suelen ser más propensos a aceptar formularios multipart, en lugar de datos binarios sin procesar.

Este ejemplo envía una imagen desde un `<canvas>` junto con algunos campos más, como un formulario utilizando `FormData`:

```
<body style="margin:0">
 <canvas id="canvasElem" width="100" height="80" style="border:1px solid"></canvas>

 <input type="button" value="Submit" onclick="submit()">
```

```

<script>
 canvasElem.onmousemove = function(e) {
 let ctx = canvasElem.getContext('2d');
 ctx.lineTo(e.clientX, e.clientY);
 ctx.stroke();
 };

 async function submit() {
 let imageBlob = await new Promise(resolve => canvasElem.toBlob(resolve, 'image/png'));

 let formData = new FormData();
 formData.append("firstName", "John");
 formData.append("image", imageBlob, "image.png");

 let response = await fetch('/article/formdata/post/image-form', {
 method: 'POST',
 body: formData
 });
 let result = await response.json();
 alert(result.message);
 }
}

</script>
</body>

```



**Submit**

Nota como la imagen `Blob` es agregada:

```
formData.append("image", imageBlob, "image.png");
```

Es lo mismo que si hubiera un campo `<input type="file" name="image">` en el formulario, y el usuario enviará un archivo con nombre `"image.png"` (3er argumento) con los datos `imageBlob` (2do argumento) desde su sistema de archivos.

El servidor lee el formulario `form-data` y el archivo tal como si de un formulario regular se tratara.

## Resumen

Los objetos [FormData](#) son utilizados para capturar un formulario HTML y enviarlo utilizando `fetch` u otro método de red.

Podemos crear el objeto con `new FormData(form)` desde un formulario HTML, o crear un objeto sin un formulario en absoluto y agregar los campos con los siguientes métodos:

- `formData.append(nombre, valor)`
- `formData.append(nombre, blob, nombreDeArchivo)`
- `formData.set(nombre, valor)`
- `formData.set(nombre, blob, nombreDeArchivo)`

Nótese aquí dos particularidades:

1. El método `set` remueve campos con el mismo nombre, mientras que `append` no. Esta es la única diferencia entre estos dos métodos.
2. Para enviar un archivo, se requiere de tres argumentos, el último argumento es el nombre del archivo, el cual normalmente es tomado desde el sistema de archivos del usuario por el `<input type="file">`.

Otros métodos son:

- `formData.delete(nombre)`
- `formData.get(nombre)`
- `formData.has(nombre)`

¡Esto es todo!

## Fetch: Progreso de la descarga

El método `fetch` permite rastrear el progreso de *descarga*.

Ten en cuenta: actualmente no hay forma de que `fetch` rastree el progreso de *carga*. Para ese propósito, utiliza [XMLHttpRequest](#), lo cubriremos más adelante.

Para rastrear el progreso de la descarga, podemos usar la propiedad `response.body`. Esta propiedad es un `ReadableStream`, un objeto especial que proporciona la transmisión del cuerpo fragmento a fragmento tal como viene. Estas se describen en la especificación de la [API de transmisiones](#).

A diferencia de `response.text()`, `response.json()` y otros métodos, `response.body` da control total sobre el proceso de lectura, y podemos contar cuánto se consume en cualquier momento.

Aquí está el bosquejo del código que lee la respuesta de `response.body`:

```
// en lugar de response.json() y otros métodos
const reader = response.body.getReader();

// bucle infinito mientras el cuerpo se descarga
while(true) {
 // done es true para el último fragmento
 // value es Uint8Array de los bytes del fragmento
 const {done, value} = await reader.read();

 if (done) {
 break;
 }

 console.log(`Recibí ${value.length} bytes`)
}
```

El resultado de la llamada `await reader.read()` es un objeto con dos propiedades:

- `done` – `true` cuando la lectura está completa, de lo contrario `false`.
- `value` – una matriz de tipo `bytes`: `Uint8Array`.

 **Por favor tome nota:**

La API de transmisiones también describe la iteración asincrónica sobre `ReadableStream` con el bucle `for await..of`, pero aún no es ampliamente compatible (consulta [problemas del navegador](#)), por lo que usamos el bucle `while`.

Recibimos fragmentos de respuesta en el bucle, hasta que finaliza la carga, es decir: hasta que `done` se convierte en `true`.

Para registrar el progreso, solo necesitamos que cada `value` de fragmento recibido agregue su longitud al contador.

Aquí está el ejemplo funcional completo que obtiene la respuesta y registra el progreso en la consola, seguido de su explicación:

```
// Paso 1: iniciar la búsqueda y obtener un lector
let response = await fetch('https://api.github.com/repos/javascript-tutorial/es.javascript.info/commits?per_page=100');

const reader = response.body.getReader();

// Paso 2: obtener la longitud total
const contentLength = +response.headers.get('Content-Length');

// Paso 3: leer los datos
let receivedLength = 0; // cantidad de bytes recibidos hasta el momento
let chunks = []; // matriz de fragmentos binarios recibidos (comprende el cuerpo)
while(true) {
 const {done, value} = await reader.read();

 if (done) {
 break;
 }

 chunks.push(value);
}

console.log(`Total de bytes recibidos: ${receivedLength}`);
```

```

receivedLength += value.length;

console.log(`Recibí ${receivedLength} de ${contentLength}`)
}

// Paso 4: concatenar fragmentos en un solo Uint8Array
let chunksAll = new Uint8Array(receivedLength); // (4.1)
let position = 0;
for(let chunk of chunks) {
 chunksAll.set(chunk, position); // (4.2)
 position += chunk.length;
}

// Paso 5: decodificar en un string
let result = new TextDecoder("utf-8").decode(chunksAll);

// ¡Hemos terminado!
let commits = JSON.parse(result);
alert(commits[0].author.login);

```

Expliquemos esto paso a paso:

- Realizamos `fetch` como de costumbre, pero en lugar de llamar a `response.json()`, obtenemos un lector de transmisión `response.body.getReader()`.

Ten en cuenta que no podemos usar ambos métodos para leer la misma respuesta: usa un lector o un método de respuesta para obtener el resultado.

- Antes de leer, podemos averiguar la longitud completa de la respuesta del encabezado `Content-Length`.

Puede estar ausente para solicitudes cross-origin (consulta el capítulo [Fetch: Cross-Origin Requests](#)) y, bueno, técnicamente un servidor no tiene que configurarlo. Pero generalmente está en su lugar.

- Llama a `await reader.read()` hasta que esté listo.

Recopilamos fragmentos de respuesta en la matriz `chunks`. Eso es importante, porque después de consumir la respuesta, no podremos “releerla” usando `response.json()` u otra forma (puedes intentarlo, habrá un error).

- Al final, tenemos `chunks` – una matriz de fragmentos de bytes `Uint8Array`. Necesitamos unirlos en un solo resultado. Desafortunadamente, no hay un método simple que los concatene, por lo que hay un código para hacerlo:

- Creamos `chunksAll = new Uint8Array(selectedLength)` – una matriz del mismo tipo con la longitud combinada.
- Luego usa el método `.set(chunk, position)` para copiar cada `chunk` uno tras otro en él.
- Tenemos el resultado en `chunksAll`. Sin embargo, es una matriz de bytes, no un string.

Para crear un string, necesitamos interpretar estos bytes. El `TextDecoder` nativo hace exactamente eso. Luego podemos usar el resultado en `JSON.parse`, si es necesario.

¿Qué pasa si necesitamos contenido binario en lugar de un string? Eso es aún más sencillo. Reemplaza los pasos 4 y 5 con una sola línea que crea un `Blob` de todos los fragmentos:

```
let blob = new Blob(chunks);
```

Al final tenemos el resultado (como un string o un blob, lo que sea conveniente) y el seguimiento del progreso en el proceso.

Una vez más, ten en cuenta que eso no es para el progreso de carga (hasta ahora eso no es posible con `fetch`), solo para el progreso de *descarga*.

Además, si el tamaño es desconocido, deberíamos chequear `receivedLength` en el bucle y cortarlo en cuanto alcance cierto límite, así los `chunks` no agotarán la memoria.

## Fetch: Abort

Como sabemos `fetch` devuelve una promesa. Y generalmente JavaScript no tiene un concepto de “abortar” una promesa. Entonces, ¿cómo podemos abortar una llamada al método `fetch`? Por ejemplo si las acciones del usuario en nuestro sitio indican que `fetch` no se necesitará más.

Existe para esto de forma nativa un objeto especial: `AbortController`. Puede ser utilizado para abortar no solo `fetch` sino otras tareas asincrónicas también.

Su uso es muy sencillo:

## El objeto AbortController

Crear un controlador:

```
let controller = new AbortController();
```

Este controlador es un objeto extremadamente simple.

- Tiene un único método `abort()`,
- y una única propiedad `signal` que permite establecerle escuchadores de eventos.

Cuando `abort()` es invocado:

- `controller.signal` emite el evento "abort".
- La propiedad `controller.signal.aborted` toma el valor `true`.

Generalmente tenemos dos partes en el proceso:

1. El que ejecuta la operación de cancelación, genera un listener que escucha a `controller.signal`.
2. El que cancela: este llama a `controller.abort()` cuando es necesario.

Tal como se muestra a continuación (por ahora sin `fetch`):

```
let controller = new AbortController();
let signal = controller.signal;

// La parte que ejecuta la operación de cancelación
// obtiene el objeto "signal"
// y genera un listener que se dispara cuando es llamado controller.abort()
signal.addEventListener('abort', () => alert("abort!"));

// El que cancela (más tarde en cualquier punto):
controller.abort(); // abort!

// El evento se dispara y signal.aborted se vuelve true
alert(signal.aborted); // true
```

Como podemos ver, `AbortController` es simplemente la vía para pasar eventos `abort` cuando `abort()` es llamado sobre él.

Podríamos implementar alguna clase de escucha de evento en nuestro código por nuestra cuenta, sin el objeto `AbortController` en absoluto.

Pero lo valioso es que `fetch` sabe cómo trabajar con el objeto `AbortController`, está integrado con él.

## Uso con fetch

Para posibilitar la cancelación de `fetch`, pasa la propiedad `signal` de un `AbortController` como una opción de `fetch`:

```
let controller = new AbortController();
fetch(url, {
 signal: controller.signal
});
```

El método `fetch` conoce cómo trabajar con `AbortController`. Este escuchará eventos `abort` sobre `signal`.

Ahora, para abortar, llamamos `controller.abort()`:

```
controller.abort();
```

Terminamos: `fetch` obtiene el evento desde `signal` y aborta el requerimiento.

Cuando un fetch es abortado, su promesa es rechazada con un error `AbortError`, así podemos manejarlo, por ejemplo en `try..catch`.

Aquí hay un ejemplo completo con `fetch` abortado después de 1 segundo:

```
// Se abortara en un segundo
let controller = new AbortController();
setTimeout(() => controller.abort(), 1000);

try {
 let response = await fetch('/article/fetch-abort/demo/hang', {
 signal: controller.signal
 });
} catch(err) {
 if (err.name == 'AbortError') { // se maneja el abort()
 alert("Aborted!");
 } else {
 throw err;
 }
}
```

## AbortController es escalable

`AbortController` es escalable, permite cancelar múltiples fetch de una vez.

Aquí hay un bosquejo de código que de muchos fetch de `url` en paralelo, y usa un simple controlador para abortarlos a todos:

```
let urls = [...]; // una lista de urls para utilizar fetch en paralelo

let controller = new AbortController();

// un array de promesas fetch
let fetchJobs = urls.map(url => fetch(url, {
 signal: controller.signal
}));

let results = await Promise.all(fetchJobs);

// si controller.abort() es llamado,
// se abortaran todas las solicitudes fetch
```

En el caso de tener nuestras propias tareas asincrónicas aparte de `fetch`, podemos utilizar un único `AbortController` para detenerlas junto con `fetch`.

Solo es necesario escuchar el evento `abort` en nuestras tareas:

```
let urls = [...];
let controller = new AbortController();

let ourJob = new Promise((resolve, reject) => { // nuestra tarea
 ...
 controller.signal.addEventListener('abort', reject);
});

let fetchJobs = urls.map(url => fetch(url, { // varios fetch
 signal: controller.signal
}));

// Se espera por la finalización de los fetch y nuestra tarea
let results = await Promise.all([...fetchJobs, ourJob]);

// en caso de que se llame al método controller.abort() desde algún sitio,
// se abortan todos los fetch y nuestra tarea.
```

## Resumen

- `AbortController` es un simple objeto que genera un evento `abort` sobre su propiedad `signal` cuando el método `abort()` es llamado (y también establece `signal.aborted` en `true`).
- `fetch` está integrado con él: pasamos la propiedad `signal` como opción, y entonces `fetch` la escucha, así se vuelve posible abortar `fetch`.
- Podemos usar `AbortController` en nuestro código. La interacción "llamar `abort()`" → "escuchar evento `abort`" es simple y universal. Podemos usarla incluso sin `fetch`.

## Fetch: Cross-Origin Requests

Si enviamos una petición `fetch` hacia otro sitio seguramente fallará.

Por ejemplo, probemos una petición a `https://example.com`:

```
try {
 await fetch('https://example.com');
} catch(err) {
 alert(err); // Failed to fetch
}
```

El método `fetch` falla, tal como lo esperábamos.

El concepto clave aquí es *el origen (origin)*, triple combinación de dominio/puerto/protocolo.

Las solicitudes de origen cruzado `Cross-origin requests` (aquellas que son enviadas hacia otro dominio --incluso subdominio--, protocolo o puerto), requieren de unas cabeceras especiales desde el sitio remoto.

Esta política es denominada "CORS", por sus siglas en inglés Cross-Origin Resource Sharing.

### ¿Por qué CORS es necesario?, Una breve historia

CORS existe para proteger Internet de los hackers malvados.

En verdad... Déjame contarte un breve resumen de esta historia.

#### Durante muchos años un script de un sitio no podía acceder al contenido de otro sitio.

Esta simple, pero poderosa regla, fue parte fundamental de la seguridad de Internet. Por ejemplo, un script malicioso desde el sitio `hacker.com` no podía acceder a la casilla de correo en el sitio `gmail.com`. La gente se podía sentir segura.

Así mismo en ese momento, JavaScript no tenía ningún método especial para realizar solicitudes de red. Simplemente era un lenguaje juguete para decorar páginas web.

Pero los desarrolladores web demandaron más poder. Una variedad de trucos fueron inventados para poder pasar por alto las limitaciones, y realizar solicitudes a otros sitios.

### Utilizando formularios

Una forma de comunicarse con otros servidores es y era utilizando un `<form>`. Se lo utilizaba para enviar el resultado hacia un `<iframe>`, y de este modo mantenerse en el mismo sitio:

```
<!-- objetivo del form -->
<iframe name="iframe"></iframe>

<!-- Un formulario puede ser generado de forma dinámica y ser enviado por JavaScript -->
<form target="iframe" method="POST" action="http://another.com/...">
 ...
</form>
```

Entonces, de este modo era posible realizar solicitudes GET/POST hacia otro sitio, incluso sin métodos de red, ya que los formularios pueden enviar mensajes a cualquier sitio. Pero ya que no es posible acceder al contenido de un `<iframe>` de otro sitio, esto evita que sea posible leer la respuesta.

Para ser precisos, en realidad había trucos para eso, requerían scripts especiales tanto en el iframe como en la página. Entonces la comunicación con el iframe era técnicamente posible. Pero ya no hay necesidad de entrar en detalles, dejemos a los dinosaurios descansar en paz.

### Utilizando scripts

Otro truco es en el modo de utilizar la etiqueta `script`. Un script puede tener cualquier origen `src`, con cualquier dominio, tal como `<script src="http://another.com/...">`. De este modo es posible ejecutar un script de cualquier sitio web.

Si un sitio, por ejemplo, `another.com` requiere exponer datos con este tipo de acceso, se utilizaba el protocolo llamado en ese entonces “JSONP (JSON con padding)” .

Veamos como se utilizaba.

Digamos que, en nuestro sitio es necesario obtener datos de `http://another.com`, como podría ser el pronóstico del tiempo:

1. Primero, adelantándonos, creamos una función global para aceptar los datos, por ejemplo: `gotWeather` .

```
// 1. Se declara la función para procesar los datos del tiempo
function gotWeather({ temperature, humidity }) {
 alert(`temperature: ${temperature}, humidity: ${humidity}`);
}
```

2. Entonces creamos una etiqueta `<script>` donde `src="http://another.com/weather.json?callback=gotWeather"` , utilizando el nombre de nuestra función como un parámetro `callback` , dentro de la URL.

```
let script = document.createElement('script');
script.src = `http://another.com/weather.json?callback=gotWeather`;
document.body.append(script);
```

3. El servidor remoto `another.com` de forma dinámica genera un script que invoca el método `gotWeather(...)` con los datos que nosotros necesitamos recibir.

```
// The expected answer from the server looks like this:
gotWeather({
 temperature: 25,
 humidity: 78
});
```

4. Entonces el script remoto carga y es ejecutado, la función `gotWeather` se invoca, y ya que es nuestra función, obtenemos los datos.

Esto funciona, y no viola la seguridad ya que ambos sitios acuerdan en intercambiar los datos de este modo. Y cuando ambos lados concuerdan, definitivamente no se trata de un hackeo. Aún hay servicios que proveen este tipo de acceso, lo que puede ser útil ya que funciona en navegadores obsoletos.

Tiempo después aparecieron métodos de red en los navegadores para JavaScript.

Al comienzo, las solicitudes de origen cruzado fueron prohibidas, pero luego de prolongadas discusiones se permitieron, requiriendo consentimiento explícito por parte del servidor, esto expresado en cabezales especiales.

## Solicitudes seguras

Existen dos tipos de solicitudes de origen cruzado:

1. Solicitudes seguras.
2. Todas las demás.

Las solicitudes seguras son más fáciles de hacer, comenzemos con ellas.

Una solicitud es segura si cumple dos condiciones:

1. [método seguro](#) : GET, POST o HEAD
2. [Cabeceras seguras](#) – Las únicas cabeceras permitidas son:

- `Accept`,
- `Accept-Language`,
- `Content-Language`,
- `Content-Type` con el valor `application/x-www-form-urlencoded`, `multipart/form-data` o `text/plain`.

Cualquier otra solicitud es considerada “insegura”. Por lo tanto, una solicitud con el método `PUT` o con una cabecera HTTP `API-Key` no cumple con las limitaciones.

**La diferencia esencial es que una solicitud segura puede ser realizada mediante un `<form>` o un `<script>`, sin la necesidad de utilizar un método especial.**

Por lo tanto, incluso un servidor obsoleto debería ser capaz de aceptar una solicitud segura.

Contrario a esto, las solicitudes con cabeceras no estándar o métodos como el `DELETE` no pueden ser creados de este modo. Durante mucho tiempo no fue posible para JavaScript realizar este tipo de solicitudes. Por lo que un viejo servidor podía asumir que ese tipo de solicitudes provenía desde una fuente privilegiada, "ya que una página web es incapaz de enviarlas".

Cuando intentamos realizar una solicitud insegura, el navegador envía una solicitud especial de "pre-vuelo" consultando al servidor: ¿está de acuerdo en aceptar tal solicitud de origen cruzado o no?

Y, salvo que el servidor lo confirme de forma explícita, cualquier solicitud insegura no es enviada.

Vayamos ahora a los detalles.

## CORS para solicitudes seguras

Si una solicitud es de origen cruzado, el navegador siempre le agregará una cabecera `Origin`.

Por ejemplo, si realizamos una solicitud de `https://anywhere.com/request` a `https://javascript.info/page`, las cabeceras podrían ser algo así:

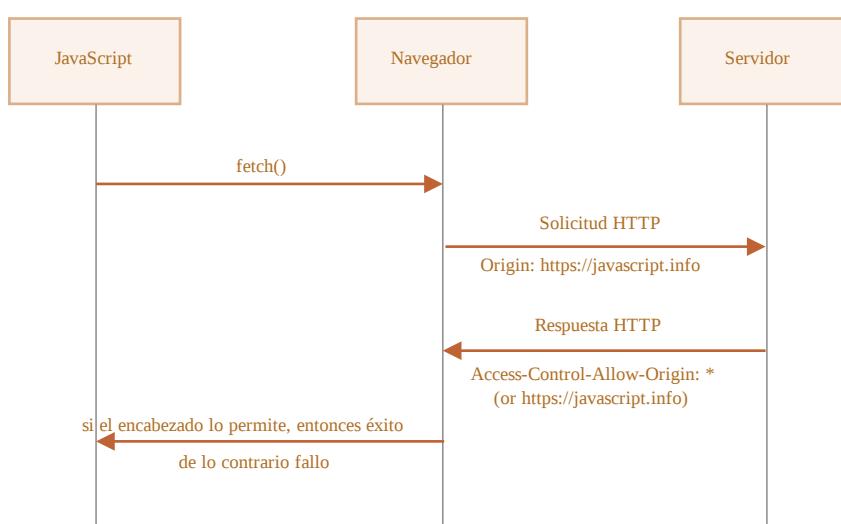
```
GET /request
Host: anywhere.com
Origin: https://javascript.info
...
...
```

Tal como se puede ver, la cabecera `Origin` contiene exactamente el origen (protocolo/dominio/puerto), sin el path.

El servidor puede inspeccionar el origen `Origin` y, si esta de acuerdo en aceptar ese tipo de solicitudes, agrega una cabecera especial `Access-Control-Allow-Origin` a la respuesta. Esta cabecera debe contener el origen permitido (en nuestro caso `https://javascript.info`), o un asterisco `*`. En ese caso la respuesta es satisfactoria, de otro modo falla.

El navegador cumple el papel de mediador de confianza:

1. Ante una solicitud de origen cruzado, se asegura de que se envíe el origen correcto.
2. Chequea que la respuesta contenga la cabecera `Access-Control-Allow-Origin`, de ser así JavaScript tiene permitido acceder a la respuesta, de no ser así la solicitud falla con un error.



Aquí tenemos un ejemplo de una respuesta permisiva desde el servidor:

```
200 OK
Content-Type:text/html; charset=UTF-8
Access-Control-Allow-Origin: https://javascript.info
```

## Cabeceras de respuesta

Para las respuestas de origen cruzado, por defecto JavaScript sólo puede acceder a las cabeceras llamadas “seguras”:

- Cache-Control
- Content-Language
- Content-Length
- Content-Type
- Expires
- Last-Modified
- Pragma

El acceso a otro tipo de cabeceras de la respuesta generará un error.

Para permitir a JavaScript acceso a cualquier otra cabecera de respuesta, el servidor debe incluir la cabecera `Access-Control-Expose-Headers`. Este campo contiene una lista separada por comas de las cabeceras inseguras que podrán ser accesibles.

Por ejemplo:

```
200 OK
Content-Type:text/html; charset=UTF-8
Content-Length: 12345
Content-Encoding: gzip
API-Key: 2c9de507f2c54aa1
Access-Control-Allow-Origin: https://javascript.info
Access-Control-Expose-Headers: Content-Encoding, API-Key
```

Con tal cabecera, `Access-Control-Expose-Headers`, el script tendrá permitido acceder a los valores de las cabeceras `Content-Encoding` y `API-Key` de la respuesta.

## Solicitudes “inseguras”

Podemos utilizar cualquier método HTTP: no únicamente `GET/POST`, sino también `PATCH`, `DELETE` y otros.

Hace algún tiempo nadie podía siquiera imaginar que un sitio web pudiera realizar ese tipo de solicitudes. Por lo que aún existen servicios web que cuando reciben un método no estándar los consideran como una señal de que: “Del otro lado no hay un navegador”. Ellos pueden tener en cuenta esto cuando revisan los derechos de acceso.

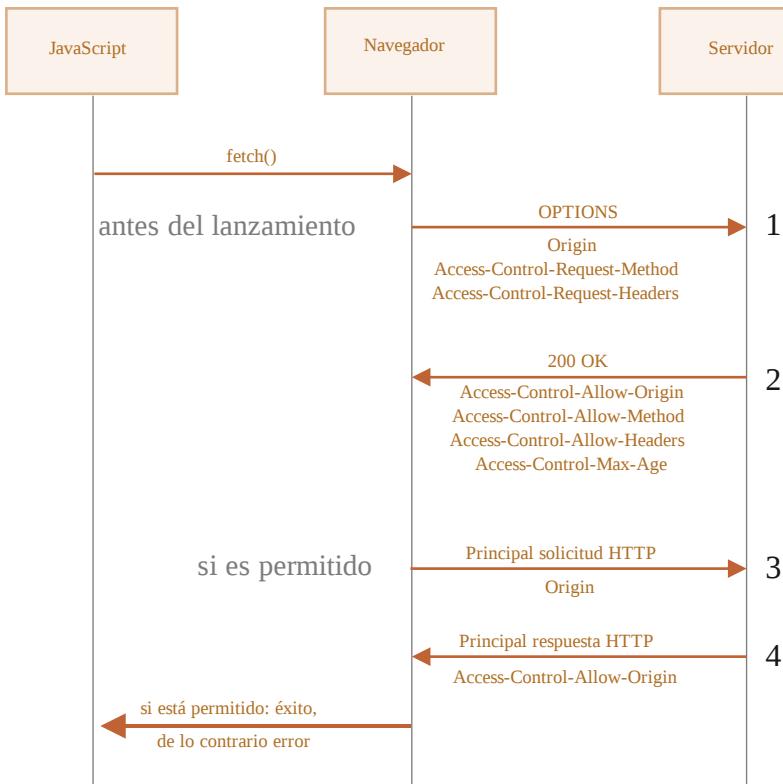
Por lo tanto, para evitar malentendidos, cualquier solicitud “insegura” (Estas que no podían ser realizadas en los viejos tiempos), no será realizada por el navegador en forma directa. Antes, enviará una solicitud preliminar llamada solicitud de “pre-vuelo”, solicitando que se le concedan los permisos.

Una solicitud de “pre-vuelo” utiliza el método `OPTIONS`, sin contenido en el cuerpo y con tres cabeceras:

- `Access-Control-Request-Method`, cabecera que contiene el método de la solicitud “insegura”.
- `Access-Control-Request-Headers` provee una lista separada por comas de las cabeceras inseguras de la solicitud.
- `Origin` cabecera que informa de dónde viene la solicitud. (como `https://javascript.info`)

Si el servidor está de acuerdo con lo solicitado, entonces responderá con el código de estado 200 y un cuerpo vacío:

- `Access-Control-Allow-Origin` debe ser `*` o el origen de la solicitud, tal como `https://javascript.info`, para permitir el acceso.
- `Access-Control-Allow-Methods` contiene el método permitido.
- `Access-Control-Allow-Headers` contiene un listado de las cabeceras permitidas.
- Además, la cabecera `Access-Control-Max-Age` puede especificar el número máximo de segundos que puede recordar los permisos. Por lo que el navegador no necesita volver a requerirlos en las próximas solicitudes.



Vamos a ver cómo funciona paso a paso, mediante un ejemplo para una solicitud de origen cruzado `PATCH` (este método suele utilizarse para actualizar datos):

```
let response = await fetch('https://site.com/service.json', {
 method: 'PATCH',
 headers: {
 'Content-Type': 'application/json',
 'API-Key': 'secret'
 }
});
```

Hay tres motivos por los cuales esta solicitud no es segura (una es suficiente):

- Método `PATCH`
- `Content-Type` no es del tipo: `application/x-www-form-urlencoded`, `multipart/form-data`, `text/plain`.
- Cabecera `API-Key` “insegura”.

### Paso 1 (solicitud de pre-vuelo)

Antes de enviar una solicitud de este tipo, el navegador envía una solicitud de pre-vuelo que se ve de este modo:

```
OPTIONS /service.json
Host: site.com
Origin: https://javascript.info
Access-Control-Request-Method: PATCH
Access-Control-Request-Headers: Content-Type, API-Key
```

- Método: `OPTIONS`.
- El path – exactamente el mismo que el de la solicitud principal: `/service.json`.
- Cabeceras especiales de origen cruzado (Cross-origin):
  - `Origin` – el origen de la fuente.
  - `Access-Control-Request-Method` – método solicitado.
  - `Access-Control-Request-Headers` – listado separado por comas de las cabeceras “inseguras”.

### Paso 2 (solicitud de pre-vuelo)

El servidor debe responder con el código de estado 200 y las cabeceras:

- Access-Control-Allow-Origin: https://javascript.info
- Access-Control-Allow-Methods: PATCH
- Access-Control-Allow-Headers: Content-Type, API-Key .

Esto permitirá la comunicación futura, de otro modo se disparará un error.

Si el servidor espera otro método y cabeceras en el futuro, tiene sentido permitirlos por adelantado agregándolos a la lista.

Por ejemplo, esta respuesta habilita además los métodos PUT , DELETE y otras cabeceras:

```
200 OK
Access-Control-Allow-Origin: https://javascript.info
Access-Control-Allow-Methods: PUT,PATCH,DELETE
Access-Control-Allow-Headers: API-Key,Content-Type,If-Modified-Since,Cache-Control
Access-Control-Max-Age: 86400
```

Ahora el navegador puede ver que PATCH se encuentra dentro de la cabecera Access-Control-Allow-Methods y Content-Type, API-Key dentro de la lista Access-Control-Allow-Headers , por lo que permitirá enviar la solicitud principal.

Si se encuentra con una cabecera Access-Control-Max-Age con determinada cantidad de segundos, entonces los permisos son almacenados en el caché por ese determinado tiempo. La solicitud anterior será cacheada por 86400 segundos (un día). Durante ese marco de tiempo, las solicitudes siguientes no requerirán la solicitud de pre-vuelo. Asumiendo que están dentro de lo permitido en la respuesta cacheada, serán enviadas de forma directa.

### Paso 3 (solicitud real)

Una vez el pre-vuelo se realiza de forma satisfactoria, el navegador realiza la solicitud principal. El algoritmo aquí es el mismo que el utilizado para una solicitud segura.

La solicitud principal tiene la cabecera Origin (ya que se trata de una solicitud de origen cruzado):

```
PATCH /service.json
Host: site.com
Content-Type: application/json
API-Key: secret
Origin: https://javascript.info
```

### Paso 4 (respuesta real)

El server no debe olvidar agregar la cabecera Access-Control-Allow-Origin a la respuesta principal. Un pre-vuelo exitoso no lo libera de esto:

```
Access-Control-Allow-Origin: https://javascript.info
```

Entonces JavaScript es capaz de leer la respuesta principal del servidor.

**i Por favor tome nota:**

La solicitud de pre-vuelo ocurre “detrás de escena”, es invisible a JavaScript.

JavaScript únicamente obtiene la respuesta a la solicitud principal o un error en caso de que el servidor no otorgue la autorización.

## Credenciales

Una solicitud de origen cruzado realizada por código JavaScript, por defecto no provee ningún tipo de credenciales (cookies o autenticación HTTP).

Esto es poco común para solicitudes HTTP. Usualmente una solicitud a un sitio http://site.com es acompañada por todas las cookies de ese dominio. Pero una solicitud de origen cruzado realizada por métodos de JavaScript son una excepción.

Por ejemplo, fetch('http://another.com') no enviará ninguna cookie, ni siquiera (!) esas que pertenecen al dominio another.com .

## ¿Por qué?

El motivo de esto es que una solicitud con credenciales es mucho más poderosa que sin ellas. Si se permitiera, esto garantizaría a JavaScript el completo poder de actuar en representación del usuario y de acceder a información sensible utilizando sus credenciales.

¿En verdad el servidor confía lo suficiente en el script? En ese caso el servidor deberá enviar explicitamente que permite solicitudes con credenciales mediante otra cabecera especial.

Para permitir el envío de credenciales en `fetch`, necesitamos agregar la opción `credentials: "include"`, de este modo:

```
fetch('http://another.com', {
 credentials: "include"
});
```

Ahora `fetch` envía cookies originadas desde `another.com` con las solicitudes a ese sitio.

Si el servidor está de acuerdo en aceptar solicitudes *con credenciales*, debe agregar la cabecera `Access-Control-Allow-Credentials: true` a la respuesta, además de `Access-Control-Allow-Origin`.

Por ejemplo:

```
200 OK
Access-Control-Allow-Origin: https://javascript.info
Access-Control-Allow-Credentials: true
```

Cabe destacar que: `Access-Control-Allow-Origin` no se puede utilizar con un asterisco `*` para solicitudes con credenciales. Tal como se muestra arriba debe proveer el origen exacto. Esto es una medida adicional de seguridad, para asegurar de que el servidor conozca exactamente en quién confiar para que le envíe este tipo de solicitudes.

## Resumen

Desde el punto de vista del navegador, existen dos tipos de solicitudes de origen cruzado: solicitudes “seguras” y todas las demás.

[Solicitudes seguras](#) deben cumplir las siguientes condiciones:

- Método: GET, POST o HEAD.
- Cabeceras – solo podemos establecer:
  - `Accept`
  - `Accept-Language`
  - `Content-Language`
  - `Content-Type` con el valor `application/x-www-form-urlencoded`, `multipart/form-data` o `text/plain`.

La diferencia esencial es que las solicitudes seguras eran posibles desde los viejos tiempos utilizando las etiquetas `<form>` o `<script>`, mientras que las solicitudes “inseguras” fueron imposibles para el navegador durante mucho tiempo.

Por lo tanto, en la práctica, la diferencia se encuentra en que las solicitudes seguras son realizadas de forma directa, utilizando la cabecera `Origin`, mientras que para las otras el navegador realiza una solicitud extra de “pre-vuelo” para requerir la autorización.

### Para una solicitud segura:

- → El navegador envía una cabecera `Origin` con el origen.
- ← Para solicitudes sin credenciales (no enviadas por defecto), el servidor debe establecer:
  - `Access-Control-Allow-Origin` como `*` o el mismo valor que en `Origin`.
- ← Para solicitudes con credenciales, el servidor deberá establecer:
  - `Access-Control-Allow-Origin` con el mismo valor que en `Origin`.
  - `Access-Control-Allow-Credentials` en `true`

Adicionalmente, para garantizar a JavaScript acceso a cualquier cabecera de la respuesta, con excepción de `Cache-Control`, `Content-Language`, `Content-Type`, `Expires`, `Last-Modified` o `Pragma`, el servidor debe

agregarlas como permitidas en la lista de la cabecera `Access-Control-Expose-Headers`.

**Para solicitudes inseguras, se utiliza una solicitud preliminar “pre-vuelo” antes de la solicitud principal:**

- → El navegador envía una solicitud del tipo `OPTIONS` a la misma URL, con las cabeceras:
  - `Access-Control-Request-Method` con el método requerido.
  - `Access-Control-Request-Headers` listado de las cabeceras inseguras.
- ← El servidor debe responder con el código de estado 200 y las cabeceras:
  - `Access-Control-Allow-Methods` con la lista de todos los métodos permitidos,
  - `Access-Control-Allow-Headers` con una lista de cabeceras permitidas,
  - `Access-Control-Max-Age` con los segundos en los que se podrá almacenar la autorización en caché.
- Tras lo cual la solicitud es enviada, y se aplica el esquema previo “seguro”.

## ✓ Tareas

### ¿Por qué necesitamos el origen (Origin)?

importancia: 5

Como seguramente ya sepas, existe la cabecera HTTP `Referer`, la cual por lo general contiene la url del sitio que generó la solicitud.

Por ejemplo, cuando solicitamos la url `http://google.com` desde `http://javascript.info/alguna/url`, las cabeceras se ven de este modo:

```
Accept: */*
Accept-Charset: utf-8
Accept-Encoding: gzip, deflate, sdch
Connection: keep-alive
Host: google.com
Origin: http://javascript.info
Referer: http://javascript.info/alguna/url
```

Tal como se puede ver, tanto `Referer` como `Origin` están presentes.

Las preguntas:

1. ¿Por qué la cabecera `Origin` es necesaria, si `Referer` contiene incluso más información?
2. ¿Es posible que no se incluya `Referer` u `Origin`, o que contengan datos incorrectos?

## A solución

## Fetch API

Hasta ahora, sabemos bastante sobre `fetch`.

Veamos el resto de API, para cubrir todas sus capacidades.

### **i Por favor tome nota:**

Ten en cuenta: la mayoría de estas opciones se utilizan con poca frecuencia. Puedes saltarte este capítulo y seguir utilizando bien `fetch`.

Aún así, es bueno saber lo que puede hacer `fetch`, por lo que si surge la necesidad, puedes regresar y leer los detalles.

Aquí está la lista completa de todas las posibles opciones de `fetch` con sus valores predeterminados (alternativas en los comentarios):

```
let promise = fetch(url, {
 method: "GET", // POST, PUT, DELETE, etc.
 headers: {
 // el valor del encabezado Content-Type generalmente se establece automáticamente
```

```

 // dependiendo del cuerpo de la solicitud
 "Content-Type": "text/plain;charset=UTF-8"
},
body: undefined, // string, FormData, Blob, BufferSource, o URLSearchParams
referrer: "about:client", // o "" para no enviar encabezado de Referrer,
// o una URL del origen actual
referrerPolicy: "strict-origin-when-cross-origin", // no-referrer-when-downgrade, no-referrer, origin, same-origin...
mode: "cors", // same-origin, no-cors
credentials: "same-origin", // omit, include
cache: "default", // no-store, reload, no-cache, force-cache, o only-if-cached
redirect: "follow", // manual, error
integrity: "", // un hash, como "sha256-abcdef1234567890"
keepalive: false, // true
signal: undefined, // AbortController para cancelar la solicitud
window: window // null
});

```

Una lista impresionante, ¿verdad?

Cubrimos completamente `method`, `headers` y `body` en el capítulo [Fetch](#).

La opción `signal` está cubierta en [Fetch: Abort](#).

Ahora exploremos el resto de capacidades.

## referrer, referrerPolicy

Estas opciones gobiernan cómo `fetch` establece el encabezado HTTP `Referer`.

Por lo general, ese encabezado se establece automáticamente y contiene la URL de la página que realizó la solicitud. En la mayoría de los escenarios, no es importante en absoluto, a veces, por motivos de seguridad, tiene sentido eliminarlo o acortarlo.

**La opción `referrer` permite establecer cualquier `Referer` (dentro del origen actual) o eliminarlo.**

Para no enviar ningún referrer, establece un string vacío:

```

fetch('/page', {
 referrer: "" // sin encabezado Referer
});

```

Para establecer otra URL dentro del origen actual:

```

fetch('/page', {
 // asumiendo que estamos en https://javascript.info
 // podemos establecer cualquier encabezado Referer, pero solo dentro del origen actual
 referrer: "https://javascript.info/anotherpage"
});

```

**La opción `referrerPolicy` establece reglas generales para `Referer`.**

Las solicitudes se dividen en 3 tipos:

1. Solicitud al mismo origen.
2. Solicitud a otro origen.
3. Solicitud de HTTPS a HTTP (de protocolo seguro a no seguro).

A diferencia de la opción `referrer` que permite establecer el valor exacto de `Referer`, `referrerPolicy` indica al navegador las reglas generales para cada tipo de solicitud.

Los valores posibles se describen en la [Especificación de la política Referrer ↗](#):

- `"strict-origin-when-cross-origin"` – El valor predeterminado. Para el mismo origen, envía el `Referer` completo. Para el envío cross-origin envía solo el origen, a menos que sea una solicitud HTTPS → HTTP, entonces no envía nada.
- `"no-referrer-when-downgrade"` – el `Referer` completo se envía siempre, a menos que enviemos una solicitud de HTTPS a HTTP (a un protocolo menos seguro).
- `"no-referrer"` – nunca envía `Referer`.

- `"origin"` – solo envía el origen en `Referer`, no la URL de la página completa. Por ejemplo, solo `http://site.com` en lugar de `http://site.com/path`.
- `"origin-when-cross-origin"` – envía el `Referrer` completo al mismo origen, pero solo la parte de origen para solicitudes cross-origin (como se indica arriba).
- `"same-origin"` – envía un `Referer` completo al mismo origen, pero no un `Referer` para solicitudes cross-origin.
- `"strict-origin"` – envía solo el origen, no envía `Referer` para solicitudes HTTPS → HTTP.
- `"unsafe-url"` – envía siempre la URL completa en `Referer`, incluso para solicitudes HTTPS → HTTP.

Aquí hay una tabla con todas las combinaciones:

Valor	Al mismo origen	A otro origen	HTTPS → HTTP
<code>"no-referrer"</code>	-	-	-
<code>"no-referrer-when-downgrade"</code>	completo	completo	-
<code>"origin"</code>	origen	origen	origen
<code>"origin-when-cross-origin"</code>	completo	origen	origen
<code>"same-origin"</code>	completo	-	-
<code>"strict-origin"</code>	origen	origen	-
<code>"strict-origin-when-cross-origin"</code> or <code>""</code> (predeterminado)	completo	origen	-
<code>"unsafe-url"</code>	completo	completo	completo

Digamos que tenemos una zona de administración con una estructura de URL que no debería conocerse desde fuera del sitio.

Si enviamos un `fetch`, entonces de forma predeterminada siempre envía el encabezado `Referer` con la URL completa de nuestra página (excepto cuando solicitamos de HTTPS a HTTP, entonces no hay `Referer`).

Por ejemplo, `Referer: https://javascript.info/admin/secret/paths`.

Si queremos que otros sitios web solo conozcan la parte del origen, no la ruta de la URL, podemos configurar la opción:

```
fetch('https://another.com/page', {
 // ...
 referrerPolicy: "origin-when-cross-origin" // Referer: https://javascript.info
});
```

Podemos ponerlo en todas las llamadas `fetch`, tal vez integrarlo en la biblioteca JavaScript de nuestro proyecto que hace todas las solicitudes y que usa `fetch` por dentro.

Su única diferencia en comparación con el comportamiento predeterminado es que para las solicitudes a otro origen, `fetch` envía solo la parte de origen de la URL (por ejemplo, `https://javascript.info`, sin ruta). Para las solicitudes a nuestro origen, todavía obtenemos el `Referer` completo (quizás útil para fines de depuración).

#### La política Referrer no es solo para `fetch`

La política Referrer, descrita en la [especificación](#), no es solo para `fetch`, sino más global.

En particular, es posible establecer la política predeterminada para toda la página utilizando el encabezado HTTP `Referrer-Policy`, o por enlace, con `<a rel="noreferrer">`.

## mode

La opción `mode` es una protección que evita solicitudes cross-origin ocasionales:

- `"cors"` – por defecto, se permiten las solicitudes cross-origin predeterminadas, como se describe en [Fetch: Cross-Origin Requests](#),
- `"same-origin"` – las solicitudes cross-origin están prohibidas,
- `"no-cors"` – solo se permiten solicitudes cross-origin seguras.

Esta opción puede ser útil cuando la URL de `fetch` proviene de un tercero y queremos un “interruptor de apagado” para limitar las capacidades cross-origin.

## credentials

La opción `credentials` especifica si `fetch` debe enviar cookies y encabezados de autorización HTTP con la solicitud.

- `"same-origin"` – el valor predeterminado, no enviar solicitudes cross-origin,
- `"include"` – enviar siempre, requiere `Access-Control-Allow-Credentials` del servidor cross-origin para que JavaScript acceda a la respuesta, que se cubrió en el capítulo [Fetch: Cross-Origin Requests](#),
- `"omit"` – nunca enviar, incluso para solicitudes del mismo origen.

## cache

De forma predeterminada, las solicitudes `fetch` utilizan el almacenamiento en caché HTTP estándar. Es decir, respeta los encabezados `Expires`, `Cache-Control`, envía `If-Modified-Since`, y así sucesivamente. Al igual que lo hacen las solicitudes HTTP habituales.

Las opciones de `cache` permiten ignorar el caché HTTP o ajustar su uso:

- `"default"` – `fetch` utiliza reglas y encabezados de caché HTTP estándar,
- `"no-store"` – ignoramos por completo el caché HTTP, este modo se convierte en el predeterminado si configuramos un encabezado `If-Modified-Since`, `If-None-Match`, `If-Unmodified-Since`, `If-Match` o `If-Range`,
- `"reload"` – no toma el resultado del caché HTTP (si corresponde), pero completa el caché con la respuesta (si los encabezados de respuesta lo permiten),
- `"no-cache"` – crea una solicitud condicional si hay una respuesta en caché y una solicitud normal en caso contrario. Llena el caché HTTP con la respuesta,
- `"force-cache"` – usa una respuesta del caché HTTP, incluso si está obsoleta. Si no hay respuesta en el caché HTTP, hace una solicitud HTTP regular, se comporta normalmente,
- `"only-if-cached"` – usa una respuesta del caché HTTP, incluso si está obsoleta. Si no hay respuesta en el caché HTTP, entonces envía un error. Solo funciona cuando `mode` es `"same-origin"`.

## redirect

Normalmente, `fetch` sigue de forma transparente las redirecciones HTTP, como 301, 302, etc.

La opción `redirect` permite cambiar eso:

- `"follow"` – el predeterminado, sigue las redirecciones HTTP,
- `"error"` – error en caso de redireccionamiento HTTP,
- `"manual"` – permite procesar redireccionamiento HTTP manualmente. En caso de redireccionamiento obtendremos un objeto `response` especial, con `response.type="opaqueRedirect"` y cero o vacío en la mayor parte de las demás propiedades.

## integrity

La opción `integrity` permite comprobar si la respuesta coincide con el known-ahead checksum.

Como se describe en la [especificación ↗](#), las funciones hash admitidas son SHA-256, SHA-384 y SHA-512. Puede haber otras dependiendo de un navegador.

Por ejemplo, estamos descargando un archivo y sabemos que su checksum SHA-256 es “abcdef” (un checksum real es más largo, por supuesto).

Lo podemos poner en la opción `integrity`, así:

```
fetch('http://site.com/file', {
 integrity: 'sha256-abcdef'
});
```

Luego, `fetch` calculará SHA-256 por sí solo y lo comparará con nuestro string. En caso de discrepancia, se activa un error.

## keepalive

La opción `keepalive` indica que la solicitud puede “vivir más allá” de la página web que la inició.

Por ejemplo, recopilamos estadísticas sobre cómo el visitante actual usa nuestra página (clics del mouse, fragmentos de página que ve), para analizar y mejorar la experiencia del usuario.

Cuando el visitante abandona nuestra página, nos gustaría guardar los datos en nuestro servidor.

Podemos usar el evento `window.onunload` para eso:

```
window.onunload = Function() {
 fetch('/analytics', {
 method: 'POST',
 body: "statistics",
 keepalive: true
 });
};
```

Normalmente, cuando se descarga un documento, se cancelan todas las solicitudes de red asociadas. Pero la opción `keepalive` le dice al navegador que realice la solicitud en segundo plano, incluso después de salir de la página. Por tanto, esta opción es fundamental para que nuestra solicitud tenga éxito.

Tiene algunas limitaciones:

- No podemos enviar megabytes: el límite de cuerpo para las solicitudes `keepalive` es de 64 KB.
  - Si necesitamos recopilar muchas estadísticas sobre la visita, deberíamos enviarlas regularmente en paquetes, de modo que no quede mucho para la última solicitud `onunload`.
  - Este límite se aplica a todas las solicitudes `keepalive` juntas. En otras palabras, podemos realizar múltiples solicitudes `keepalive` en paralelo, pero la suma de las longitudes de sus cuerpos no debe exceder los 64 KB.
- No podemos manejar la respuesta del servidor si el documento no está cargado. Entonces, en nuestro ejemplo, `fetch` tendrá éxito debido a `keepalive`, pero las funciones posteriores no funcionarán.
  - En la mayoría de los casos, como enviar estadísticas, no es un problema, ya que el servidor simplemente acepta los datos y generalmente envía una respuesta vacía a tales solicitudes.

## Objetos URL

La clase [URL](#) incorporada brinda una interfaz conveniente para crear y analizar URLs.

No hay métodos de networking que requieran exactamente un objeto `URL`, los strings son suficientemente buenos para eso. Así que técnicamente no tenemos que usar `URL`. Pero a veces puede ser realmente útil.

### Creando una URL

La sintaxis para crear un nuevo objeto `URL` es:

```
new URL(url, [base])
```

- `url` – La URL completa o ruta única (si se establece base, mira a continuación),
- `base` – una URL base opcional: si se establece y el argumento `url` solo tiene una ruta, entonces la URL se genera relativa a `base`.

Por ejemplo:

```
let url = new URL('https://javascript.info/profile/admin');
```

Estas dos URLs son las mismas:

```
let url1 = new URL('https://javascript.info/profile/admin');
let url2 = new URL('/profile/admin', 'https://javascript.info');

alert(url1); // https://javascript.info/profile/admin
alert(url2); // https://javascript.info/profile/admin
```

Fácilmente podemos crear una nueva URL basada en la ruta relativa a una URL existente:

```

let url = new URL('https://javascript.info/profile/admin');
let newUrl = new URL('tester', url);

alert(newUrl); // https://javascript.info/profile/tester

```

El objeto `URL` inmediatamente nos permite acceder a sus componentes, por lo que es una buena manera de analizar la url, por ej.:

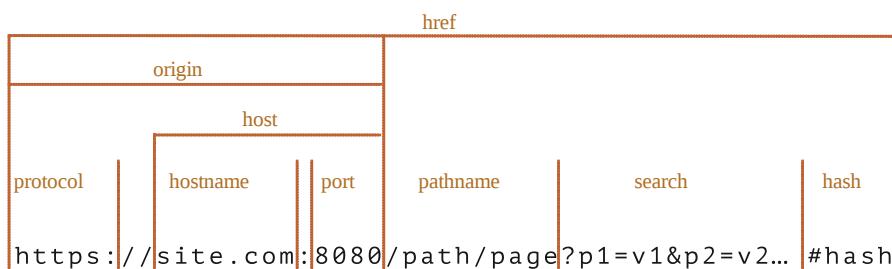
```

let url = new URL('https://javascript.info/url');

alert(url.protocol); // https:
alert(url.host); // javascript.info
alert(url.pathname); // /url

```

Aquí está la hoja de trucos para los componentes URL:



- `href` es la url completa, igual que `url.toString()`
- `protocol` acaba con el carácter dos puntos `:`
- `search` – un string de parámetros, comienza con el signo de interrogación `?`
- `hash` comienza con el carácter de hash `#`
- También puede haber propiedades `user` y `password` si la autenticación HTTP esta presente: `http://login:password@site.com` (no mostrados arriba, raramente usados)

#### **i Podemos pasar objetos URL a métodos de red (y la mayoría de los demás) en lugar de un string**

Podemos usar un objeto `URL` en `fetch` o `XMLHttpRequest`, casi en todas partes donde se espera un URL-string.

Generalmente, un objeto `URL` puede pasarse a cualquier método en lugar de un string, ya que la mayoría de métodos llevarán a cabo la conversión del string, eso convierte un objeto `URL` en un string con URL completa.

## Parámetros de búsqueda “?...”

Digamos que queremos crear una url con determinados parámetros de búsqueda, por ejemplo, `https://google.com/search?query=JavaScript`.

Podemos proporcionarlos en el string URL:

```
new URL('https://google.com/search?query=JavaScript')
```

...Pero los parámetros necesitan estar codificados si contienen espacios, letras no latinas, entre otros (Más sobre eso debajo).

Por lo que existe una propiedad `URL` para eso: `urlSearchParams`, un objeto de tipo [URLSearchParams](#).

Esta proporciona métodos convenientes para los parámetros de búsqueda:

- `append(name, value)` – añade el parámetro por `name`,
- `delete(name)` – elimina el parámetro por `name`,
- `get(name)` – obtiene el parámetro por `name`,

- `getAll(name)` – obtiene todos los parámetros con el mismo `name` (Eso es posible, por ej. `?user=John&user=Pete`),
- `has(name)` – comprueba la existencia del parámetro por `name`,
- `set(name, value)` – establece/reemplaza el parámetro,
- `sort()` – ordena parámetros por `name`, raramente necesitado,
- ...y además es iterable, similar a `Map`.

Un ejemplo con parámetros que contienen espacios y signos de puntuación:

```
let url = new URL('https://google.com/search');

url.searchParams.set('q', 'test me!'); // Parámetro añadido con un espacio y !

alert(url); // https://google.com/search?q=test+me%21

url.searchParams.set('tbs', 'qdr:y'); // Parámetro añadido con dos puntos :

// Los parámetros son automáticamente codificados
alert(url); // https://google.com/search?q=test+me%21&tbs=qdr%3Ay

// Iterar sobre los parámetros de búsqueda (Decodificados)
for(let [name, value] of url.searchParams) {
 alert(` ${name}=${value}`); // q=test me!, then tbs=qdr:y
}
```

## Codificación

Existe un estándar [RFC3986](#) que define cuales caracteres son permitidos en URLs y cuales no.

Esos que no son permitidos, deben ser codificados, por ejemplo letras no latinas y espacios – reemplazados con sus códigos UTF-8, con el prefijo %, tal como %20 (un espacio puede ser codificado con +, por razones históricas, pero esa es una excepción).

La buena noticia es que los objetos `URL` manejan todo eso automáticamente. Nosotros sólo proporcionamos todos los parámetros sin codificar, y luego convertimos la `URL` a string:

```
// Usando algunos caracteres cirílicos para este ejemplo

let url = new URL('https://ru.wikipedia.org/wiki/Тест');

urlSearchParams.set('key', 'т');
alert(url); //https://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D1%81%D1%82?key=%D1%8A
```

Como puedes ver, ambos `Тест` en la ruta url y `т` en el parámetro están codificados.

La URL se alarga, ya que cada letra cirílica es representada con dos bytes en UTF-8, por lo que hay dos entidades %... .

## Codificando strings

En los viejos tiempos, antes de que los objetos `URL` aparecieran, la gente usaba strings para las URL.

A partir de ahora, los objetos `URL` son frecuentemente más convenientes, pero también aún pueden usarse los strings. En muchos casos usando un string se acorta el código.

Aunque si usamos un string, necesitamos codificar/decodificar caracteres especiales manualmente.

Existen funciones incorporadas para eso:

- [encodeURI](#) – Codifica la URL como un todo.
- [decodeURI](#) – La decodifica de vuelta.
- [encodeURIComponent](#) – Codifica un componente URL, como un parametro de búsqueda, un hash, o un pathname.
- [decodeURIComponent](#) – La decodifica de vuelta.

Una pregunta natural es: "¿Cuál es la diferencia entre `encodeURIComponent` y `encodeURI`? ¿Cuándo deberíamos usar una u otra?"

Eso es fácil de entender si miramos a la URL, que está separada en componentes en la imagen de arriba:

```
https://site.com:8080/path/page?p1=v1&p2=v2#hash
```

Como podemos ver, caracteres tales como `:`, `?`, `=`, `&`, `#` son admitidos en URL.

...Por otra parte, si miramos a un único componente URL, como un parámetro de búsqueda, estos caracteres deben estar codificados, para no romper el formateo.

- `encodeURI` Codifica solo caracteres que están totalmente prohibidos en URL
- `encodeURIComponent` Codifica los mismos caracteres, y, en adición a ellos, los caracteres `#`, `$`, `&`, `+`, `,`, `/`, `:`, `;`, `=`, `?` y `@`.

Entonces, para una URL completa podemos usar `encodeURI`:

```
// Usando caracteres cirílicos en el path URL
let url = encodeURI('http://site.com/привет');

alert(url); // http://site.com/%D0%BF%D1%80%D0%B8%D0%B2%D0%B5%D1%82
```

...Mientras que para parámetros URL deberíamos usar `encodeURIComponent` en su lugar:

```
let music = encodeURIComponent('Rock&Roll');

let url = `https://google.com/search?q=${music}`;
alert(url); // https://google.com/search?q=Rock%26Roll
```

Compáralo con `encodeURI`:

```
let music = encodeURI('Rock&Roll');

let url = `https://google.com/search?q=${music}`;
alert(url); // https://google.com/search?q=Rock&Roll
```

Como podemos ver, `encodeURI` no codifica `&`, ya que este es un carácter legítimo en la URL como un todo.

Pero debemos codificar `&` dentro de un parámetro de búsqueda, de otra manera, obtendremos `q=Rock&Roll` - que es realmente `q=Rock` más algún parámetro `Roll` oscuro. No según lo previsto.

Así que debemos usar solo `encodeURIComponent` para cada parámetro de búsqueda, para insertarlo correctamente en el string URL. Lo más seguro es codificar tanto nombre como valor, a menos que estemos absolutamente seguros de que solo haya admitido caracteres

#### Diferencia de codificación comparado con `URL`

Las clases `URL` [🔗](#) y `URLSearchParams` [🔗](#) están basadas en la especificación URI mas reciente: [RFC3986](#) [🔗](#), mientras que las funciones `encode*` están basadas en la versión obsoleta [RFC2396](#) [🔗](#).

Existen algunas diferencias, por ej. las direcciones IPv6 se codifican de otra forma:

```
// Url válida con dirección IPv6
let url = 'http://[2607:f8b0:4005:802::1007]/';

alert(encodeURI(url)); // http://%5B2607:f8b0:4005:802::1007%5D/
alert(new URL(url)); // http://[2607:f8b0:4005:802::1007]/
```

Como podemos ver, `encodeURI` reemplazó los corchetes `[ . . . ]`, eso es incorrecto, la razón es: las urls IPv6 no existían en el tiempo de RFC2396 (August 1998).

Tales casos son raros, las funciones `encode*` mayormente funcionan bien.

## XMLHttpRequest

`XMLHttpRequest` es un objeto nativo del navegador que permite hacer solicitudes HTTP desde JavaScript.

A pesar de tener la palabra “XML” en su nombre, se puede operar sobre cualquier dato, no solo en formato XML. Podemos cargar y descargar archivos, dar seguimiento y mucho más.

Ahora hay un método más moderno `fetch` que en algún sentido hace obsoleto a `XMLHttpRequest`.

En el desarrollo web moderno `XMLHttpRequest` se usa por tres razones:

1. Razones históricas: necesitamos soportar scripts existentes con `XMLHttpRequest`.
2. Necesitamos soportar navegadores viejos, y no queremos `polyfills` (p.ej. para mantener los scripts pequeños).
3. Necesitamos hacer algo que `fetch` no puede todavía, ej. rastrear el progreso de subida.

¿Te suena familiar? Si es así, está bien, adelante con `XMLHttpRequest`. De otra forma, por favor, dirígete a [Fetch](#).

## Lo básico

`XMLHttpRequest` tiene dos modos de operación: sincrónica y asíncrona.

Veamos primero la asíncrona, ya que es utilizada en la mayoría de los casos.

Para hacer la petición, necesitamos seguir 3 pasos:

1. Crear el objeto `XMLHttpRequest`:

```
let xhr = new XMLHttpRequest();
```

El constructor no tiene argumentos.

2. Inicializarlo, usualmente justo después de `new XMLHttpRequest`:

```
xhr.open(method, URL, [async, user, password])
```

Este método especifica los parámetros principales para la petición:

- `method` – método HTTP. Usualmente `"GET"` o `"POST"`.
- `URL` – la URL a solicitar, una cadena, puede ser un objeto `URL`.
- `async` – si se asigna explícitamente a `false`, entonces la petición será asíncrona. Cubriremos esto un poco más adelante.
- `user, password` – usuario y contraseña para autenticación HTTP básica (si se requiere).

Por favor, toma en cuenta que la llamada a `open`, contrario a su nombre, no abre la conexión. Solo configura la solicitud, pero la actividad de red solo empieza con la llamada del método `send`.

3. Enviar.

```
xhr.send([body])
```

Este método abre la conexión y envía la solicitud al servidor. El parámetro adicional `body` contiene el cuerpo de la solicitud.

Algunos métodos como `GET` no tienen un cuerpo. Y otros como `POST` usan el parámetro `body` para enviar datos al servidor. Vamos a ver unos ejemplos de eso más tarde.

4. Escuchar los eventos de respuesta `xhr`.

Estos son los tres eventos más comúnmente utilizados:

- `load` – cuando la solicitud está completa (incluso si el estado HTTP es 400 o 500), y la respuesta se descargó por completo.
- `error` – cuando la solicitud no pudo ser realizada satisfactoriamente, ej. red caída o una URL inválida.
- `progress` – se dispara periódicamente mientras la respuesta está siendo descargada, reporta cuánto se ha descargado.

```
xhr.onload = function() {
 alert(`Cargado: ${xhr.status} ${xhr.response}`);
};
```

```

xhr.onerror = function() { // solo se activa si la solicitud no se puede realizar
 alert(`Error de red`);
};

xhr.onprogress = function(event) { // se dispara periódicamente
 // event.loaded - cuántos bytes se han descargado
 // event.lengthComputable = devuelve true si el servidor envía la cabecera Content-Length (longitud del contenido)
 // event.total - número total de bytes (si `lengthComputable` es `true`)
 alert(`Recibido ${event.loaded} of ${event.total}`);
};

```

Aquí un ejemplo completo. El siguiente código carga la URL en `/article/xmlhttprequest/example/load` desde el servidor e imprime el progreso:

```

// 1. Crea un nuevo objeto XMLHttpRequest
let xhr = new XMLHttpRequest();

// 2. Configuración: solicitud GET para la URL /article/.../load
xhr.open('GET', '/article/xmlhttprequest/example/load');

// 3. Envía la solicitud a la red
xhr.send();

// 4. Esto se llamará después de que la respuesta se reciba
xhr.onload = function() {
 if (xhr.status != 200) { // analiza el estado HTTP de la respuesta
 alert(`Error ${xhr.status}: ${xhr.statusText}`); // ej. 404: No encontrado
 } else { // muestra el resultado
 alert(`Hecho, obtenidos ${xhr.response.length} bytes`); // Respuesta del servidor
 }
};

xhr.onprogress = function(event) {
 if (event.lengthComputable) {
 alert(`Recibidos ${event.loaded} de ${event.total} bytes`);
 } else {
 alert(`Recibidos ${event.loaded} bytes`); // sin Content-Length
 }
};

xhr.onerror = function() {
 alert("Solicitud fallida");
};

```

Una vez el servidor haya respondido, podemos recibir el resultado en las siguientes propiedades de `xhr`:

#### `status`

Código del estado HTTP (un número): `200`, `404`, `403` y así por el estilo, puede ser `0` en caso de una falla no HTTP.

#### `statusText`

Mensaje del estado HTTP (una cadena): usualmente `OK` para `200`, `Not Found` para `404`, `Forbidden` para `403` y así por el estilo.

#### `response (scripts antiguos deben usar responseText)`

El cuerpo de la respuesta del servidor.

También podemos especificar un tiempo límite usando la propiedad correspondiente:

```

xhr.timeout = 10000; // límite de tiempo en milisegundos, 10 segundos

```

Si la solicitud no es realizada con éxito dentro del tiempo dado, se cancela y el evento `timeout` se activa.

### Parámetros de búsqueda URL

Para agregar los parámetros a la URL, como `?nombre=valor`, y asegurar la codificación adecuada, podemos utilizar un objeto [URL](#):

```
let url = new URL('https://google.com/search');
url.searchParams.set('q', 'pruébame!');

// el parámetro 'q' está codificado
xhr.open('GET', url); // https://google.com/search?q=test+me%21
```

## Tipo de respuesta

Podemos usar la propiedad `xhr.responseType` para asignar el formato de la respuesta:

- "" (default) – obtiene una cadena,
- "text" – obtiene una cadena,
- "arraybuffer" – obtiene un `ArrayBuffer` (para datos binarios, ve el capítulo [ArrayBuffer, arrays binarios](#)),
- "blob" – obtiene un `Blob` (para datos binarios, ver el capítulo [Blob](#)),
- "document" – obtiene un documento XML (puede usar XPath y otros métodos XML) o un documento HTML (en base al tipo MIME del dato recibido),
- "json" – obtiene un JSON (automáticamente analizado).

Por ejemplo, obtengamos una respuesta como JSON:

```
let xhr = new XMLHttpRequest();

xhr.open('GET', '/article/xmlhttprequest/example/json');

xhr.responseType = 'json';

xhr.send();

// la respuesta es {"message": "Hola, Mundo!"}
xhr.onload = function() {
 let responseObj = xhr.response;
 alert(responseObj.message); // Hola, Mundo!
};
```

### Por favor tome nota:

En los scripts antiguos puedes encontrar también las propiedades `xhr.responseText` e incluso `xhr.responseXML`.

Existen por razones históricas, para obtener ya sea una cadena o un documento XML. Hoy en día, debemos seleccionar el formato en `xhr.responseType` y obtener `xhr.response` como se demuestra debajo.

## Estados

`XMLHttpRequest` cambia entre estados a medida que avanza. El estado actual es accesible como `xhr.readyState`.

Todos los estados, como en [la especificación](#) :

```
UNSENT = 0; // estado inicial
OPENED = 1; // llamada abierta
HEADERS_RECEIVED = 2; // cabeceras de respuesta recibidas
LOADING = 3; // la respuesta está cargando (un paquete de datos es recibido)
DONE = 4; // solicitud completa
```

Un objeto `XMLHttpRequest` escala en orden `0 → 1 → 2 → 3 → ... → 3 → 4`. El estado `3` se repite cada vez que un paquete de datos se recibe a través de la red.

Podemos seguirlos usando el evento `readystatechange`:

```
xhr.onreadystatechange = function() {
 if (xhr.readyState == 3) {
 // cargando
 }
 if (xhr.readyState == 4) {
 // solicitud finalizada
 }
};
```

Puedes encontrar oyentes del evento `readystatechange` en código realmente viejo, está ahí por razones históricas, había un tiempo cuando no existían `load` y otros eventos. Hoy en día los manipuladores `load/error/progress` lo hacen obsoleto.

## Abortando solicitudes

Podemos terminar la solicitud en cualquier momento. La llamada a `xhr.abort()` hace eso:

```
xhr.abort(); // termina la solicitud
```

Este dispara el evento `abort`, y el `xhr.status` se convierte en `0`.

## Solicitudes sincrónicas

Si en el método `open` el tercer parámetro `async` se asigna como `false`, la solicitud se hace sincrónicamente.

En otras palabras, la ejecución de JavaScript se pausa en el `send()` y se reanuda cuando la respuesta es recibida. Algo como los comandos `alert` o `prompt`.

Aquí está el ejemplo reescrito, el tercer parámetro de `open` es `false`:

```
let xhr = new XMLHttpRequest();

xhr.open('GET', '/article/xmlhttprequest/hello.txt', false);

try {
 xhr.send();
 if (xhr.status != 200) {
 alert(`Error ${xhr.status}: ${xhr.statusText}`);
 } else {
 alert(xhr.response);
 }
} catch(err) { // en lugar de onerror
 alert("Solicitud fallida");
}
```

Puede verse bien, pero las llamadas sincrónicas son rara vez utilizadas porque bloquean todo el JavaScript de la página hasta que la carga está completa. En algunos navegadores se hace imposible hacer scroll. Si una llamada síncrona toma mucho tiempo, el navegador puede sugerir cerrar el sitio web “colgado”.

Algunas capacidades avanzadas de `XMLHttpRequest`, como solicitar desde otro dominio o especificar un tiempo límite, no están disponibles para solicitudes sincrónicas. Tampoco, como puedes ver, la indicación de progreso.

La razón de esto es que las solicitudes sincrónicas son utilizadas muy escasamente, casi nunca. No hablaremos más sobre ellas.

## Cabeceras HTTP

`XMLHttpRequest` permite tanto enviar cabeceras personalizadas como leer cabeceras de la respuesta.

Existen 3 métodos para las cabeceras HTTP:

```
setRequestHeader(name, value)
```

Asigna la cabecera de la solicitud con los valores `name` y `value` provistos.

Por ejemplo:

```
xhr.setRequestHeader('Content-Type', 'application/json');
```

### ⚠ Limitaciones de cabeceras

Muchas cabeceras se administran exclusivamente por el navegador, ej. `Referer` y `Host`. La lista completa está [en la especificación ↗](#).

`XMLHttpRequest` no está permitido cambiarlos, por motivos de seguridad del usuario y la exactitud de la solicitud.

### ⚠ No se pueden eliminar cabeceras

Otra peculiaridad de `XMLHttpRequest` es que no puede deshacer un `setRequestHeader`.

Una vez que una cabecera es asignada, ya está asignada. Llamadas adicionales agregan información a la cabecera, no la sobreescriben.

Por ejemplo:

```
xhr.setRequestHeader('X-Auth', '123');
xhr.setRequestHeader('X-Auth', '456');

// la cabecera será:
// X-Auth: 123, 456
```

### `getResponseHeader(name)`

Obtiene la cabecera de la respuesta con el `name` dado (excepto `Set-Cookie` y `Set-Cookie2`).

Por ejemplo:

```
xhr.getResponseHeader('Content-Type')
```

### `getAllResponseHeaders()`

Devuelve todas las cabeceras de la respuesta, excepto por `Set-Cookie` y `Set-Cookie2`.

Las cabeceras se devuelven como una sola línea, ej.:

```
Cache-Control: max-age=31536000
Content-Length: 4260
Content-Type: image/png
Date: Sat, 08 Sep 2012 16:53:16 GMT
```

El salto de línea entre las cabeceras siempre es un "`\r\n`" (independiente del SO), así podemos dividirlas en cabeceras individuales. El separador entre el nombre y el valor siempre es dos puntos seguido de un espacio "`:`". Eso quedó establecido en la especificación.

Así, si queremos obtener un objeto con pares nombre/valor, necesitamos tratarlas con un poco de JS.

Como esto (asumiendo que si dos cabeceras tienen el mismo nombre, entonces el último sobreescribe al primero):

```
let headers = xhr
 .getAllResponseHeaders()
 .split('\r\n')
 .reduce((result, current) => {
 let [name, value] = current.split(': ');
 result[name] = value;
 return result;
 }, {});
// headers['Content-Type'] = 'image/png'
```

## POST, Formularios

Para hacer una solicitud POST, podemos utilizar el objeto [FormData](#) nativo.

La sintaxis:

```
let formData = new FormData([form]); // crea un objeto, opcionalmente se completa con un <form>
formData.append(name, value); // añade un campo
```

Lo creamos, opcionalmente lleno desde un formulario, `append` (agrega) más campos si se necesitan, y entonces:

1. `xhr.open('POST', ...)` – se utiliza el método POST.
2. `xhr.send(formData)` para enviar el formulario al servidor.

Por ejemplo:

```
<form name="person">
 <input name="name" value="John">
 <input name="surname" value="Smith">
</form>

<script>
 // pre llenado del objeto FormData desde el formulario
 let formData = new FormData(document.forms.person);

 // agrega un campo más
 formData.append("middle", "Lee");

 // lo enviamos
 let xhr = new XMLHttpRequest();
 xhr.open("POST", "/article/xmlhttprequest/post/user");
 xhr.send(formData);

 xhr.onload = () => alert(xhr.response);
</script>
```

El formulario fue enviado con codificación `multipart/form-data`.

O, si nos gusta más JSON, entonces, un `JSON.stringify` y lo enviamos como un string.

Solo no te olvides de asignar la cabecera `Content-Type: application/json`, muchos frameworks del lado del servidor decodifican automáticamente JSON con este:

```
let xhr = new XMLHttpRequest();

let json = JSON.stringify({
 name: "John",
 surname: "Smith"
});

xhr.open("POST", '/submit')
xhr.setRequestHeader('Content-type', 'application/json; charset=utf-8');

xhr.send(json);
```

El método `.send(body)` es bastante omnívoro. Puede enviar casi cualquier `body`, incluyendo objetos `Blob` y `BufferSource`.

## Progreso de carga

El evento `progress` se dispara solo en la fase de descarga.

Esto es: si hacemos un `POST` de algo, `XMLHttpRequest` primero sube nuestros datos (el cuerpo de la respuesta), entonces descarga la respuesta.

Si estamos subiendo algo grande, entonces seguramente estaremos interesados en rastrear el progreso de nuestra carga. Pero `xhr.onprogress` no ayuda aquí.

Hay otro objeto, sin métodos, exclusivamente para rastrear los eventos de subida: `xhr.upload`.

Este genera eventos similares a `xhr`, pero `xhr.upload` se dispara solo en las subidas:

- `loadstart` – carga iniciada.
- `progress` – se dispara periódicamente durante la subida.
- `abort` – carga abortada.
- `error` – error no HTTP.
- `load` – carga finalizada con éxito.
- `timeout` – carga caducada (si la propiedad `timeout` está asignada).
- `loadend` – carga finalizada con éxito o error.

Ejemplos de manejadores:

```
xhr.upload.onprogress = function(event) {
 alert(`Uploaded ${event.loaded} of ${event.total} bytes`);
};

xhr.upload.onload = function() {
 alert(`Upload finished successfully.`);
};

xhr.upload.onerror = function() {
 alert(`Error durante la carga: ${xhr.status}`);
};
```

Aquí un ejemplo de la vida real: indicación del progreso de subida de un archivo:

```
<input type="file" onchange="upload(this.files[0])">

<script>
function upload(file) {
 let xhr = new XMLHttpRequest();

 // rastrea el progreso de la subida
 xhr.upload.onprogress = function(event) {
 console.log(`Uploaded ${event.loaded} of ${event.total}`);
 };

 // seguimiento completado: sea satisfactorio o no
 xhr.onloadend = function() {
 if (xhr.status == 200) {
 console.log("Logrado");
 } else {
 console.log("error " + this.status);
 }
 };

 xhr.open("POST", "/article/xmlhttprequest/post/upload");
 xhr.send(file);
}
</script>
```

## Solicitudes de origen cruzado (Cross-origin)

`XMLHttpRequest` puede hacer solicitudes de origen cruzado, utilizando la misma política CORS que se [solicita](#).

Tal como `fetch`, no envía cookies ni autorización HTTP a otro origen por omisión. Para activarlas, asigna `xhr.withCredentials` como `true`:

```
let xhr = new XMLHttpRequest();
xhr.withCredentials = true;

xhr.open('POST', 'http://anywhere.com/request');
...
```

Ve el capítulo [Fetch: Cross-Origin Requests](#) para detalles sobre las cabeceras de origen cruzado.

## Resumen

Codificación típica de la solicitud GET con XMLHttpRequest :

```
let xhr = new XMLHttpRequest();

xhr.open('GET', '/my/url');

xhr.send();

xhr.onload = function() {
 if (xhr.status != 200) { // error HTTP?
 // maneja el error
 alert('Error: ' + xhr.status);
 return;
 }

 // obtiene la respuesta de xhr.response
};

xhr.onprogress = function(event) {
 // reporta progreso
 alert(`Loaded ${event.loaded} of ${event.total}`);
};

xhr.onerror = function() {
 // manejo de un error no HTTP (ej. red caída)
};
```

De hecho hay más eventos, la [especificación moderna](#) ↗ los lista (en el orden del ciclo de vida):

- `loadstart` – la solicitud ha comenzado.
- `progress` – un paquete de datos de la respuesta ha llegado, el cuerpo completo de la respuesta al momento está en `response`.
- `abort` – la solicitud ha sido cancelada por la llamada de `xhr.abort()`.
- `error` – un error de conexión ha ocurrido, ej. nombre de dominio incorrecto. No pasa con errores HTTP como 404.
- `load` – la solicitud se ha completado satisfactoriamente.
- `timeout` – la solicitud fue cancelada debido a que caducó (solo pasa si fue configurado).
- `loadend` – se dispara después de `load`, `error`, `timeout` o `abort`.

Los eventos `error`, `abort`, `timeout`, y `load` son mutuamente exclusivos. Solo uno de ellos puede pasar.

Los eventos más usados son la carga terminada (`load`), falla de carga (`error`), o podemos usar un solo manejador `loadend` y comprobar las propiedades del objeto solicitado `xhr` para ver qué ha pasado.

Ya hemos visto otro evento: `readystatechange`. Históricamente, apareció hace mucho tiempo, antes de que la especificación fuera publicada. Hoy en día no es necesario usarlo; podemos reemplazarlo con eventos más nuevos, pero puede ser encontrado a menudo en scripts viejos.

Si necesitamos rastrear específicamente, entonces debemos escuchar a los mismos eventos en el objeto `xhr.upload`.

## Carga de archivos reanudable

Con el método `fetch` es bastante fácil cargar un archivo.

¿Cómo reanudar la carga de un archivo después de perder la conexión? No hay una opción incorporada para eso, pero tenemos las piezas para implementarlo.

Las cargas reanudables deberían venir con indicación de progreso, ya que esperamos archivos grandes (Si necesitamos reanudar). Entonces, ya que `fetch` no permite rastrear el progreso de carga, usaremos [XMLHttpRequest](#).

## Evento de progreso poco útil

Para reanudar la carga, necesitamos saber cuánto fue cargado hasta la pérdida de la conexión.

Disponemos de `xhr.upload.onprogress` para rastrear el progreso de carga.

Desafortunadamente, esto no nos ayudará a reanudar la descarga. Ya que se origina cuando los datos son enviados, ¿pero fue recibida por el servidor? el navegador no lo sabe.

Tal vez fue almacenada por un proxy de la red local, o quizás el proceso del servidor remoto solo murió y no pudo procesarla, o solo se perdió en el medio y no alcanzó al receptor.

Es por eso que este evento solo es útil para mostrar una barra de progreso bonita.

Para reanudar una carga, necesitamos saber *exactamente* el número de bytes recibidos por el servidor. Y eso solo lo sabe el servidor, por lo tanto haremos una solicitud adicional.

## Algoritmos

1. Primero, crear un archivo id, para únicamente identificar el archivo que vamos a subir:

```
let fileId = file.name + '-' + file.size + '-' + file.lastModified;
```

Eso es necesario para reanudar la carga, para decirle al servidor lo que estamos reanudando.

Si el nombre o tamaño de la última fecha de modificación cambia, entonces habrá otro `fileId`.

2. Envía una solicitud al servidor, preguntando cuántos bytes tiene, así:

```
let response = await fetch('status', {
 headers: {
 'X-File-Id': fileId
 }
});

// El servidor tiene tanta cantidad de bytes
let startByte = +await response.text();
```

Esto asume que el servidor rastrea archivos cargados por el encabezado `X-File-Id`. Debe ser implementado en el lado del servidor.

Si el archivo no existe aún en el servidor, entonces su respuesta debe ser `0`.

3. Entonces, podemos usar el método `Blob slice` para enviar el archivo desde `startByte`:

```
xhr.open("POST", "upload");

// Archivo, de modo que el servidor sepa qué archivo subimos
xhr.setRequestHeader('X-File-Id', fileId);

// El byte desde el que estamos reanudando, así el servidor sabe que estamos reanudando
xhr.setRequestHeader('X-Start-Byte', startByte);

xhr.upload.onprogress = (e) => {
 console.log(`Uploaded ${startByte + e.loaded} of ${startByte + e.total}`);
};

// El archivo puede ser de input.files[0] u otra fuente
xhr.send(file.slice(startByte));
```

Aquí enviamos al servidor ambos archivos id como `X-File-Id`, para que de esa manera sepa que archivos estamos cargando, y el byte inicial como `X-Start-Byte`, para que sepa que no lo estamos cargando inicialmente, si no reanudándolo.

El servidor debe verificar sus registros, y si hubo una carga de ese archivo, y si el tamaño de carga actual es exactamente `X-Start-Byte`, entonces agregarle los datos.

Aquí está la demostración con el código tanto del cliente como del servidor, escrito en Node.js.

Este funciona solo parcialmente en este sitio, ya que Node.js está detrás de otro servidor llamado Nginx, que almacena cargas, pasándolas a Node.js cuando está completamente lleno.

Pero puedes cargarlo y ejecutarlo localmente para la demostración completa:

[https://plnkr.co/edit/jAV1UjbuOz4ZJSZr?p=preview ↗](https://plnkr.co/edit/jAV1UjbuOz4ZJSZr?p=preview)

Como podemos ver, los métodos de red modernos están cerca de los gestores de archivos en sus capacidades – control sobre header, indicador de progreso, enviar partes de archivos, etc.

Podemos implementar la carga reanudable y mucho mas.

## Sondeo largo

El "sondeo largo" es la forma más sencilla de tener una conexión persistente con el servidor. No utiliza ningún protocolo específico como "WebSocket" o "SSE".

Es muy fácil de implementar, y es suficientemente bueno en muchos casos.

## Sondeo regular

La forma más sencilla de obtener información nueva desde el servidor es un sondeo periódico. Es decir, solicitudes regulares al servidor: "Hola, estoy aquí, ¿tienes información para mí?". Por ejemplo, una vez cada 10 segundos.

En respuesta, el servidor primero se da cuenta de que el cliente está en línea, y segundo, envía un paquete con los mensajes que recibió hasta ese momento.

Esto funciona, pero tiene sus desventajas:

1. Los mensajes desde el servidor se transmiten con un retraso de hasta 10 segundos (el tiempo entre solicitudes de nuestro ejemplo).
  2. El servidor es bombardeado con solicitudes cada 10 segundos aunque no haya mensajes, incluso si el usuario cambió a otro lugar o está dormido. En términos de rendimiento, esto es bastante difícil de manejar.

Entonces: si hablamos de un servicio muy pequeño, este enfoque es viable. Pero en general, se necesita algo mejor.

## Sondeo largo

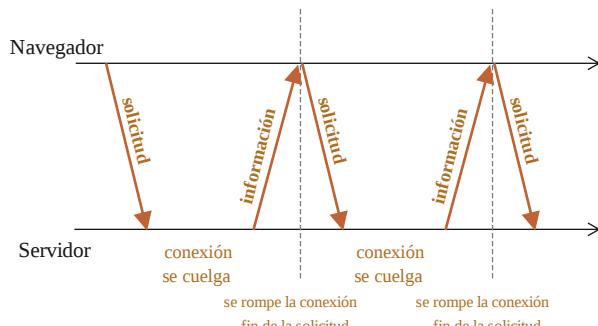
El llamado “sondeo largo” es una forma mucho mejor de sondear el servidor.

También es muy fácil de implementar, y envía los mensajes sin demoras.

El flujo es:

1. El navegador envía una solicitud al servidor.
  2. El servidor mantiene la conexión abierta mientras no tenga mensajes para enviar.
  3. Cuando aparece un mensaje, el servidor responde a la solicitud con dicho mensaje y cierra la conexión.
  4. El navegador recibe el mensaje y de inmediato realiza una nueva solicitud.

Esta situación, en la que el navegador envió una solicitud y se mantiene abierta una conexión con el servidor, es estándar para este método. En cuanto se entrega un mensaje, la conexión se cierra y restablece.



Si se pierde la conexión (debido a un error de red, por ejemplo), el navegador envía inmediatamente una nueva solicitud.

Este es el esquema, del lado del cliente, de una función de suscripción que realiza solicitudes largas:

```
async function subscribe() {
 let response = await fetch("/subscribe");
```

```

if (response.status == 502) {
 // El estado 502 es un error de "tiempo de espera agotado" en la conexión,
 // puede suceder cuando la conexión estuvo pendiente durante demasiado tiempo,
 // y el servidor remoto o un proxy la cerró
 // vamos a reconectarnos
 subscribe();
} else if (response.status != 200) {
 // Un error : vamos a mostrarlo
 showMessage(response.statusText);
 // Vuelve a conectar en un segundo
 await new Promise(resolve => setTimeout(resolve, 1000));
 subscribe();
} else {
 // Recibe y muestra el mensaje
 let message = await response.text();
 showMessage(message);
 // Llama a subscribe () nuevamente para obtener el siguiente mensaje
 subscribe();
}
}

subscribe();

```

Como puedes ver, la función `subscribe` realiza una búsqueda, espera la respuesta, la maneja, y se llama a sí misma nuevamente.

#### **El servidor debe ser capaz de mantener muchas conexiones pendientes**

La arquitectura del servidor debe poder funcionar bien con muchas conexiones pendientes.

Algunas arquitecturas de servidor ejecutan un proceso por conexión, resultando en que habrá tantos procesos como conexiones, y cada proceso requiere bastante memoria. Demasiadas conexiones la consumirán toda.

Este suele ser el caso de los backends escritos en lenguajes como PHP y Ruby.

Los servidores escritos con Node.js generalmente no tienen este problema.

Dicho esto, no es un problema del lenguaje sino de la implementación. La mayoría de los lenguajes modernos, incluyendo PHP y Ruby, permiten la implementación de un backend adecuado. Por favor, asegúrate de que la arquitectura del servidor funcione bien con múltiples conexiones simultáneas.

## Demostración: un chat

Este es un chat de demostración, que también puedes descargar y ejecutar localmente (si estás familiarizado con Node.js y puedes instalar módulos):

<https://plnkr.co/edit/BN2crU323159RXIJ?p=preview>

El código del navegador está en `browser.js`.

## Área de uso

El sondeo largo funciona muy bien en situaciones en las que los mensajes son escasos.

Pero si los mensajes llegan con mucha frecuencia, entonces el gráfico de arriba, mensajes solicitados/recibidos, se vuelve en forma de “diente de sierra”.

Cada mensaje es una solicitud separada: provista de encabezados, sobrecarga de autenticación, etc.

En este caso se prefieren otros métodos, como [WebSocket](#), o [SSE](#) (Eventos enviados por el servidor).

## WebSocket

El protocolo [WebSocket](#), descrito en la especificación [RFC 6455](#), brinda una forma de intercambiar datos entre el navegador y el servidor por medio de una conexión persistente. Los datos pueden ser pasados en ambas direcciones como paquetes “packets”, sin cortar la conexión y sin pedidos adicionales de HTTP “HTTP-requests”.

WebSocket es especialmente bueno para servicios que requieren intercambio de información continua, por ejemplo juegos en línea, sistemas de negocios en tiempo real, entre otros.

## Un ejemplo simple

Para abrir una conexión websocket, necesitamos crearla `new WebSocket` usando el protocolo especial `ws` en la url:

```
let socket = new WebSocket("ws://javascript.info");
```

También hay una versión encriptada `wss://`. Equivale al HTTPS para los websockets.

### Siempre dé preferencia a `wss://`

El protocolo `wss://` no solamente está encriptado, también es más confiable.

Esto es porque los datos en `ws://` no están encriptados y son visibles para cualquier intermediario. Entonces los servidores proxy viejos que no reconocen el protocolo WebSocket podrían interpretar los datos como cabeceras "extrañas" y abortar la conexión.

En cambio `wss://` es WebSocket sobre TLS (al igual que HTTPS es HTTP sobre TLS), la seguridad de la capa de transporte encripta los datos en el envío y los desencripta en el destino. Así, los paquetes de datos pasan encriptados a través de los proxy, estos servidores no pueden ver lo que hay dentro y los dejan pasar.

Una vez que el socket es creado, debemos escuchar los eventos que ocurren en él. Hay en total 4 eventos:

- `open` – conexión establecida,
- `message` – datos recibidos,
- `error` – error en websocket,
- `close` – conexión cerrada.

...Y si queremos enviar algo, `socket.send(data)` lo hará.

Aquí un ejemplo:

```
let socket = new WebSocket("wss://javascript.info/article/websocket/demo/hello");

socket.onopen = function(e) {
 alert("[open] Conexión establecida");
 alert("Enviando al servidor");
 socket.send("Mi nombre es John");
};

socket.onmessage = function(event) {
 alert(`[message] Datos recibidos del servidor: ${event.data}`);
};

socket.onclose = function(event) {
 if (event.wasClean) {
 alert(`[close] Conexión cerrada limpiamente, código=${event.code} motivo=${event.reason}`);
 } else {
 // ej. El proceso del servidor se detuvo o la red está caída
 // event.code es usualmente 1006 en este caso
 alert('[close] La conexión se cayó');
 }
};

socket.onerror = function(error) {
 alert(`[error]`);
};
```

Para propósitos de demostración, tenemos un pequeño servidor [server.js](#), escrito en Node.js, ejecutándose para el ejemplo de arriba. Este responde con "Hello from server, John", espera 5 segundos, y cierra la conexión.

Entonces verás los eventos `open` → `message` → `close`.

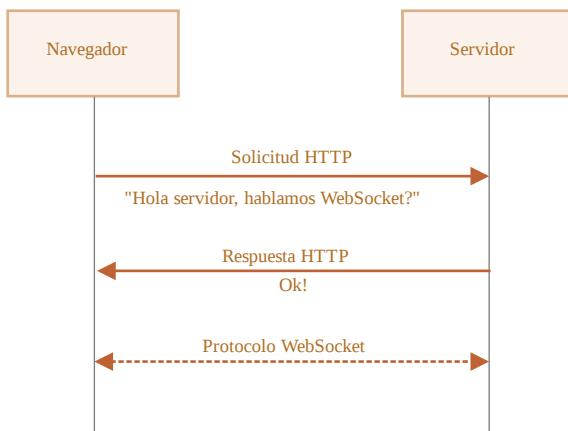
Eso es realmente todo, ya podemos conversar con WebSocket. Bastante simple, ¿no es cierto?

Ahora hablemos más en profundidad.

## Abriendo un websocket

Cuando se crea `new WebSocket(url)`, comienza la conexión de inmediato.

Durante la conexión, el navegador (usando cabeceras o "header") le pregunta al servidor: "¿Soportas Websockets?" y si si el servidor responde "Sí", la comunicación continúa en el protocolo WebSocket, que no es HTTP en absoluto.



Aquí hay un ejemplo de cabeceras de navegador para una petición hecha por `new WebSocket("wss://javascript.info/chat")`.

```
GET /chat
Host: javascript.info
Origin: https://javascript.info
Connection: Upgrade
Upgrade: websocket
Sec-WebSocket-Key: Iv8io/9s+lYFgZWcXczP8Q==
Sec-WebSocket-Version: 13
```

- `Origin` – La página de origen del cliente, ej. `https://javascript.info`. Los objetos WebSocket son cross-origin por naturaleza. No existen las cabeceras especiales ni otras limitaciones. De cualquier manera los servidores viejos son incapaces de manejar WebSocket, así que no hay problemas de compatibilidad. Pero la cabecera `Origin` es importante, pues habilita al servidor decidir si permite o no la comunicación WebSocket con el sitio web.
- `Connection: Upgrade` – señala que el cliente quiere cambiar el protocolo.
- `Upgrade: websocket` – el protocolo requerido es "websocket".
- `Sec-WebSocket-Key` – una clave de aleatoriedad generada por el navegador, usada para asegurar que el servidor soporta el protocolo WebSocket. Es aleatoriedad para evitar que servidores proxy almacenen en caché la comunicación que sigue.
- `Sec-WebSocket-Version` – Versión del protocolo WebSocket, 13 es la actual.

#### **i El intercambio WebSocket no puede ser emulado**

No podemos usar `XMLHttpRequest` o `fetch` para hacer este tipo de peticiones HTTP, porque JavaScript no tiene permitido establecer esas cabeceras.

Si el servidor concede el cambio a WebSocket, envía como respuesta el código 101:

```
101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: hsBlbuDTkk24srzE0TBULZAlC2g=
```

Aquí `Sec-WebSocket-Accept` es `Sec-WebSocket-Key`, recodificado usando un algoritmo especial. Al verlo, el navegador entiende que el servidor realmente soporta el protocolo WebSocket.

A continuación los datos son transferidos usando el protocolo WebSocket. Pronto veremos su estructura ("frames", marcos o cuadros en español). Y no es HTTP en absoluto.

#### **Extensiones y subprotocolos**

Puede tener las cabeceras adicionales `Sec-WebSocket-Extensions` y `Sec-WebSocket-Protocol` que describen extensiones y subprotocolos.

Por ejemplo:

- Sec-WebSocket-Extensions: deflate-frame significa que el navegador soporta compresión de datos. Una extensión es algo relacionado a la transferencia de datos, funcionalidad que extiende el protocolo WebSocket. La cabecera Sec-WebSocket-Extensions es enviada automáticamente por el navegador, con la lista de todas las extensiones que soporta.
- Sec-WebSocket-Protocol: soap, wamp significa que queremos transferir no cualquier dato, sino datos en protocolos [SOAP](#) o WAMP ("The WebSocket Application Messaging Protocol"). Los subprotocolos de WebSocket están registrados en el [catálogo IANA](#). Entonces, esta cabecera describe los formatos de datos que vamos a usar.

Esta cabecera opcional se establece usando el segundo parámetro de `new WebSocket`, que es el array de subprotocolos. Por ejemplo, si queremos usar SOAP o WAMP:

```
let socket = new WebSocket("wss://javascript.info/chat", ["soap", "wamp"]);
```

El servidor debería responder con una lista de protocolos o extensiones que acepta usar.

Por ejemplo, la petición:

```
GET /chat
Host: javascript.info
Upgrade: websocket
Connection: Upgrade
Origin: https://javascript.info
Sec-WebSocket-Key: Iv8io/9s+lYFgZWcXczP8Q==
Sec-WebSocket-Version: 13
Sec-WebSocket-Extensions: deflate-frame
Sec-WebSocket-Protocol: soap, wamp
```

Respuesta:

```
101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: hsBlbuDTkk24srzE0TBULZAlC2g=
Sec-WebSocket-Extensions: deflate-frame
Sec-WebSocket-Protocol: soap
```

Aquí el servidor responde que soporta la extensión "deflate-frame", y únicamente SOAP de los subprotocolos solicitados.

## Transferencia de datos

La comunicación WebSocket consiste de "frames" (cuadros) de fragmentos de datos, que pueden ser enviados de ambos lados y pueden ser de varias clases:

- "text frames" – contiene datos de texto que las partes se mandan entre sí.
- "binary data frames" – contiene datos binarios que las partes se mandan entre sí.
- "ping/pong frames" son usados para testear la conexión; enviados desde el servidor, el navegador responde automáticamente.
- También existe "connection close frame", y algunos otros frames de servicio.

En el navegador, trabajamos directamente solamente con frames de texto y binarios.

**El método `WebSocket .send()` puede enviar tanto datos de texto como binarios.**

Una llamada `socket.send(body)` permite en `body` datos en formato string o binarios, incluyendo `Blob`, `ArrayBuffer`, etc. No se requiere configuración: simplemente se envían en cualquier formato.

**Cuando recibimos datos, el texto siempre viene como string. Y para datos binarios, podemos elegir entre los formatos `Blob` y `ArrayBuffer`.**

Esto se establece en la propiedad `socket.binaryType`, que es "blob" por defecto y entonces los datos binarios vienen como objetos `Blob`.

`Blob` es un objeto binario de alto nivel que se integra directamente con `<a>`, `<img>` y otras etiquetas, así que es una opción predeterminada saludable. Pero para procesamiento binario, para acceder a bytes individuales, podemos cambiarlo a `"arraybuffer"`:

```
socket.binaryType = "arraybuffer";
socket.onmessage = (event) => {
 // event.data puede ser string (si es texto) o arraybuffer (si es binario)
};
```

## Limitaciones de velocidad

Supongamos que nuestra app está generando un montón de datos para enviar. Pero el usuario tiene una conexión de red lenta, posiblemente internet móvil fuera de la ciudad.

Podemos llamar `socket.send(data)` una y otra vez. Pero los datos serán acumulados en memoria (en un “buffer”) y enviados solamente tan rápido como la velocidad de la red lo permita.

La propiedad `socket.bufferedAmount` registra cuántos bytes quedan almacenados (“buffered”) hasta el momento esperando a ser enviados a la red.

Podemos examinarla para ver si el “socket” está disponible para transmitir.

```
// examina el socket cada 100ms y envía más datos
// solamente si todos los datos existentes ya fueron enviados
setInterval(() => {
 if (socket.bufferedAmount == 0) {
 socket.send(moreData());
 }
}, 100);
```

## Cierre de conexión

Normalmente, cuando una parte quiere cerrar la conexión (servidor o navegador, ambos tienen el mismo derecho), envía un “frame de cierre de conexión” con un código numérico y un texto con el motivo.

El método para eso es:

```
socket.close([code], [reason]);
```

- `code` es un código especial de cierre de WebSocket (opcional)
- `reason` es un string que describe el motivo de cierre (opcional)

Entonces el manejador del evento `close` de la otra parte obtiene el código y el motivo, por ejemplo:

```
// la parte que hace el cierre:
socket.close(1000, "Work complete");

// la otra parte:
socket.onclose = event => {
 // event.code === 1000
 // event.reason === "Work complete"
 // event.wasClean === true (clean close)
};
```

Los códigos más comunes:

- `1000` – cierre normal. Es el predeterminado (usado si no se proporciona `code`),
- `1006` – no hay forma de establecerlo manualmente, indica que la conexión se perdió (no hay frame de cierre).

Hay otros códigos como:

- `1001` – una parte se va, por ejemplo el server se está apagando, o el navegador deja la página,
- `1009` – el mensaje es demasiado grande para procesar,
- `1011` – error inesperado en el servidor,

- ...y así.

La lista completa puede encontrarse en [RFC6455, §7.4.1](#).

Los códigos de WebSocket son como los que hay de HTTP, pero diferentes. En particular, los códigos menores a 1000 son reservados, habrá un error si tratamos de establecerlos.

```
// en caso de conexión que se rompe
socket.onclose = event => {
 // event.code === 1006
 // event.reason === ""
 // event.wasClean === false (no hay un frame de cierre)
};
```

## Estado de la conexión

Para obtener el estado (state) de la conexión, tenemos la propiedad `socket.readyState` con valores:

- 0 – “CONNECTING”: la conexión aún no fue establecida,
- 1 – “OPEN”: comunicando,
- 2 – “CLOSING”: la conexión se está cerrando,
- 3 – “CLOSED”: la conexión está cerrada.

## Ejemplo Chat

Revisemos un ejemplo de chat usando la API WebSocket del navegador y el módulo WebSocket de Node.js <https://github.com/websockets/ws>. Prestaremos atención al lado del cliente, pero el servidor es igual de simple.

HTML: necesitamos un `<form>` para enviar mensajes y un `<div>` para los mensajes entrantes:

```
<!-- message form -->
<form name="publish">
 <input type="text" name="message">
 <input type="submit" value="Send">
</form>

<!-- div with messages -->
<div id="messages"></div>
```

De JavaScript queremos tres cosas:

1. Abrir la conexión.
2. Ante el “submit” del form, enviar `socket.send(message)` el mensaje.
3. Al llegar un mensaje, agregarlo a `div#messages`.

Aquí el código:

```
let socket = new WebSocket("wss://javascript.info/article/websocket/chat/ws");

// enviar el mensaje del form
document.forms.publish.onSubmit = function() {
 let outgoingMessage = this.message.value;

 socket.send(outgoingMessage);
 return false;
};

// mensaje recibido - muestra el mensaje en div#messages
socket.onmessage = function(event) {
 let message = event.data;

 let messageElem = document.createElement('div');
 messageElem.textContent = message;
 document.getElementById('messages').prepend(messageElem);
}
```

El código de servidor está fuera de nuestro objetivo. Aquí usaremos Node.js, pero no necesitas hacerlo. Otras plataformas también tienen sus formas de trabajar con WebSocket.

El algoritmo de lado de servidor será:

1. Crear `clients = new Set()` – un conjunto de sockets.
2. Para cada websocket aceptado, sumarlo al conjunto `clients.add(socket)` y establecer un “event listener” `message` para obtener sus mensajes.
3. Cuando un mensaje es recibido: iterar sobre los clientes y enviarlo a todos ellos.
4. Cuando una conexión se cierra: `clients.delete(socket)`.

```
const ws = new require('ws');
const wss = new ws.Server({noServer: true});

const clients = new Set();

http.createServer((req, res) => {
 // aquí solo manejamos conexiones websocket
 // en proyectos reales tendremos también algún código para manejar peticiones no websocket
 wss.handleUpgrade(req, req.socket, Buffer.alloc(0), onSocketConnect);
});

function onSocketConnect(ws) {
 clients.add(ws);

 ws.on('message', function(message) {
 message = message.slice(0, 50); // la longitud máxima del mensaje será 50

 for(let client of clients) {
 client.send(message);
 }
 });

 ws.on('close', function() {
 clients.delete(ws);
 });
}
```

Aquí está el ejemplo funcionando:

Puedes descargarlo (botón arriba/derecha en el iframe) y ejecutarlo localmente. No olvides instalar [Node.js](#) y `npm install ws` antes de hacerlo.

## Resumen

WebSocket es la forma moderna de tener conexiones persistentes entre navegador y servidor .

- Los WebSockets no tienen limitaciones “cross-origin”.
- Están muy bien soportados en los navegadores.
- Pueden enviar y recibir datos string y binarios.

La API es simple.

Métodos:

- `socket.send(data)`,
- `socket.close([code], [reason])`.

Eventos:

- `open`,
- `message`,
- `error`,
- `close`.

El WebSocket por sí mismo no incluye reconexión, autenticación ni otros mecanismos de alto nivel. Hay librerías cliente/servidor para eso, y también es posible implementar esas capacidades manualmente.

A veces, para integrar WebSocket a un proyecto existente, se ejecuta un servidor WebSocket en paralelo con el servidor HTTP principal compartiendo la misma base de datos. Las peticiones a WebSocket usan `wss://ws.site.com`, un subdominio que se dirige al servidor de WebSocket mientras que `https://site.com` va al servidor HTTP principal.

Seguro, otras formas de integración también son posibles.

## Eventos enviados por el servidor

La especificación de los [Eventos enviados por el servidor](#) describe una clase incorporada `EventSource`, que mantiene la conexión con el servidor y permite recibir eventos de él.

Similar a `WebSocket`, la conexión es persistente.

Pero existen varias diferencias importantes:

WebSocket	EventSource
Bidireccional: tanto el cliente como el servidor pueden intercambiar mensajes	Unidireccional: solo el servidor envía datos
Datos binarios y de texto	Solo texto
Protocolo WebSocket	HTTP regular

`EventSource` es una forma menos poderosa de comunicarse con el servidor que `WebSocket`.

¿Por qué debería uno usarlo?

El motivo principal: es más sencillo. En muchas aplicaciones, el poder de `WebSocket` es demasiado.

Necesitamos recibir un flujo de datos del servidor: tal vez mensajes de chat o precios de mercado, o lo que sea. Para eso es bueno `EventSource`. También admite la reconexión automática, algo que debemos implementar manualmente con `WebSocket`. Además, es HTTP común, no un protocolo nuevo.

## Recibir mensajes

Para comenzar a recibir mensajes, solo necesitamos crear un `new EventSource(url)`.

El navegador se conectará a la `url` y mantendrá la conexión abierta, esperando eventos.

El servidor debe responder con el estado 200 y el encabezado `Content-Type:text/event-stream`, entonces mantener la conexión y escribir mensajes en el formato especial, así:

```
data: Mensaje 1
data: Mensaje 2
data: Mensaje 3
data: de dos líneas
```

- Un mensaje de texto va después de `data:`, el espacio después de los dos puntos es opcional.
- Los mensajes están delimitados con saltos de línea dobles `\n\n`.
- Para enviar un salto de línea `\n`, podemos enviar inmediatamente un `data:` (tercer mensaje arriba) más.

En la práctica, los mensajes complejos generalmente se envían codificados en JSON. Los saltos de línea están codificados así `\n` dentro de los mensajes, por lo que los mensajes `data:` multilínea no son necesarios.

Por ejemplo:

```
data: {"user":"John", "message":"Primera línea\n Segunda línea"}
```

... Entonces podemos asumir que un `data:` contiene exactamente un mensaje.

Para cada uno de estos mensajes, se genera el evento `message`:

```
let eventSource = new EventSource("/events/subscribe");

eventSource.onmessage = function(event) {
 console.log("Nuevo mensaje", event.data);
 // registrará apuntes 3 veces para el flujo de datos anterior
};

// o eventSource.addEventListener('message', ...)
```

## Solicitudes Cross-origin

`EventSource` admite solicitudes cross-origin, como `fetch` o cualquier otro método de red. Podemos utilizar cualquier URL:

```
let source = new EventSource("https://another-site.com/events");
```

El servidor remoto obtendrá el encabezado `Origin` y debe responder con `Access-Control-Allow-Origin` para continuar.

Para pasar las credenciales, debemos configurar la opción adicional `withCredentials`, así:

```
let source = new EventSource("https://another-site.com/events", {
 withCredentials: true
});
```

Consulte el capítulo [Fetch: Cross-Origin Requests](#) para obtener más detalles sobre los encabezados cross-origin.

## Reconexión

Tras la creación con `new EventSource`, el cliente se conecta al servidor y, si la conexión se interrumpe, se vuelve a conectar.

Eso es muy conveniente, ya que no tenemos que preocuparnos por eso.

Hay un pequeño retraso entre las reconexiones, unos segundos por defecto.

El servidor puede establecer la demora recomendada usando `retry`: dentro de la respuesta (en milisegundos):

```
retry: 15000
data: Hola, configuré el retraso de reconexión en 15 segundos
```

El `retry`: puede venir junto con algunos datos, o como un mensaje independiente.

El navegador debe esperar esa cantidad de milisegundos antes de volver a conectarse. O más, por ejemplo: si el navegador sabe (desde el sistema operativo) que no hay conexión de red en este momento, puede esperar hasta que aparezca la conexión y luego volver a intentarlo.

- Si el servidor desea que el navegador deje de volver a conectarse, debería responder con el estado HTTP 204.
- Si el navegador quiere cerrar la conexión, debe llamar a `eventSource.close()`:

```
let eventSource = new EventSource(...);

eventSource.close();
```

Además, no habrá reconexión si la respuesta tiene un `Content-Type` incorrecto o su estado HTTP difiere de 301, 307, 200 y 204. En tales casos, se emitirá el evento `"error"` y el navegador no se volverá a conectar.

### Por favor tome nota:

Cuando una conexión finalmente se cierra, no hay forma de "reabrirla". Si queremos conectarnos de nuevo, simplemente crea un nuevo `EventSource`.

## ID del mensaje

Cuando una conexión se interrumpe debido a problemas de red, ninguna de las partes puede estar segura de qué mensajes se recibieron y cuáles no.

Para reanudar correctamente la conexión, cada mensaje debe tener un campo `id`, así:

```
data: Mensaje 1
id: 1

data: Mensaje 2
id: 2

data: Mensaje 3
data: de dos líneas
id: 3
```

Cuando se recibe un mensaje con `id`, el navegador:

- Establece la propiedad `eventSource.lastEventId` a su valor.
- Tras la reconexión, el navegador envía el encabezado `Last-Event-ID` con ese `id`, para que el servidor pueda volver a enviar los siguientes mensajes.

**i** **Pon `id`: después de `data`:**

Ten en cuenta: el `id` es adjuntado debajo del mensaje `data` por el servidor, para garantizar que `lastEventId` se actualice después de recibir el mensaje.

## Estado de conexión: `readyState`

El objeto `EventSource` tiene la propiedad `readyState`, que tiene uno de tres valores:

```
EventSource.CONNECTING = 0; // conectando o reconectando
EventSource.OPEN = 1; // conectado
EventSource.CLOSED = 2; // conexión cerrada
```

Cuando se crea un objeto, o la conexión no funciona, siempre es `EventSource.CONNECTING` (es igual a `0`).

Podemos consultar esta propiedad para conocer el estado de `EventSource`.

## Tipos de eventos

Por defecto, el objeto `EventSource` genera tres eventos:

- `message` – un mensaje recibido, disponible como `event.data`.
- `open` – la conexión está abierta.
- `error` – no se pudo establecer la conexión, por ejemplo, el servidor devolvió el estado HTTP 500.

El servidor puede especificar otro tipo de evento con `event: ...` al inicio del evento.

Por ejemplo:

```
event: join
data: Bob

data: Hola

event: leave
data: Bob
```

Para manejar eventos personalizados, debemos usar `addEventListener`, no `onmessage`:

```
eventSource.addEventListener('join', event => {
 alert(`Se unió ${event.data}`);
});
```

```

eventSource.addEventListener('message', event => {
 alert(`Dijo: ${event.data}`);
});

eventSource.addEventListener('leave', event => {
 alert(`Salió ${event.data}`);
});

```

## Ejemplo completo

Aquí está el servidor que envía mensajes con 1, 2, 3, luego bye y cierra la conexión.

Luego, el navegador se vuelve a conectar automáticamente.

<https://plnkr.co/edit/5hQ7AiSlf3HVWW2F?p=preview>

## Resumen

El objeto `EventSource` establece automáticamente una conexión persistente y permite al servidor enviar mensajes a través de él.

Ofrece:

- Reconexión automática, con tiempo de espera de `reintento` ajustable.
- IDs en cada mensaje para reanudar los eventos, el último identificador recibido se envía en el encabezado `Last-Event-ID` al volver a conectarse.
- El estado actual está en la propiedad `readyState`.

Eso hace que `EventSource` sea una alternativa viable a `WebSocket`, ya que es de un nivel más bajo y carece de esas características integradas (aunque se pueden implementar).

En muchas aplicaciones de la vida real, el poder de `EventSource` es suficiente.

Compatible con todos los navegadores modernos (no IE).

La sintaxis es:

```
let source = new EventSource(url, [credentials]);
```

El segundo argumento tiene solo una opción posible: `{withCredentials: true}`, permite enviar credenciales de cross-origin.

La seguridad general de cross-origin es la misma que para `fetch` y otros métodos de red.

## Propiedades de un objeto `EventSource`

### `readyState`

El estado de conexión actual: `EventSource.CONNECTING` (=0), `EventSource.OPEN` (=1) o `EventSource.CLOSED` (=2).

### `lastEventId`

El último `id` recibido. Tras la reconexión, el navegador lo envía en el encabezado `Last-Event-ID`.

## Métodos

### `close()`

Cierra la conexión.

## Eventos

### `message`

Mensaje recibido, los datos están en `event.data`.

### `open`

Se establece la conexión.

## error

En caso de error, se incluyen tanto la pérdida de conexión (se reconectará automáticamente) como los errores fatales. Podemos comprobar `readyState` para ver si se está intentando la reconexión.

El servidor puede establecer un nombre de evento personalizado en `event:`. Tales eventos deben manejarse usando `addEventListener`, no `on<evento>`.

## Formato de respuesta del servidor

El servidor envía mensajes, delimitados por `\n\n`.

Un mensaje puede tener los siguientes campos:

- `data:` – cuerpo del mensaje, una secuencia de múltiples `datos` se interpreta como un solo mensaje, con `\n` entre las partes.
- `id:` – renueva `lastEventId`, enviado en el encabezado `Last-Event-ID` al volver a conectarse.
- `retry:` – recomienda una demora de reinicio para las reconexiones en milisegundos. No hay forma de configurarlo desde JavaScript.
- `event:` – nombre del evento, debe preceder a `data:`.

Un mensaje puede incluir uno o más campos en cualquier orden, pero `id:` suele ser el último.

## Almacenando datos en el navegador

### Cookies, document.cookie

Las cookies son pequeñas cadenas de datos que se almacenan directamente en el navegador. Son parte del protocolo HTTP, definido por la especificación [RFC 6265 ↗](#).

Las cookies son usualmente establecidos por un servidor web utilizando la cabecera de respuesta HTTP `Set-Cookie`. Entonces, el navegador los agrega automáticamente a (casi) toda solicitud del mismo dominio usando la cabecera HTTP `Cookie`.

Uno de los casos de uso más difundidos es la autenticación:

1. Al iniciar sesión, el servidor usa la cabecera HTTP `Set-Cookie` en respuesta para establecer una cookie con un "identificador de sesión" único.
2. Al enviar la siguiente solicitud al mismo dominio, el navegador envía la cookie usando la cabecera HTTP `Cookie`.
3. Así el servidor sabe quién hizo la solicitud.

También podemos acceder a las cookies desde el navegador usando la propiedad `document.cookie`.

Hay muchas cosas intrincadas acerca de las cookies y sus opciones. En este artículo las vamos a ver en detalle.

### Leyendo a document.cookie

Asumiendo que estás en un sitio web, es posible ver sus cookies así:

```
// En javascript.info, usamos Google Analytics para estadísticas,
// así que debería haber algunas cookies
alert(document.cookie); // cookie1=value1; cookie2=value2; ...
```

El valor de `document.cookie` consiste de pares `name=value` delimitados por `;`. Cada uno es una cookie separada.

Para encontrar una cookie particular, podemos separar (`split`) `document.cookie` por `;` y encontrar el nombre correcto. Podemos usar tanto una expresión regular como funciones de array para ello.

Lo dejamos como ejercicio para el lector. Al final del artículo encontrarás funciones de ayuda para manipular cookies.

### Escribiendo en document.cookie

Podemos escribir en `document.cookie`. Pero no es una propiedad de datos, es un [accessor \(getter/setter\)](#). Una asignación a él se trata especialmente.

**Una operación de escritura a `document.cookie` actualiza solo las cookies mencionadas en ella, y no toca las demás.**

Por ejemplo, este llamado establece una cookie con el nombre `user` y el valor `John`:

```
document.cookie = "user=John"; // modifica solo la cookie llamada 'user'
alert(document.cookie); // muestra todas las cookies
```

Si lo ejecutas, probablemente verás múltiples cookies. Esto es porque la operación `document.cookie=` no sobrescribe todas las cookies. Solo configura la cookie mencionada `user`.

Técnicamente, nombre y valor pueden tener cualquier carácter. Pero para mantener un formato válido, los caracteres especiales deben escaparse usando la función integrada `encodeURIComponent`:

```
// los caracteres especiales (espacios), necesitan codificarse
let name = "my name";
let value = "John Smith"

// codifica la cookie como my%20name=John%20Smith
document.cookie = encodeURIComponent(name) + '=' + encodeURIComponent(value);

alert(document.cookie); // ...; my%20name=John%20Smith
```

### ⚠ Limitaciones

Hay algunas limitaciones:

- El par `name=value`, después de `encodeURIComponent`, no debe exceder 4KB. Así que no podemos almacenar algo enorme en una cookie.
- La cantidad total de cookies por dominio está limitada a alrededor de 20+, el límite exacto depende del navegador.

Las cookies tienen varias opciones, muchas de ellas importantes y deberían ser configuradas.

Las opciones son listadas después de `key=value`, delimitadas por un `;`:

```
document.cookie = "user=John; path=/; expires=Tue, 19 Jan 2038 03:14:07 GMT"
```

## path

- `path=/mypath`

La ruta del prefijo `path` debe ser absoluto. Esto hace la cookie accesible a las páginas bajo esa ruta. La forma predeterminada es la ruta actual.

Si una cookie es establecida con `path=/admin`, será visible en las páginas `/admin` y `/admin/something`, pero no en `/home` o `/adminpage`.

Usualmente, debemos configurarlo en la raíz: `path=/` para hacer la cookie accesible a todas las páginas del sitio web.

## domain

- `domain=site.com`

Un dominio define dónde la cookie es accesible. Aunque en la práctica hay limitaciones y no podemos configurar cualquier dominio.

**No hay forma de hacer que una cookie sea accesible desde otro dominio de segundo nivel, entonces `other.com` nunca recibirá una cookie establecida en `site.com`.**

Es una restricción de seguridad, para permitirnos almacenar datos sensibles en cookies que deben estar disponibles para un único sitio solamente.

De forma predeterminada, una cookie solo es accesible en el dominio que la establece.

Pero por favor toma nota: de forma predeterminada una cookie tampoco es compartida por un subdominio, como `forum.site.com`.

```
// en site.com
document.cookie = "user=John"

// en forum.site.com
alert(document.cookie); // no user
```

...aunque esto puede cambiarse. Si queremos permitir que un subdominio como `forum.site.com` obtenga una cookie establecida por `site.com`, eso es posible.

Para ello, cuando establecemos una cookie en `site.com`, debemos configurar explícitamente la raíz del dominio en la opción `domain: domain=site.com`. Entonces todos los subdominios verán la cookie.

Por ejemplo:

```
// en site.com
// hacer la cookie accesible en cualquier subdominio *.site.com:
document.cookie = "user=John; domain=site.com"

// ...luego

// en forum.site.com
alert(document.cookie); // tiene la cookie user=John
```

Por razones históricas, `domain=.site.com` (con un punto antes de `site.com`) también funciona de la misma forma, permitiendo acceso a la cookie desde subdominios. Esto es una vieja notación y debe ser usada si necesitamos dar soporte a navegadores muy viejos.

Entonces: la opción `domain` permite que las cookies sean accesibles en los subdominios.

## expires, max-age

De forma predeterminada, si una cookie no tiene una de estas opciones, desaparece cuando el navegador se cierra. Tales cookies se denominan “cookies de sesión”.

Para que las cookies sobrevivan al cierre del navegador, podemos usar las opciones `expires` o `max-age`.

- `expires=Tue, 19 Jan 2038 03:14:07 GMT`

La fecha de expiración define el momento en que el navegador la borra automáticamente.

La fecha debe estar exactamente en ese formato, en el huso horario GMT. Podemos obtenerlo con `date.toUTCString()`. Por ejemplo, podemos configurar que la cookie expire en un día:

```
// +1 dia desde ahora
let date = new Date(Date.now() + 86400e3);
date = date.toUTCString();
document.cookie = "user=John; expires=" + date;
```

Si establecemos `expires` en una fecha en el pasado, la cookie es eliminada.

- `max-age=3600`

`max-age` es una alternativa a `expires`, y especifica la expiración de la cookie en segundos desde el momento actual.

Si la configuramos a cero o un valor negativo, la cookie es eliminada:

```
// la cookie morirá en +1 hora a partir de ahora
document.cookie = "user=John; max-age=3600";

// borra la cookie (la hacemos expirar ya)
document.cookie = "user=John; max-age=0";
```

## secure

- `secure`

La cookie debe ser transferida solamente a través de HTTPS.

De forma predeterminada, si establecemos una cookie en `http://site.com`, entonces también aparece en `https://site.com` y viceversa.

Esto es, las cookies están basadas en el dominio, no distinguen entre protocolos.

Con la opción `secure`, si una cookie se establece por `https://site.com`, entonces no aparecerá cuando el mismo sitio es accedido por HTTP, como `http://site.com`. Entonces, si una cookie tiene información sensible que nunca debe ser enviada sobre HTTP sin encriptar, debe configurarse `secure`.

```
// asumiendo que estamos en https:// ahora
// configuraremos la cookie para ser segura (solo accesible sobre HTTPS)
document.cookie = "user=John; secure";
```

## samesite

Este es otro atributo de seguridad. `samesite` sirve para proteger contra los ataques llamados XSRF (falsificación de solicitud entre sitios, cross-site request forgery).

Para entender cómo funciona y su utilidad, veamos primero los ataques XSRF.

### ataque XSRF

Imagina que tienes una sesión en el sitio `bank.com`. Esto es: tienes una cookie de autenticación para ese sitio. Tu navegador lo envía a `bank.com` en cada solicitud, así aquel te reconoce y ejecuta todas las operaciones financieras sensibles.

Ahora, mientras navegas la red en otra ventana, accidentalmente entras en otro sitio `evil.com`. Este sitio tiene código JavaScript que envía un formulario `<form action="https://bank.com/pay">` a `bank.com` con los campos que inician una transacción a la cuenta el hacker.

El navegador envía cookies cada vez que visitas el sitio `bank.com`, incluso si el form fue enviado desde `evil.com`. Entonces el banco te reconoce y realmente ejecuta el pago.



Ese es el ataque llamado “Cross-Site Request Forgery” (XSRF).

Los bancos reales están protegidos contra esto por supuesto. Todos los formularios generados por `bank.com` tienen un campo especial, llamado “token de protección XSRF”, que una página maliciosa no puede generar o extraer desde una página remota. Puede enviar el form, pero no obtiene respuesta a la solicitud. El sitio `bank.com` verifica tal token en cada form que recibe.

Tal protección toma tiempo para implementarla. Necesitamos asegurarnos de que cada form tiene dicho campo token, y debemos verificar todas las solicitudes.

### Opción de cookie `samesite`

La opción `samesite` brinda otra forma de proteger tales ataques, que (en teoría) no requiere el “token de protección XSRF”.

Tiene dos valores posibles:

- `samesite=strict` (lo mismo que `samesite` sin valor)

Una cookie con `samesite=strict` nunca es enviada si el usuario viene desde fuera del mismo sitio.

En otras palabras, si el usuario sigue un enlace desde su correo, envía un form desde `evil.com`, o hace cualquier operación originada desde otro dominio, la cookie no será enviada.

Si las cookies de autenticación tienen la opción `samesite`, un ataque XSRF no tiene posibilidad de éxito porque el envío de `evil.com` llega sin cookies. Así `bank.com` no reconoce el usuario y no procederá con el pago.

La protección es muy confiable. Solo las operaciones que vienen de `bank.com` enviarán la cookie `samesite`, por ejemplo un form desde otra página de `bank.com`.

Aunque hay un pequeño inconveniente.

Cuando el usuario sigue un enlace legítimo a `bank.com`, por ejemplo desde sus propio correo, será sorprendido con que `bank.com` no lo reconoce. Efectivamente, las cookies `samesite=strict` no son enviadas en ese caso.

Podemos sortear esto usando dos cookies: una para el “reconocimiento general”, con el solo propósito de decir: “Hola, John”, y la otra para operaciones de datos con `samesite=strict`. Entonces, la persona que venga desde fuera del sitio llega a la página de bienvenida, pero los pagos serán iniciados desde dentro del sitio web del banco, entonces la segunda cookie sí será enviada.

- `samesite=lax`

Un enfoque más laxo que también protege de ataques CSRF y no afecta la experiencia de usuario.

El modo `lax`, como `strict`, prohíbe al navegador enviar cookies cuando viene desde fuera del sitio, pero agrega una excepción.

Una cookie `samesite=lax` es enviada si se cumplen dos condiciones:

1. El método HTTP es seguro (por ejemplo GET, pero no POST).

La lista completa de métodos seguros HTTP está en la especificación [RFC7231](#). Básicamente son métodos usados para leer, pero no escribir datos. Los que no ejecutan ninguna operación de alteración de datos. Seguir un enlace es siempre GET, el método seguro.

2. La operación ejecuta una navegación del más alto nivel (cambia la URL en la barra de dirección del navegador).

Esto es usualmente verdad, pero si la navegación es ejecutada dentro de un `<iframe>`, entonces no es de alto nivel. Tampoco encajan aquí los métodos JavaScript para solicitudes de red porque no ejecutan ninguna navegación.

Entonces, lo que hace `samesite=lax` es básicamente permitir la operación más común “ir a URL” para obtener cookies. Por ejemplo, abrir un sitio desde la agenda satisface estas condiciones.

Pero cualquier cosa más complicada, como solicitudes de red desde otro sitio, o un “form submit”, pierde las cookies.

Si esto es adecuado para ti, entonces agregar `samesite=lax` probablemente no dañe la experiencia de usuario y agrega protección.

Por sobre todo, `samesite` es una excelente opción.

Tiene una importante debilidad:

- `samesite` es ignorado (no soportado) por navegadores viejos, de alrededor de 2017.

### **Así que si solo confiamos en `samesite` para brindar protección, habrá navegadores que serán vulnerables.**

Pero con seguridad podemos usar `samesite`, junto con otras medidas de protección como los tokens xsrf, para agregar una capa adicional de defensa. En el futuro probablemente podamos descartar la necesidad de tokens xsrf.

## **httpOnly**

Esta opción no tiene nada que ver con JavaScript, pero tenemos que mencionarla para completar la guía.

El servidor web usa la cabecera `Set-Cookie` para establecer la cookie. También puede configurar la opción `httpOnly`.

Esta opción impide a JavaScript el acceso a la cookie. No podemos ver ni manipular tal cookie usando `document.cookie`.

Esto es usado como medida de precaución, para proteger de ciertos ataques donde el hacker inyecta su propio código en una página y espera que el usuario visite esa página. Esto no debería ser posible en absoluto, los hackers bo deberían poder insertar su código en nuestro sitio, pero puede haber bugs que les permite hacerlo.

Normalmente, si eso sucede y el usuario visita una página web con el código JavaScript del hacker, entonces ese código se ejecuta y gana acceso a `document.cookie` con las cookies del usuario conteniendo información de autenticación. Eso es malo.

Pero si una cookie es `httpOnly`, `document.cookie` no la ve y está protegida.

## **Apéndice: Funciones de cookies**

Aquí hay un pequeño conjunto de funciones para trabajar con cookies, más conveniente que la modificación manual de `document.cookie`.

Existen muchas librerías de cookies para eso, así que estas son para demostración solamente. Aunque completamente funcionales.

## getCookie(name)

La forma más corta de acceder a una cookie es usar una expresión regular.

La función `getCookie(name)` devuelve la cookie con el nombre `name` dado:

```
// devuelve la cookie con el nombre dado,
// o undefined si no la encuentra
function getCookie(name) {
 let matches = document.cookie.match(new RegExp(
 "(?:^|;)"+name.replace(/([\.\$\?*\|\{\}\(\)\[\]\\\\\\\+^])/g, '\\$1')+"=([^;]*"
));
 return matches ? decodeURIComponent(matches[1]) : undefined;
}
```

Aquí `new RegExp` se genera dinámicamente para coincidir ; `name=<value>`.

Nota que el valor de una cookie está codificado, entonces `getCookie` usa una función integrada `decodeURIComponent` para decodificarla.

## setCookie(name, value, options)

Establece el nombre de la cookie `name` al valor dado `value`, con la ruta por defecto `path=/`, y puede ser modificada para agregar otros valores predeterminados:

```
function setCookie(name, value, options = {}) {

 options = {
 path: '/',
 // agregar otros valores predeterminados si es necesario
 ...options
 };

 if (options.expires instanceof Date) {
 options.expires = options.expires.toUTCString();
 }

 let updatedCookie = encodeURIComponent(name) + "=" + encodeURIComponent(value);

 for (let optionKey in options) {
 updatedCookie += "; " + optionKey;
 let optionValue = options[optionKey];
 if (optionValue !== true) {
 updatedCookie += "=" + optionValue;
 }
 }

 document.cookie = updatedCookie;
}

// Ejemplo de uso:
setCookie('user', 'John', {secure: true, 'max-age': 3600});
```

## deleteCookie(name)

Para borrar una cookie, podemos llamarla con una fecha de expiración negativa:

```
function deleteCookie(name) {
 setCookie(name, "", {
 'max-age': -1
 })
}
```

### ⚠️ La modificación o eliminación debe usar la misma ruta y dominio

Por favor nota que cuando alteramos o borramos una cookie debemos usar exactamente el mismo "path" y "domain" que cuando la establecimos.

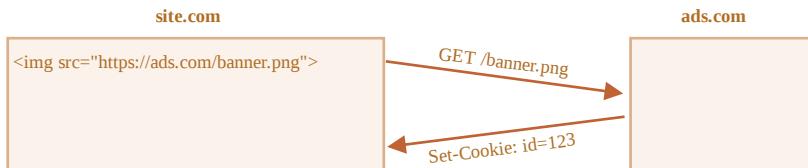
Completo: [cookie.js](#).

## Apéndice: Cookies de terceros

Una cookie es llamada “third-party” o “de terceros” si es colocada por un dominio distinto al de la página que el usuario está visitando.

Por ejemplo:

1. Una página en `site.com` carga un banner desde otro sitio: ``.
2. Junto con el banner, el servidor remoto en `ads.com` puede configurar la cabecera `Set-Cookie` con una cookie como `id=123`. Tal cookie tiene origen en el dominio `ads.com`, y será visible solamente en `ads.com`:



3. La próxima vez que se accede a `ads.com`, el servidor remoto obtiene la cookie `id` y reconoce al usuario:



4. Lo que es más importante aquí, cuando el usuario cambia de `site.com` a otro sitio `other.com` que también tiene un banner, entonces `ads.com` obtiene la cookie porque pertenece a `ads.com`, reconociendo al visitante y su movimiento entre sitios:



Las cookies de terceros son tradicionalmente usados para rastreo y servicios de publicidad (ads) debido a su naturaleza. Ellas están vinculadas al dominio de origen, entonces `ads.com` puede rastrear al mismo usuario a través de diferentes sitios si ellos los acceden.

Naturalmente, a algunos no les gusta ser seguidos, así que los navegadores permiten deshabilitar tales cookies.

Además, algunos navegadores modernos emplean políticas especiales para tales cookies:

- Safari no permite cookies de terceros en absoluto.
- Firefox viene con una “lista negra” de dominios de terceros y bloquea las cookies de tales orígenes.

#### **i Por favor tome nota:**

Si cargamos un script desde un dominio de terceros, como `<script src="https://google-analytics.com/analytics.js">`, y ese script usa `document.cookie` para configurar una cookie, tal cookie no es “de terceros”.

Si un script configura una cookie, no importa de dónde viene el script: la cookie pertenece al dominio de la página web actual.

## Apéndice: GDPR

Este tópico no está relacionado a JavaScript en absoluto, solo es algo para tener en mente cuando configuramos cookies.

Hay una legislación en Europa llamada GDPR que es un conjunto de reglas que fuerza a los sitios web a respetar la privacidad del usuario. Una de estas reglas es requerir un permiso explícito del usuario para el uso de cookies de seguimiento.

Nota que esto solo se refiere a cookies de seguimiento, identificación y autorización.

Así que si queremos configurar una cookie que solo guarda alguna información pero no hace seguimiento ni identificación del usuario, somos libres de hacerlo.

Pero si vamos a configurar una cookie con una sesión de autenticación o un id de seguimiento, el usuario debe dar su permiso.

Los sitios web generalmente tienen dos variantes para cumplir con el GDPR. Debes de haberlas visto en la web:

1. Si un sitio web quiere establecer cookies de seguimiento solo para usuarios autenticados.

Para hacerlo, el form de registro debe tener un checkbox como: "aceptar la política de privacidad" (que describe cómo las cookies son usadas), el usuario debe marcarlo, entonces el sitio web es libre para establecer cookies de autenticación.

2. Si un sitio web quiere establecer cookies de seguimiento a todo visitante.

Para hacerlo legalmente, el sitio web muestra un mensaje del tipo "pantalla de bienvenida (splash screen)" a los recién llegados que les pide aceptar las cookies. Entonces el sitio web puede configurarlas y les deja ver el contenido. Esto puede ser molesto para el visitante. A nadie le gusta que aparezca una pantalla modal con la obligación de cliquear en ella en lugar del contenido. Pero el GDPR requiere el acuerdo explícito.

El GDPR no trata solo de cookies, también es acerca de otros problemas relacionados a la privacidad, pero eso va más allá de nuestro objetivo.

## Resumen

`document.cookie` brinda acceso a las cookies.

- la operación de escritura modifica solo cookies mencionadas en ella.
- nombre y valor deben estar codificados.
- Una cookie no debe exceder los 4KB, y están limitadas a unas 20+ cookies por sitio (depende del navegador).

Opciones de Cookie:

- `path=/`, por defecto la ruta actual, hace la cookie visible solo bajo esa ruta.
- `domain=site.com`, por defecto una cookie es visible solo en el dominio actual. Si el dominio se establece explícitamente, la cookie se hace visible a los subdominios.
- `expires` o `max-age` configuran el tiempo de expiración de la cookie. Sin ellas la cookie muere cuando el navegador es cerrado.
- `secure` hace la cookie solo para HTTPS.
- `samesite` prohíbe al navegador enviar la cookie a solicitudes que vengan desde fuera del sitio. Esto ayuda a prevenir ataques XSRF.

Adicionalmente:

- Las cookies de terceros pueden estar prohibidas por el navegador, por ejemplo Safari lo hace por defecto.
- Cuando se configuran cookies de seguimiento para ciudadanos de la UE, la regulación GDPR requiere la autorización del usuario.

## LocalStorage, sessionStorage

Los objetos de almacenaje web `localStorage` y `sessionStorage` permiten guardar pares de clave/valor en el navegador.

Lo que es interesante sobre ellos es que los datos sobreviven a una recarga de página (en el caso de `sessionStorage`) y hasta un reinicio completo de navegador (en el caso de `localStorage`). Lo veremos en breve.

Ya tenemos cookies. ¿Por qué tener objetos adicionales?

- Al contrario que las cookies, los objetos de almacenaje web no se envían al servidor en cada petición. Debido a esto, podemos almacenar mucha más información. La mayoría de los navegadores modernos permiten almacenar, como mínimo, 5 megabytes de datos y tienen opciones para configurar estos límites.

- También diferente de las cookies es que el servidor no puede manipular los objetos de almacenaje vía cabeceras HTTP, todo se hace vía JavaScript.
- El almacenaje está vinculado al origen (al triplete dominio/protocolo/puerto). Esto significa que distintos protocolos o subdominios tienen distintos objetos de almacenaje, no pueden acceder a otros datos que no sean los suyos.

Ambos objetos de almacenaje proveen los mismos métodos y propiedades:

- `setItem(clave, valor)` – almacenar un par clave/valor.
- `getItem(clave)` – obtener el valor por medio de la clave.
- `removeItem(clave)` – eliminar la clave y su valor.
- `clear()` – borrar todo.
- `key(indice)` – obtener la clave de una posición dada.
- `length` – el número de ítems almacenados.

Como puedes ver, es como una colección `Map` (`setItem/getItem/removeItem`), pero también permite el acceso a través de `index` con `key(index)`.

Vamos a ver cómo funciona.

## Demo de localStorage

Las principales funcionalidades de `localStorage` son:

- Es compartido entre todas las pestañas y ventanas del mismo origen.
- Los datos no expiran. Persisten a los reinicios de navegador y hasta del sistema operativo.

Por ejemplo, si ejecutas éste código...

```
localStorage.setItem('test', 1);
```

... y cierras/abres el navegador, o simplemente abres la misma página en otra ventana, puedes coger el ítem que hemos guardado de este modo:

```
alert(localStorage.getItem('test')); // 1
```

Solo tenemos que estar en el mismo dominio/puerto/protocolo, la url puede ser distinta.

`localStorage` es compartido por toda las ventanas del mismo origen, de modo que si guardamos datos en una ventana, el cambio es visible en la otra.

## Acceso tipo Objeto

También podemos utilizar un modo de acceder/guardar claves del mismo modo que se hace con objetos, así:

```
// guarda una clave
localStorage.test = 2;

// coge una clave
alert(localStorage.test); // 2

// borra una clave
delete localStorage.test;
```

Esto se permite por razones históricas, y principalmente funciona, pero en general no se recomienda por dos motivos:

1. Si la clave es generada por el usuario, puede ser cualquier cosa, como `length` o `toString`, u otro método propio de `localStorage`. En este caso `getItem/setItem` funcionan correctamente, pero el acceso de simil-objeto falla;

```
let key = 'length';
localStorage[key] = 5; // Error, no se puede asignar 'length'
```

2. Existe un evento `storage`, que se dispara cuando modificamos los datos. Este evento no se dispara si utilizamos el acceso tipo objeto. Lo veremos más tarde en este capítulo.

## Iterando sobre las claves

Los métodos proporcionan la funcionalidad `get` / `set` / `remove`. ¿Pero cómo conseguimos todas las claves o valores guardados?

Desafortunadamente, los objetos de almacenaje no son iterables.

Una opción es utilizar iteración sobre un array:

```
for(let i=0; i<localStorage.length; i++) {
 let key = localStorage.key(i);
 alert(`#${key}: ${localStorage.getItem(key)}`);
}
```

Otra opción es utilizar el loop específico para objetos `for key in localStorage` tal como hacemos en objetos comunes.

Esta opción itera sobre las claves, pero también devuelve campos propios de `localStorage` que no necesitamos:

```
// mal intento
for(let key in localStorage) {
 alert(key); // muestra getItem,.setItem y otros campos que no nos interesan
}
```

... De modo que necesitamos o bien filtrar campos des del prototipo con la validación `hasOwnProperty`:

```
for(let key in localStorage) {
 if (!localStorage.hasOwnProperty(key)) {
 continue; // se salta claves como "setItem", "getItem" etc
 }
 alert(`#${key}: ${localStorage.getItem(key)}`);
}
```

... O simplemente acceder a las claves “propias” con `Object.keys` y iterar sobre éstas si es necesario:

```
let keys = Object.keys(localStorage);
for(let key of keys) {
 alert(`#${key}: ${localStorage.getItem(key)}`);
}
```

Esta última opción funciona, ya que `Object.keys` solo devuelve las claves que pertenecen al objeto, ignorando el prototipo.

## Solo strings

Hay que tener en cuenta que tanto la clave como el valor deben ser strings.

Si fueran de cualquier otro tipo, como un número o un objeto, se convertirían a cadena de texto automáticamente:

```
localStorage.user = {name: "John"};
alert(localStorage.user); // [object Object]
```

A pesar de eso, podemos utilizar `JSON` para almacenar objetos:

```
localStorage.user = JSON.stringify({name: "John"});

// en algún momento más tarde
let user = JSON.parse(localStorage.user);
alert(user.name); // John
```

También es posible pasar a texto todo el objeto de almacenaje, por ejemplo para debugear:

```
// se ha añadido opciones de formato a JSON.stringify para que el objeto se lea mejor
alert(JSON.stringify(localStorage, null, 2));
```

## sessionStorage

El objeto `sessionStorage` se utiliza mucho menos que `localStorage`.

Las propiedades y métodos son los mismos, pero es mucho más limitado:

- `sessionStorage` solo existe dentro de la pestaña actual del navegador.
  - Otra pestaña con la misma página tendrá un almacenaje distinto.
  - Pero se comparte entre iframes en la pestaña (asumiendo que tengan el mismo origen).
- Los datos sobreviven un refresco de página, pero no cerrar/abrir la pestaña.

Vamos a verlo en acción.

Ejecuta éste código...

```
sessionStorage.setItem('test', 1);
```

... Y recarga la página. Aún puedes acceder a los datos:

```
alert(sessionStorage.getItem('test')); // después de la recarga: 1
```

... Pero si abres la misma página en otra pestaña, y lo intentas de nuevo, el código anterior devuelve `null`, que significa que no se ha encontrado nada.

Esto es exactamente porque `sessionStorage` no está vinculado solamente al origen, sino también a la pestaña del navegador. Por esta razón `sessionStorage` se usa relativamente poco.

## Evento storage

Cuando los datos se actualizan en `localStorage` o en `sessionStorage`, se dispara el evento `storage ↗` con las propiedades:

- `key` – la clave que ha cambiado, (`null` si se llama `.clear()`).
- `oldValue` – el anterior valor (`null` si se añade una clave).
- `newValue` – el nuevo valor (`null` si se borra una clave).
- `url` – la url del documento donde ha pasado la actualización.
- `storageArea` – bien el objeto `localStorage` o `sessionStorage`, donde se ha producido la actualización.

El hecho importante es: el evento se dispara en todos los objetos `window` donde el almacenaje es accesible, excepto en el que lo ha causado.

Vamos a desarrollarlo.

Imagina que tienes dos ventanas con el mismo sitio en cada una, de modo que `localStorage` es compartido entre ellas.

Si ambas ventanas están escuchando el evento `window.onstorage`, cada una reaccionará a las actualizaciones que pasen en la otra.

```
// se dispara en actualizaciones hechas en el mismo almacenaje, desde otros documentos
window.onstorage = event => { // también puede usar window.addEventListener('storage', event => {
 if (event.key != 'now') return;
 alert(event.key + ':' + event.newValue + " at " + event.url);
};

localStorage.setItem('now', Date.now());
```

Hay que tener en cuenta que el evento también contiene: `event.url` – la url del documento en que se actualizaron los datos.

También que `event.storageArea` contiene el objeto de almacenaje – el evento es el mismo para `sessionStorage` y `localStorage` --, de modo que `storageArea` referencia el que se modificó. Podemos hasta querer cambiar datos en él, para “responder” a un cambio.

**Esto permite que distintas ventanas del mismo origen puedan intercambiar mensajes.**

Los navegadores modernos también soportan la [API de Broadcast channel API ↗](#), la API específica para la comunicación entre ventanas del mismo origen. Es más completa, pero tiene menos soporte. Hay librerías que añaden polyfills para ésta API basados en `localStorage` para que se pueda utilizar en cualquier entorno.

## Resumen

Los objetos de almacenaje web `localStorage` y `sessionStorage` permiten guardar pares de clave/valor en el navegador.

- Tanto la `clave` como el `valor` deben ser strings.
- El límite es de más de 5mb+, dependiendo del navegador.
- No expiran.
- Los datos están vinculados al origen (dominio/puerto/protocolo).

<code>localStorage</code>	<code>sessionStorage</code>
Compartida entre todas las pestañas y ventanas que tengan el mismo origen	Accesible en una pestaña del navegador, incluyendo iframes del mismo origen
Sobrevive a reinicios del navegador	Muere al cerrar la pestaña

API:

- `setItem(clave, valor)` – guarda pares clave/valor.
- `getItem(clave)` – coge el valor de una clave.
- `removeItem(clave)` – borra una clave con su valor.
- `clear()` – borra todo.
- `key(indice)` – coge la clave en una posición determinada.
- `length` – el número de ítems almacenados.
- Utiliza `Object.keys` para conseguir todas las claves.
- Puede utilizar las claves como propiedades de objeto, pero en ese caso el evento `storage` no se dispara

Evento storage:

- Se dispara en las llamadas a `setItem`, `removeItem`, `clear`.
- Contiene todos los datos relativos a la operación (`key/oldValue/newValue`), la `url` del documento y el objeto de almacenaje.
- Se dispara en todos los objetos `window` que tienen acceso al almacenaje excepto el que ha generado el evento (en una pestaña en el caso de `sessionStorage` o globalmente en el caso de `localStorage`).

## ✓ Tareas

### Guardar automáticamente un campo de formulario

Crea un campo `textarea` que “autoguarde” sus valores en cada cambio.

Entonces, si el usuario cierra accidentalmente la página y la abre de nuevo, encontrará su entrada inacabada en su lugar.

Como esto:

The image shows a simple form element consisting of a text area and a clear button. The text area has a placeholder text "Write here". Below the text area is a small button labeled "Clear".

Abrir un entorno controlado para la tarea. ↗

A solución

## IndexedDB

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

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

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

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

También podemos usar `async/await` con la ayuda de un contenedor basado en promesas como `idb` <https://github.com/jakearchibald/idb>. Aunque esto es muy conveniente, hay que tener en cuenta que el contenedor no es perfecto y no puede reemplazar a los eventos en todos los casos. Así que comenzaremos con eventos y, cuando hayamos avanzado en el entendimiento de `IndexedDb`, usaremos el contenedor.

### 💡 ¿Dónde están los datos?

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

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

## Apertura de una base de datos, “open”

Para empezar a trabajar con IndexedDB, primero necesitamos conectarnos o “abrir” (`open`) una base de datos.

La sintaxis:

```
let openRequest = indexedDB.open(name, version);
```

- `name` – un string, el nombre de la base de datos.
- `version` – un entero positivo, predeterminado en `1` (explicado más abajo).

Podemos tener muchas bases de datos con nombres diferentes, pero todas ellas existen dentro del mismo origen (dominio/protocolo/puerto). Un sitio web no puede acceder bases de datos de otro.

La llamada devuelve un objeto `openRequest`, debemos escuchar en él los eventos:

- `success` : la base de datos está lista. Hay un “objeto database” en `openRequest.result` que habremos de usar en las llamadas subsiguientes.
- `error` : Apertura fallida.
- `upgradeneeded` : La base de datos está lista, pero su versión es obsoleta (ver abajo).

## IndexedDB tiene incorporado un mecanismo de “versión de esquema”, ausente en bases de datos de servidor.

A diferencia de las bases de datos del lado del servidor, IndexedDB se ejecuta en el lado del cliente y los datos son almacenados en el navegador, así que nosotros, desarrolladores, no tenemos acceso permanente a esas bases. Entonces, cuando publicamos una nueva versión de nuestra app y el usuario visita nuestra página web, podemos necesitar actualizar la estructura de su base de datos.

Si la versión de la base es menor a la especificada en `open`, entonces se dispara un evento especial `upgradeneeded` (actualización-requerida), donde podemos comparar versiones y hacer la actualización de la estructura de datos que se necesite.

El evento `upgradeneeded` también se dispara cuando la base aún no existe (técticamente, su versión es `0`), lo cual nos permite llevar a cabo su inicialización.

Digamos que publicamos la primera versión de nuestra app.

Entonces podemos abrir la base con versión 1 y hacer la inicialización en un manejador upgradeneeded :

```
let openRequest = indexedDB.open("store", 1);

openRequest.onupgradeneeded = function() {
 // se dispara si el cliente no tiene la base de datos
 // ...ejecuta la inicialización...
};

openRequest.onerror = function() {
 console.error("Error", openRequest.error);
};

openRequest.onsuccess = function() {
 let db = openRequest.result;
 // continúa trabajando con la base de datos usando el objeto db
};
```

Luego, más tarde, publicamos la segunda versión.

Podemos abrirla con versión 2 y ejecutar la actualización así:

```
let openRequest = indexedDB.open("store", 2);

openRequest.onupgradeneeded = function(event) {
 // la versión de la base existente es menor que 2 (o ni siquiera existe)
 let db = openRequest.result;
 switch(event.oldVersion) { // versión de db existente
 case 0:
 // versión 0 significa que el cliente no tiene base de datos
 // ejecutar inicialización
 case 1:
 // el cliente tiene la versión 1
 // actualizar
 }
};
```

Tenlo en cuenta: como nuestra versión actual es 2, el manejador onupgradeneeded tiene una rama de código para la versión 0, adecuada para usuarios que acceden por primera vez y no tienen una base de datos, y otra rama para la versión 1, para su actualización.

Entonces, y solamente si el manejador de onupgradeneeded finaliza sin errores, se dispara el evento openRequest.onsuccess y se considera que la base de datos fue abierta con éxito.

Para borrar una base de datos:

```
let deleteRequest = indexedDB.deleteDatabase(name)
// deleteRequest.onsuccess/onerror rastrea el resultado
```

### ⚠ No se puede abrir una base de datos usando una versión más vieja de open

Si la base del usuario tiene una versión mayor que el open que la abre, por ejemplo: la base existente tiene versión 3 e intentamos open(..., 2), se producirá un error que disparará openRequest.onerror.

Es una situación rara, pero puede ocurrir cuando un visitante carga código JavaScript viejo (por ejemplo desde un caché proxy). Así el código es viejo, pero la base de datos nueva.

Para prevenir errores, debemos verificar db.version y sugerir la recarga de página. Usa cabeceras HTTP de caché apropiadas para evitar la carga de código viejo, así nunca tendrás tales problemas.

## El problema de la actualización paralela

Hablando de versionado, encaremos un pequeño problema relacionado.

Supongamos que:

1. Un visitante, en una pestaña de su navegador, abrió nuestro sitio con un base de datos con la versión 1.

2. Luego publicamos una actualización, así que nuestro código es más reciente.

3. Y el mismo visitante abre nuestro sitio en otra pestaña.

Entonces hay una primera pestaña con una conexión abierta a una base con versión 1, mientras la segunda intenta actualizarla a la versión 2 en su manejador upgradeneeded .

El problema es que la misma base está compartida entre las dos pestañas, por ser del mismo sitio y origen. Y no puede ser versión 1 y 2 al mismo tiempo. Para ejecutar la actualización a la versión 2, todas las conexiones a la versión 1 deben ser cerradas, incluyendo las de la primera pestaña.

Para detectar estas situaciones, se dispara automáticamente el evento versionchange (cambio-de-versión) en el objeto de base de datos. Debemos escuchar dicho evento y cerrar la conexión vieja (y probablemente sugerir una recarga de página, para cargar el código actualizado).

Si no escuchamos el evento versionchange y no cerramos la conexión vieja, entonces la segunda y más nueva no se podrá hacer. El objeto openRequest emitirá el evento blocked en lugar de success . Entonces la segunda pestaña no funcionará.

Aquí tenemos el código para manejar correctamente la actualización paralela. Este instala un manejador onversionchange que se dispara si la conexión actual queda obsoleta y la cierra (la versión se actualiza en algún otro lado):

```
let openRequest = indexedDB.open("store", 2);

openRequest.onupgradeneeded = ...;
openRequest.onerror = ...;

openRequest.onsuccess = function() {
 let db = openRequest.result;

 db.onversionchange = function() {
 db.close();
 alert("La base de datos está desactualizada, por favor recargue la página.")
 };

 // ...la base db está lista, úsala...
};

openRequest.onblocked = function() {
 // este evento no debería dispararse si hemos manejado onversionchange correctamente

 // significa que hay otra conexión abierta a la misma base
 // que no fue cerrada después de que se disparó db.onversionchange
};
```

Aquí hacemos dos cosas:

1. La escucha a db.onversionchange nos informa de un intento de actualización paralela si la conexión actual se volvió obsoleta.
2. La escucha a openRequest.onblocked nos informa de la situación opuesta: hay una conexión obsoleta en algún otro lugar que no fue cerrada y por eso la conexión nueva no se pudo realizar.

Podemos manejar las cosas más suavemente en db.onversionchange , como pedirle al visitante que guarde los datos antes de cerrar la conexión.

Como alternativa podríamos no cerrar la base en db.onversionchange sino usar onblocked de la nueva pestaña para advertirle que no puede crear una nueva conexión hasta que cierre las viejas.

Estas colisiones ocurren raramente, pero deberíamos tener algún manejo de ella, como mínimo un manejador onblocked para evitar que nuestro script muera silenciosamente.

## Almacén de objetos, “store”

Para almacenar algo en IndexedDB, necesitamos un “almacén de objetos” object store.

Un almacén de objetos es un concepto central de IndexedDB. Equivale a lo que en otras bases de datos se denominan “tablas” o “colecciones”. Es donde los datos son almacenados. Una base de datos puede tener múltiples almacenes: uno para usuarios, otro para bienes, etc.

A pesar de llamarse “almacén de objetos”, también puede almacenar tipos primitivos.

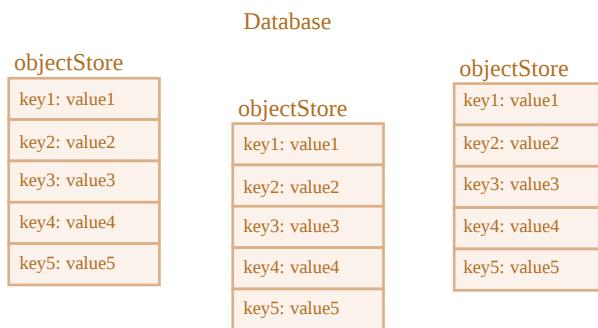
## Podemos almacenar casi cualquier valor, incluyendo objetos complejos.

IndexedDB usa el [algoritmo de serialización estándar](#) para clonar-y-almacenar un objeto. Es como `JSON.stringify`; pero más poderoso, capaz de almacenar muchos tipos de datos más.

Hay objetos que no pueden ser almacenados, por ejemplo los que tienen referencias circulares. Tales objetos no son serializables. `JSON.stringify` también falla con ellos.

## Debe haber una clave `key` única para cada valor del almacén.

Una clave debe ser de uno de estos tipos: number, date, string, binary, o array. Es un identificador único, así podemos buscar/borrar/modificar valores por medio de la clave.



Como veremos pronto, cuando agregamos un valor al almacén podemos proporcionarle una clave, de forma similar a `localStorage`. Pero cuando lo que almacenamos son objetos, IndexedDB permite asignar una propiedad del objeto como clave, lo que es mucho más conveniente. También podemos usar claves que se generan automáticamente.

Pero primero, necesitamos crear el almacén de objetos.

La sintaxis para crear un almacén de objetos u “object store”:

```
db.createObjectStore(name[, keyOptions]);
```

Ten en cuenta que esta operación es sincrónica, no requiere `await`.

- `name` es el nombre del almacén, por ejemplo `"books"`,
- `keyOptions` es un objeto opcional con una de estas dos propiedades:
  - `keyPath` – la ruta a una propiedad del objeto que IndexedDB usará como clave, por ejemplo `id`.
  - `autoIncrement` – si es `true`, la clave para el objeto nuevo que se almacene se generará automáticamente con un número autoincremental.

Si no establecemos `keyOptions`, necesitaremos proporcionar una clave explícitamente más tarde: al momento de almacenar un objeto.

Por ejemplo, este objeto usa la propiedad `id` como clave:

```
db.createObjectStore('books', {keyPath: 'id'});
```

**Un almacén de objetos solo puede ser creado o modificado durante la actualización de su versión, esto es, en el manejador `upgradeneeded`.**

Esto es una limitación técnica. Fuera del manejador podremos agregar/borrar/modificar los datos, pero los almacenes de objetos solo pueden ser creados/borrados/alterados durante la actualización de versión.

Para hacer una actualización de base de datos, hay principalmente dos enfoques:

1. Podemos implementar una función de actualización por versión: desde 1 a 2, de 2 a 3, de 3 a 4, etc. Así en `upgradeneeded` podemos comparar versiones (ejemplo: vieja 2, ahora 4) y ejecutar actualizaciones por versión paso a paso para cada versión intermedia (en el ejemplo: 2 a 3, luego 3 a 4).
2. O podemos simplemente examinar la base y alterarla en un paso. Obtenemos una lista de los almacenes existentes como `db.objectStoreNames`. Este objeto es un [DOMStringList](#) que brinda el método `contains(name)` para chequear existencias. Y podemos entonces hacer actualizaciones dependiendo de lo que existe y lo que no.

En bases de datos pequeñas la segunda variante puede ser más simple.

Aquí hay un demo del segundo enfoque:

```
let openRequest = indexedDB.open("db", 2);

// crea/actualiza la base de datos sin chequeo de versiones
openRequest.onupgradeneeded = function() {
 let db = openRequest.result;
 if (!db.objectNames.contains('books')) { // si no hay un almacén de libros ("books"),
 db.createObjectStore('books', {keyPath: 'id'}); // crearlo
 }
};
```

Para borrar un almacén de objetos:

```
db.deleteObjectStore('books')
```

## Transacciones

El término transacción es genérico, usado por muchos tipos de bases de datos.

Una transacción es un grupo de operaciones cuyos resultados están vinculados: todas deben ser exitosas o todas fallar.

Por ejemplo, cuando una persona compra algo, necesitamos:

1. Restar el dinero de su cuenta personal.
2. Agregar el ítem a su inventario.

Sería muy malo que si se completara la primera operación y algo saliera mal (como un corte de luz), fallara la segunda. Ambas deberían ser exitosas (compra completa, ¡bien!) o ambas fallar (al menos la persona mantuvo su dinero y puede reintentar).

Las transacciones garantizan eso.

**Todas las operaciones deben ser hechas dentro de una transacción en IndexedDB.**

Para iniciar una transacción:

```
db.transaction(store[, type]);
```

- `store` – el nombre de almacén al que la transacción va a acceder, por ejemplo `"books"`. Puede ser un array de nombres de almacenes si vamos a acceder a múltiples almacenes.
- `type` – el tipo de transacción, uno de estos dos:
  - `readonly` – solo puede leer (es el predeterminado).
  - `readwrite` – puede leer o escribir datos (pero no crear/quitar/alterar almacenes de objetos).

También existe el tipo de transacción `versionchange`: tal transacción puede hacer de todo, pero no podemos crearla nosotros a mano. IndexedDB la crea automáticamente cuando abre la base de datos para el manejador `upgradeneeded`. Por ello, es el único lugar donde podemos actualizar la estructura de base de datos, crear o quitar almacenes de objetos.

### ¿Por qué hay diferentes tipos de transacciones?

El rendimiento es la razón por la que necesitamos identificar las transacciones como `readonly` (lectura solamente) o `readwrite` (lectura y escritura).

Muchas transacciones `readonly` pueden leer en un mismo almacén concurrentemente, en cambio las transacciones de escritura `readwrite`, no. Una transacción `readwrite` bloquea el almacén para escribir en él. La siguiente transacción debe esperar a que la anterior termine antes de acceder al mismo almacén.

Una vez que la transacción ha sido creada, podemos agregar un ítem al almacén:

```
let transaction = db.transaction("books", "readwrite"); // (1)

// obtiene un almacén de objetos para operar con él
let books = transaction.objectStore("books"); // (2)
```

```

let book = {
 id: 'js',
 price: 10,
 created: new Date()
};

let request = books.add(book); // (3)

request.onsuccess = function() { // (4)
 console.log("Libro agregado al almacén", request.result);
};

request.onerror = function() {
 console.log("Error", request.error);
};

```

Básicamente, hay cuatro pasos:

1. Crea una transacción, mencionando todos los almacenes a los que irá a acceder, en (1).
2. Obtiene el almacén usando `transaction.objectStore(name)`, en (2).
3. Ejecuta la petición al almacén `books.add(book)`, en (3).
4. ...Maneja el éxito o fracaso de la petición (4), a continuación podemos hacer otras peticiones si lo necesitamos, etc.

Los almacenes de objetos soportan dos métodos para almacenar un valor:

- **`put(value, [key])`** Agrega `value` al almacén. La clave `key` debe ser suministrada solo si al almacén no se le asignó la opción `keyPath` o `autoIncrement`. Si ya hay un valor con la misma clave, este será reemplazado.
- **`add(value, [key])`** Lo mismo que `put`, pero si ya hay un valor con la misma clave, la petición falla y se genera un error con el nombre `"ConstraintError"`.

Al igual que al abrir una base de datos, podemos enviar una petición: `books.add(book)` y quedar a la espera de los eventos `success/error`.

- El resultado `request.result` de `add` es la clave del nuevo objeto.
- El error, si lo hay, está en `request.error`.

## Commit, culminación automática de las transacciones

En el ejemplo anterior, empezamos la transacción e hicimos una petición `add`. Pero, como explicamos antes, una transacción puede tener muchas peticiones asociadas, que deben todas ser exitosas o todas fallar. ¿Cómo marcamos que una transacción se da por finalizada, que no tendrá más peticiones asociadas?

Respuesta corta: no lo hacemos.

En la siguiente versión 3.0 de la especificación, probablemente haya una manera de finalizarla manualmente, pero ahora mismo en la 2.0 no la hay.

**Cuando todas las peticiones de una transacción terminaron y la cola de microtareas está vacía, se hace un commit (consumación) automático.**

De forma general, podemos asumir que una transacción se consuma cuando todas sus peticiones fueron completadas y el código actual finaliza.

Entonces, en el ejemplo anterior no se necesita una llamada especial para finalizar la transacción.

El principio de auto-commit de las transacciones tiene un efecto colateral importante. No podemos insertar una operación asíncrona como `fetch`, `setTimeout` en mitad de una transacción. IndexedDB no mantendrá la transacción esperando a que terminen.

En el siguiente código, `request2` en la línea (\*) falla, porque la transacción ya está finalizada y no podemos hacer más peticiones sobre ella:

```

let request1 = books.add(book);

request1.onsuccess = function() {
 fetch('/').then(response => {
 let request2 = books.add(anotherBook); // (*)
 request2.onerror = function() {

```

```
 console.log(request2.error.name); // TransactionInactiveError
 };
});
};
```

Esto es porque `fetch` es una operación asincrónica, una macrotarea. Las transacciones se cierran antes de que el navegador comience con las macrotareas.

Los autores de la especificación de IndexedDB creen que las transacciones deben ser de corta vida. Mayormente por razones de rendimiento.

Es de notar que las transacciones `readwrite` "trablan" los almacenes para escritura. Entonces si una parte de la aplicación inició `readwrite` en el almacén `books`, cuando otra parte quiera hacer lo mismo tendrá que esperar: la nueva transacción "se cuelga" hasta que la primera termine. Esto puede llevar a extraños retardos si las transacciones toman un tiempo largo.

Entonces, ¿qué hacer?

En el ejemplo de arriba podemos hacer una nueva `db.transaction` justo antes de la nueva petición (\*).

Pero, si queremos mantener las operaciones juntas en una transacción, será mucho mejor separar las transacciones IndexedDB de la parte asincrónica.

Primero, hacer `fetch` y preparar todos los datos que fueran necesarios y, solo entonces, crear una transacción y ejecutar todas las peticiones de base de datos. Así, funcionaría.

Para detectar el momento de finalización exitosa, podemos escuchar al evento `transaction.oncomplete`:

```
let transaction = db.transaction("books", "readwrite");

// ...ejecutar las operaciones...

transaction.oncomplete = function() {
 console.log("Transacción completa");
};
```

Solo `complete` garantiza que la transacción fue guardada como un todo. Las peticiones individuales pueden ser exitosas, pero la operación final de escritura puede ir mal (por ejemplo por un error de Entrada/Salida u otra cosa).

Para abortar una transacción manualmente:

```
transaction.abort();
```

Esto cancela todas las modificaciones hechas por las peticiones y dispara el evento `transaction.onabort`.

## Manejo de error

Las peticiones de escritura pueden fallar.

Esto es esperable, no solo por posibles errores de nuestro lado, sino también por razones no relacionadas con la transacción en si misma. Por ejemplo, la cuota de almacenamiento podría haberse excedido. Por tanto, debemos estar preparados para manejar tal caso.

**Una petición fallida automáticamente aborta la transacción, cancelando todos sus cambios.**

En algunas situaciones, podemos querer manejar el fallo (por ejemplo, intentar otra petición) sin cancelar los cambios en curso, y continuar la transacción. Eso es posible. El manejador `request.onerror` es capaz de evitar el aborto de la transacción llamando a `event.preventDefault()`.

En el ejemplo que sigue, un libro nuevo es agregado con la misma clave (`id`) que otro existente. El método `store.add` genera un `"ConstraintError"` en ese caso. Lo manejamos sin cancelar la transacción:

```
let transaction = db.transaction("books", "readwrite");

let book = { id: 'js', price: 10 };

let request = transaction.objectStore("books").add(book);

request.onerror = function(event) {
```

```

// ConstraintError ocurre cuando un objeto con el mismo id ya existe
if (request.error.name == "ConstraintError") {
 console.log("Ya existe un libro con ese id"); // manejo del error
 event.preventDefault(); // no abortar la transacción
 // ¿usar otra clave para el libro?
} else {
 // error inesperado, no podemos manejarlo
 // la transacción se abortará
}
};

transaction.onabort = function() {
 console.log("Error", transaction.error);
};

```

## Delegación de eventos

¿Necesitamos onerror/onsuccess en cada petición? No siempre. En su lugar podemos usar la delegación de eventos.

**Propagación de eventos IndexedDB:** `request` → `transaction` → `database`.

Todos los eventos son eventos DOM, con captura y propagación, pero generalmente solo se usa el escenario de la propagación.

Así que podemos capturar todos los errores usando el manejador `db.onerror`, para reportarlos u otros propósitos:

```

db.onerror = function(event) {
 let request = event.target; // la petición (request) que causó el error

 console.log("Error", request.error);
};

```

...Pero ¿y si el error fue completamente manejado? No queremos elevarlo en ese caso.

Podemos detener la propagación y en consecuencia `db.onerror` usando `event.stopPropagation()` en `request.onerror`.

```

request.onerror = function(event) {
 if (request.error.name == "ConstraintError") {
 console.log("Ya existe un libro con ese id"); // manejo de error
 event.preventDefault(); // no abortar la transacción
 event.stopPropagation(); // no propagar el error
 } else {
 // no hacer nada
 // la transacción será abortada
 // podemos encargarnos del error en transaction.onabort
 }
};

```

## Búsquedas

Hay dos maneras principales de buscar en un almacén de objetos:

1. Por clave o por rango de claves. En nuestro almacén “books”, puede ser por un valor o por un rango de valores de `book.id`.
2. Por algún otro campo del objeto, por ejemplo `book.price`. Esto requiere una estructura de datos adicional llamada índice “index”.

### Por clave

Veamos el primer tipo de búsqueda: por clave.

Los métodos de búsqueda soportan tanto las claves exactas como las denominadas “consultas por rango” que son objetos [IDBKeyRange](#) ↪ que especifican un “rango de claves” aceptable.

Los objetos `IDBKeyRange` son creados con las siguientes llamadas:

- `IDBKeyRange.lowerBound(lower, [open])` significa:  $\geq$  lower (o  $>$  lower si open es true)
- `IDBKeyRange.upperBound(upper, [open])` significa:  $\leq$  upper (o  $<$  upper si open es true)

- `IDBKeyRange.bound(lower, upper, [lowerOpen], [upperOpen])` significa: entre `lower` y `upper`. Si el indicador “open” es true, la clave correspondiente no es incluida en el rango.
- `IDBKeyRange.only(key)` – es un rango compuesto solamente por una clave `key`, es raramente usado.

Veremos ejemplos prácticos de uso muy pronto.

Para efectuar la búsqueda, existen los siguientes métodos. Ellos aceptan un argumento `query` que puede ser una clave exacta o un rango de claves:

- `store.get(query)` – busca el primer valor, por clave o por rango.
- `store.getAll([query], [count])` – busca todos los valores, limitado a la cantidad `count` si esta se especifica.
- `store.getKey(query)` – busca la primera clave que satisface la consulta, usualmente un rango.
- `store.getAllKeys([query], [count])` – busca todas las claves que satisfacen la consulta, usualmente un rango, hasta la cantidad `count` si es suministrada.
- `store.count([query])` – obtiene la cantidad de claves que satisfacen la consulta, usualmente un rango.

Por ejemplo, tenemos un montón de libros en nuestro almacén. Recuerda, el campo `id` es la clave, así que todos estos métodos pueden buscar por `id`.

Ejemplos de peticiones:

```
// obtiene un libro
books.get('js')

// obtiene libros con: 'css' <= id <= 'html'
books.getAll(IDBKeyRange.bound('css', 'html'))

// obtiene libros con id < 'html'
books.getAll(IDBKeyRange.upperBound('html', true))

// obtiene todos los libros
books.getAll()

// obtiene todas las claves donde id > 'js'
books.getAllKeys(IDBKeyRange.lowerBound('js', true))
```

#### **i El almacén de objetos siempre está ordenado**

El almacén internamente guarda los valores ordenados por clave.

Entonces, en las peticiones que devuelvan varios valores, estos siempre estarán ordenados por la clave.

## Buscando por cualquier campo con un índice

Para buscar por otro campo del objeto, necesitamos crear una estructura de datos adicional llamada “índice (index)”.

Un índice es un agregado al almacén que rastrea un campo determinado del objeto dado. Para cada valor de ese campo, almacena una lista de claves de objetos que tienen ese valor. Veremos una imagen más detallada abajo.

La sintaxis:

```
objectStore.createIndex(name, keyPath, [options]);
```

- `name` – nombre del índice,
- `keyPath` – ruta al campo del objeto que el índice debe seguir (vamos a buscar por ese campo),
- `option` – un objeto opcional con las propiedades:
  - `unique` – si es true, un valor no podrá repetirse en el índice. Solamente puede haber un único objeto en el almacén con un valor dado de su `keyPath`. El índice forzará esto generando un error si intentamos agregar un duplicado.
  - `multiEntry` – solo se usa si el valor en `keyPath` es un array. En ese caso, de manera predeterminada, el índice tratará el array completo como clave. Pero si `multiEntry` es true, entonces el índice mantendrá una lista de objetos almacenados para cada valor en ese array. Así los miembros del array se vuelven claves de ese índice.

En nuestro ejemplo, almacenamos libros usando la propiedad `id` como clave.

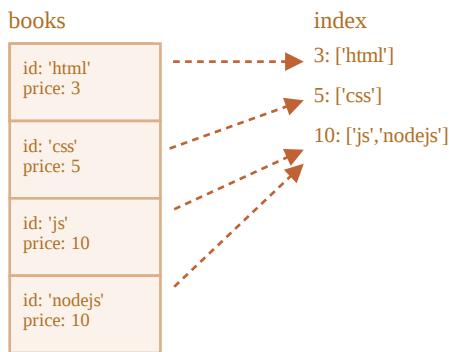
Digamos que queremos buscar por precio `price`.

Primero necesitamos crear un índice. Esto debe hacerse en `upgradeneeded`, al igual que hacíamos la creación del almacén de objetos.

```
openRequest.onupgradeneeded = function() {
 // debemos crear el índice aquí, en la transacción versionchange
 let books = db.createObjectStore('books', {keyPath: 'id'});
 let index = books.createIndex('price_idx', 'price');
};
```

- El índice hará seguimiento del campo `price`.
- El precio no es único, puede haber múltiples libros con el mismo precio así que no establecemos la opción `unique`.
- El precio no es un array, entonces el indicador `multiEntry` no es aplicable.

Imagine que nuestro `inventory` tiene 4 libros. Aquí la imagen muestra exactamente lo que es el índice:



Como se dijo, el índice para cada valor de `price` (segundo argumento) mantiene la lista de claves que tienen ese precio.

El índice se mantiene actualizado automáticamente, no necesitamos preocuparnos de eso.

Ahora, cuando queremos buscar por un determinado precio, simplemente aplicamos el mismo método de búsqueda al índice:

```
let transaction = db.transaction("books"); // readonly
let books = transaction.objectStore("books");
let priceIndex = books.index("price_idx");

let request = priceIndex.getAll(10);

request.onerror = function() {
 if (request.result !== undefined) {
 console.log("Books", request.result); // array de libros con precio = 10
 } else {
 console.log("No hay libros así");
 }
};
```

También podemos usar `IDBKeyRange` para crear rangos y vistas de libros baratos/caros:

```
// encontrar libros donde price <= 5
let request = priceIndex.getAll(IDBKeyRange.upperBound(5));
```

Los índices están ordenados internamente por el campo del índice, en nuestro caso `price`. Entonces cuando hacemos la búsqueda, los resultados estarán ordenados por `price`.

## Borrando del almacén

El método `delete` (eliminar) busca a través de una consulta valores para borrarlos. El formato de la llamada es similar a `getAll`:

- `delete(query)` – elimina valores coincidentes con una consulta (query).

Por ejemplo:

```
// borra el libro cuyo id='js'
books.delete('js');
```

Si queremos borrar libros basados en un precio u otro campo del objeto, debemos primero encontrar la clave en el índice, luego llamar a `delete` con dicha clave:

```
// encuentra la clave donde price = 5
let request = priceIndex.getKey(5);

request.onerror = function() {
 let id = request.result;
 let deleteRequest = books.delete(id);
};
```

Para borrar todo:

```
books.clear(); // clear "limpia" el almacén.
```

## Cursos

Métodos como `getAll/getAllKeys` devuelven un array de claves/valores.

Pero un almacén de objetos puede ser enorme, incluso más que la memoria disponible. Entonces `getAll` fallaría al tratar de llenar de registros el array.

¿Qué hacer?

Los cursos brindan los medios para manejar esta situación.

**Un cursor es un objeto especial que, dada una consulta, recorre el almacén y devuelve un solo par clave/valor cada vez, ahorrando así memoria.**

Como un almacén está ordenado internamente por clave, un cursor lo recorre en el orden de la clave (ascendente de forma predeterminada).

La sintaxis:

```
// como getAll, pero con un cursor:
let request = store.openCursor(query, [direction]);

// para obtener las claves y no sus valores (como getAllKeys): store.openKeyCursor
```

- `query` (consulta) es una clave o un rango de claves, al igual que para `getAll`.
- `direction` es un argumento opcional, el orden que se va a usar:
  - `"next"` – el predeterminado: el cursor recorre en orden ascendente comenzando por la clave más baja.
  - `"prev"` – el orden inverso: decrece comenzando con el registro con la clave más alta.
  - `"nextunique"`, `"prevunique"` – igual que las anteriores, pero saltando los registros con la misma clave (válido solo para cursos sobre índices; por ejemplo, de múltiples libros con `price=5`, solamente el primero será devuelto).

**La diferencia principal del cursor es que `request.onerror` se dispara múltiples veces: una por cada resultado.**

Aquí hay un ejemplo de cómo usar un cursor:

```
let transaction = db.transaction("books");
let books = transaction.objectStore("books");

let request = books.openCursor();

// llamado por cada libro encontrado por el cursor
request.onerror = function() {
 let cursor = request.result;
 if (cursor) {
```

```

let key = cursor.key; // clave del libro (el campo id)
let value = cursor.value; // el objeto libro
console.log(key, value);
cursor.continue();
} else {
 console.log("No hay más libros");
}
};

```

Los principales métodos de cursor son:

- `advance(count)` – avanza el cursor `count` veces, saltando valores.
- `continue([key])` – avanza el cursor al siguiente valor en el rango o, si se provee la clave `key`, al valor inmediatamente posterior a `key`.

El evento `onsuccess` será llamado haya o no más valores coincidentes, y en `result` obtenemos el cursor apuntando al siguiente registro o `undefined`.

En el ejemplo anterior, el cursor fue hecho sobre el almacén de objetos.

Pero también podemos hacerlo sobre un índice. Recordamos, los índices nos permiten buscar por los campos del objeto. Los cursos sobre índices hacen precisamente lo mismo que sobre el almacén de objetos: ahorran memoria al devolver un solo valor cada vez.

Para cursos sobre índices, `cursor.key` es la clave del índice (es decir “price”), y debemos usar la propiedad `cursor.primaryKey` para la clave del objeto:

```

let request = priceIdx.openCursor(IDBKeyRange.upperBound(5));

// es llamado para cada registro
request.onsuccess = function() {
 let cursor = request.result;
 if (cursor) {
 let primaryKey = cursor.primaryKey; // la siguiente clave (campo id) del almacén
 let value = cursor.value; // el siguiente objeto (objeto book) del almacén
 let key = cursor.key; // siguiente clave del índice (price)
 console.log(key, value);
 cursor.continue();
 } else {
 console.log("No hay más libros");
 }
};

```

## Contenedor promisificador

Agregar `onsuccess/onerror` a cada petición es una tarea agobiante. A veces podemos hacernos la vida más fácil usando delegación de eventos (por ejemplo, estableciendo manejadores para las transacciones completas), pero `async/await` es mucho más conveniente.

Usemos en adelante para este capítulo un contenedor (wrapper) liviano que añade promesas <https://github.com/jakearchibald/idb>. Este contenedor crea un objeto global `idb` con métodos IndexedDB promisificados.

Entonces, en lugar de `onsuccess/onerror`, podemos escribir:

```

let db = await idb.openDB('store', 1, db => {
 if (db.oldVersion == 0) {
 // ejecuta la inicialización
 db.createObjectStore('books', {keyPath: 'id'});
 }
});

let transaction = db.transaction('books', 'readwrite');
let books = transaction.objectStore('books');

try {
 await books.add(...);
 await books.add(...);

 await transaction.complete;
}

```

```
 console.log('jsbook saved');
} catch(err) {
 console.log('error', err.message);
}
```

Así tenemos todo lo dulce de “código async plano” y “try...catch”.

## Manejo de Error

Si no atrapamos un error, este se propaga hasta el `try..catch` externo más cercano.

Un error no atrapado se vuelve un evento “rechazo de promesa no manejado” sobre el objeto `window`.

Podemos manejar tales errores así:

```
window.addEventListener('unhandledrejection', event => {
 let request = event.target; // objeto request nativo de IndexedDB
 let error = event.reason; // objeto error no manejado, igual que request.error
 ...reportar el error...
});
```

## La trampa “transacción inactiva”

Como sabemos, una transacción se autofinaliza tan pronto como el navegador termina el código actual y las microtareas. Por tanto, si ponemos una *macrotarea* como `fetch` en el medio de una transacción, la transacción no esperará a que termine. Simplemente se autofinaliza. Así la siguiente petición fallaría.

Para el contenedor de promisificación y `async/await` la situación es la misma.

Este es un ejemplo de `fetch` en el medio de una transacción:

```
let transaction = db.transaction("inventory", "readwrite");
let inventory = transaction.objectStore("inventory");

await inventory.add({ id: 'js', price: 10, created: new Date() });

await fetch(...); // (*)

await inventory.add({ id: 'js', price: 10, created: new Date() }); // Error
```

El `inventory.add` que sigue a `fetch` (\*) falla con el error “transacción inactiva”, porque la transacción se autocompletó y, llegado ese momento, ya está cerrada.

La forma de sortear esto es la misma que con el IndexedDB nativo: Hacer una nueva transacción o simplemente partir las cosas.

1. Preparar los datos y buscar todo lo que sea necesario primero.
2. Solo entonces, grabar en la base de datos.

## Obtener objetos nativos

Internamente, el contenedor ejecuta una petición IndexedDB nativa, agregándole `onerror/onsuccess` y devolviendo una promesa que rechaza/resuelve con el resultado.

Esto funciona bien la mayor parte del tiempo. Los ejemplos están en la página lib de idb [https://github.com/jakearchibald/idb ↴](https://github.com/jakearchibald/idb).

En algunos raros casos necesitamos el objeto `request` original. Podemos accederlo con la propiedad `promise.request` de la promesa:

```
let promise = books.add(book); // obtiene una promesa (no espera por su resultado)

let request = promise.request; // objeto request nativo
let transaction = request.transaction; // objeto transaction nativo

// ...hace algún vudú IndexedDB...

let result = await promise; // si aún se necesita
```

## Resumen

IndexedDB puede considerarse como “localStorage con esteroides”. Es una simple base de datos de clave-valor, suficientemente poderosa para apps fuera de línea y fácil de usar.

El mejor manual es la especificación, [la actual](#) es 2.0, pero algunos métodos de [3.0](#) (no muy diferente) están soportados parcialmente.

El uso básico puede ser descrito en pocas frases:

1. Obtenga un contenedor promisificador como `idb`.
2. Abra la base de datos: `idb.openDb(name, version, onupgradeneeded)`
  - Cree almacenes de objetos e índices en el manejador `onupgradeneeded` o ejecute la actualización de versión cuando sea necesario.
3. Para peticiones:
  - Cree una transacción `db.transaction('books')` (readwrite si es necesario).
  - Obtenga el almacén de objetos `transaction.objectStore('books')`.
4. Entonces, para buscar por clave, llame métodos sobre el almacén directamente.
  - Para buscar por un campo de objeto, cree un índice.
5. Si los datos son demasiados para la memoria, use un cursor.

Una pequeña app de demo:

<https://plnkr.co/edit/FZY0VJ3rZQAhKXmg?p=preview>

## Animaciones

Animaciones con CSS y JavaScript.

### Curva de Bézier

Las curvas de Bézier se utilizan en gráficos por ordenador para dibujar formas, para animación CSS y en muchos otros lugares.

En realidad, son algo muy sencillo, vale la pena estudiarlos una vez y luego sentirse cómodo en el mundo de los gráficos vectoriales y las animaciones avanzadas.

#### **i** Un poco de teoría, por favor

Este artículo brinda una base teórica, pero muy necesaria, de lo que son las curvas Bezier; mientras que el [próximo](#) muestra cómo podemos usarlas en animaciones CSS.

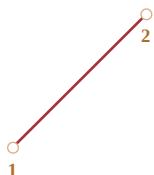
Por favor toma tu tiempo en leer y entender el concepto, te servirá bien.

### Puntos de Control

Una [curva de Bézier](#) está definida por puntos de control.

Puede haber 2, 3, 4 o más.

Por ejemplo, curva de dos puntos:



Curva de tres puntos:

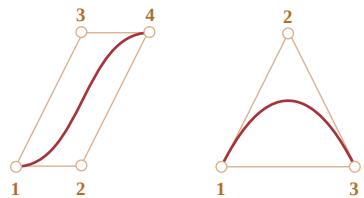


Curva de cuatro puntos:



Si observas detenidamente estas curvas, puedes notar inmediatamente que:

1. **Los puntos no siempre están en la curva.** Eso es perfectamente normal, luego veremos cómo se construye la curva.
2. **El orden de la curva es igual al número de puntos menos uno.** Para dos puntos tenemos una curva lineal (que es una línea recta), para tres puntos – curva cuadrática (parabólica), para cuatro puntos – curva cúbica.
3. **Una curva siempre está dentro del casco convexo ↗ de los puntos de control:**



Debido a esa última propiedad, en gráficos por ordenador es posible optimizar las pruebas de intersección. Si los cascillos convexos no se intersecan, las curvas tampoco. Por tanto, comprobar primero la intersección de los cascillos convexos puede dar un resultado “sin intersección” muy rápido. La comprobación de la intersección o los cascillos convexos es mucho más fácil, porque son rectángulos, triángulos, etc. (vea la imagen de arriba), figuras mucho más simples que la curva.

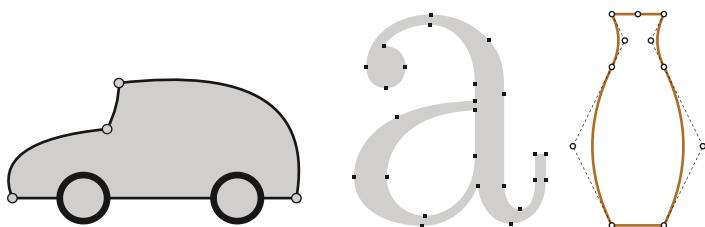
**El valor principal de las curvas de Bézier para dibujar: al mover los puntos, la curva cambia de manera intuitiva.**

Intenta mover los puntos de control con el ratón en el siguiente ejemplo:

Como puedes observar, la curva se extiende a lo largo de las líneas tangenciales  $1 \rightarrow 2$  y  $3 \rightarrow 4$ .

Después de algo de práctica, se vuelve obvio cómo colocar puntos para obtener la curva necesaria. Y al conectar varias curvas podemos obtener prácticamente cualquier cosa.

Aquí tenemos algunos ejemplos:



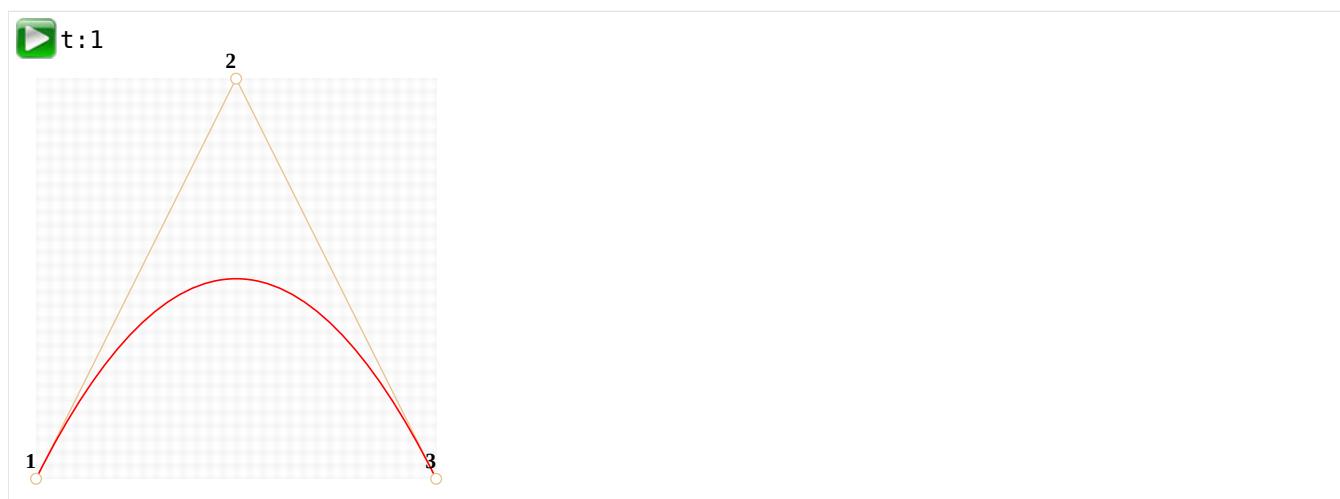
## Algoritmo de De Casteljau

Hay una fórmula matemática para las curvas de Bézier, pero la veremos un poco más tarde, porque el [algoritmo de De Casteljau](#) ↗ es idéntico a la definición matemática y muestra visualmente cómo se construye.

Primero veamos el ejemplo de los 3 puntos.

Aquí está la demostración, y la explicación a continuación.

Los puntos de control (1,2 y 3) se pueden mover con el ratón. Presiona el botón "play" para ejecutarlo.



### El algoritmo de De Casteljau para construir la curva de Bézier de 3 puntos:

1. Dibujar puntos de control. En la demostración anterior están etiquetados: 1, 2, 3.
2. Construir segmentos entre los puntos de control  $1 \rightarrow 2 \rightarrow 3$ . En la demo anterior son marrones.
3. El parámetro  $t$  se mueve de 0 a 1. En el ejemplo de arriba se usa el paso 0.05: el bucle pasa por 0, 0.05, 0.1, 0.15, ... 0.95, 1.

Para cada uno de estos valores de  $t$ :

- En cada segmento marrón tomamos un punto ubicado en la distancia proporcional a  $t$  desde su comienzo. Como hay dos segmentos, tenemos dos puntos.

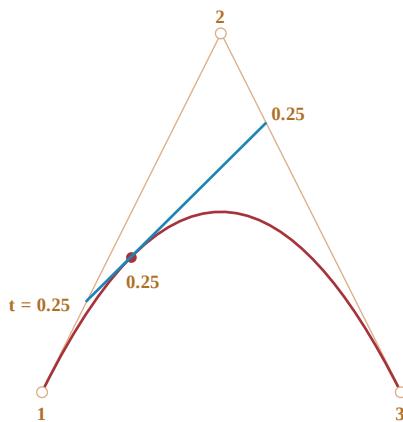
Por ejemplo, para  $t=0$  – ambos puntos estarán al comienzo de los segmentos, y para  $t=0.25$  – en el 25% de la longitud del segmento desde el comienzo, para  $t=0.5$  – 50% (el medio), for  $t=1$  – al final de los segmentos.

- Conecta los puntos. En la imagen de abajo el segmento de conexión está pintado de azul.

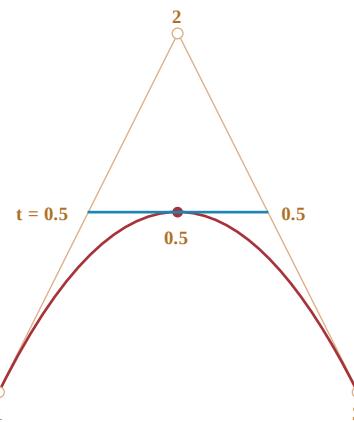
Para  $t=0.25$

Para  $t=0.5$

Para  $t=0.25$



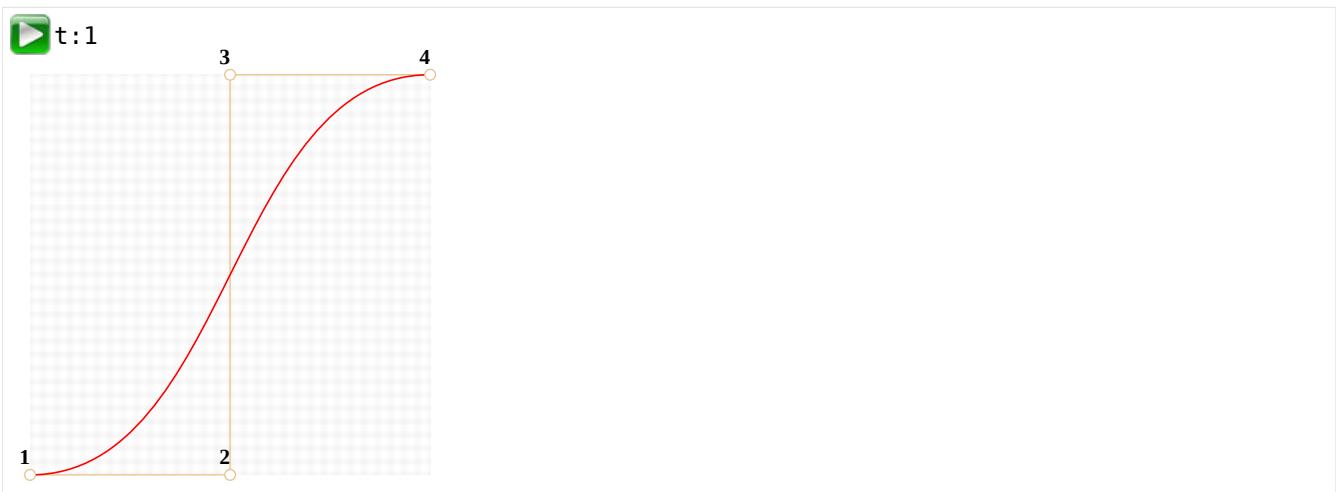
Para  $t=0.5$



4. Ahora, en el segmento azul, toma un punto en la distancia proporcional al mismo valor de  $t$ . Es decir, para  $t=0.25$  (la imagen de la izquierda) tenemos un punto al final del cuarto izquierdo del segmento, y para  $t=0.5$  (la imagen de la derecha) – en la mitad del segmento. En las imágenes de arriba ese punto es rojo.
5. Como  $t$  va de 0 a 1, cada valor de  $t$  añade un punto a la curva. El conjunto de tales puntos forma la curva de Bézier. Es rojo y parabólico en las imágenes de arriba.

Este fue el proceso para 3 puntos. Sería lo mismo para 4 puntos.

La demo para 4 puntos (los puntos se pueden mover con el ratón):



El algoritmo para 4 puntos:

- Conectar puntos de control por segmentos:  $1 \rightarrow 2, 2 \rightarrow 3, 3 \rightarrow 4$ . Habrá 3 segmentos marrones.
- Para cada  $t$  en el intervalo de 0 a 1:
  - Tomamos puntos en estos segmentos en la distancia proporcional a  $t$  desde el principio. Estos puntos están conectados, por lo que tenemos dos segmentos verdes.
  - En estos segmentos tomamos puntos proporcionales a  $t$ . Obtenemos un segmento azul.
  - En el segmento azul tomamos un punto proporcional a  $t$ . En el ejemplo anterior es rojo.
- Estos puntos juntos forman la curva.

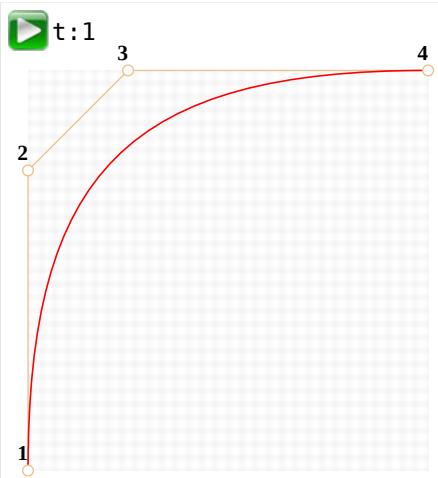
El algoritmo es recursivo y se puede generalizar para cualquier número de puntos de control.

Dados N de puntos de control:

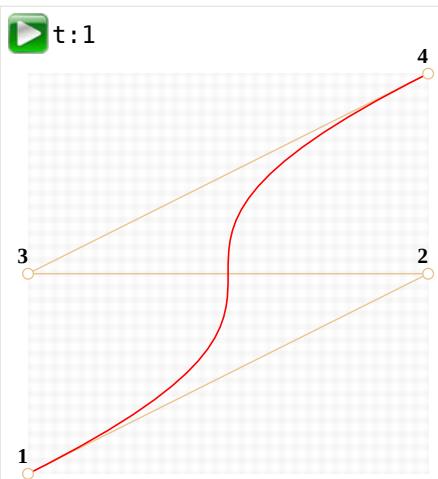
1. Los conectamos para obtener inicialmente  $N-1$  segmentos.
2. Entonces, para cada  $t$  de 0 a 1, tomamos un punto en cada segmento en la distancia proporcional a  $t$  y los conectamos. Habrá  $N-2$  segmentos.
3. Repetimos el paso 2 hasta que solo quede un punto.

Estos puntos forman la curva.

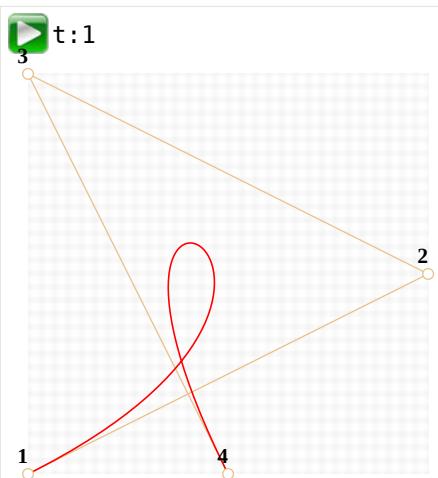
Una curva que se parece a  $y=1/t$ :



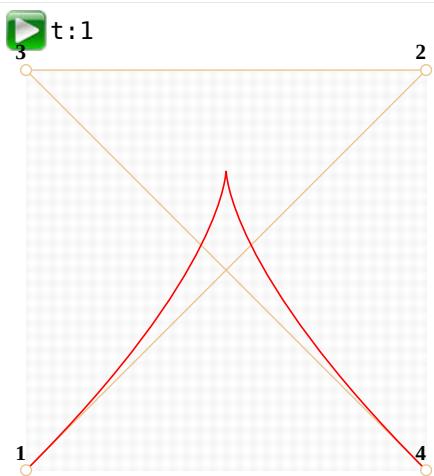
Los puntos de control en zig-zag también funcionan bien:



Es posible hacer un bucle:



Una curva de Bézier no suave (sí, eso también es posible):



Como el algoritmo es recursivo, podemos construir curvas de Bézier de cualquier orden, es decir, usando 5, 6 o más puntos de control. Pero en la práctica muchos puntos son menos útiles. Por lo general, tomamos 2-3 puntos, y para líneas complejas pegamos varias curvas juntas. Eso es más simple de desarrollar y calcular.

#### **i** ¿Cómo dibujar una curva a través de puntos dados?

Para especificar una curva de Bézier se utilizan puntos de control. Como podemos ver, no están en la curva, excepto el primero y el último.

A veces tenemos otra tarea: dibujar una curva *a través de varios puntos*, de modo que todos ellos estén en una sola curva suave. Esta tarea se llama [interpolación](#), y aquí no la cubrimos.

Hay fórmulas matemáticas para tales curvas, por ejemplo el [polinomio de Lagrange](#). En gráficos por ordenador la [interpolación de spline](#) se usa a menudo para construir curvas suaves que conectan muchos puntos.

## Matemáticas

Una curva de Bézier se puede describir usando una fórmula matemática.

Como vimos, en realidad no hay necesidad de saberlo, la mayoría de la gente simplemente dibuja la curva moviendo los puntos con un mouse. Pero si te gustan las matemáticas, aquí están.

Dadas las coordenadas de los puntos de control  $P_i$ : el primer punto de control tiene las coordenadas  $P_1 = (x_1, y_1)$ , el segundo:  $P_2 = (x_2, y_2)$ , y así sucesivamente, las coordenadas de la curva se describen mediante la ecuación que depende del parámetro  $t$  del segmento  $[0, 1]$ .

- La fórmula para una curva de 2 puntos:

$$P = (1-t)P_1 + tP_2$$

- Para 3 puntos de control:

$$P = (1-t)^2P_1 + 2(1-t)tP_2 + t^2P_3$$

- Para 4 puntos de control:

$$P = (1-t)^3P_1 + 3(1-t)^2tP_2 + 3(1-t)t^2P_3 + t^3P_4$$

Estas son las ecuaciones vectoriales. En otras palabras, podemos poner  $x$  e  $y$  en lugar de  $P$  para obtener las coordenadas correspondientes.

Por ejemplo, la curva de 3 puntos está formada por puntos  $(x, y)$  calculados como:

- $x = (1-t)^2x_1 + 2(1-t)tx_2 + t^2x_3$
- $y = (1-t)^2y_1 + 2(1-t)ty_2 + t^2y_3$

En lugar de  $x_1, y_1, x_2, y_2, x_3, y_3$  deberíamos poner coordenadas de 3 puntos de control, y luego a medida que te  $t$  se mueve de  $0$  a  $1$ , para cada valor de  $t$  tendremos  $(x, y)$  de la curva.

Por ejemplo, si los puntos de control son  $(0, 0), (0.5, 1)$  y  $(1, 0)$ , las ecuaciones se convierten en:

- $x = (1-t)^2 * 0 + 2(1-t)t * 0.5 + t^2 * 1 = (1-t)t + t^2 = t$
- $y = (1-t)^2 * 0 + 2(1-t)t * 1 + t^2 * 0 = 2(1-t)t = -2t^2 + 2t$

Ahora como `t` se ejecuta desde `0` a `1`, el conjunto de valores `(x, y)` para cada `t` forman la curva para dichos puntos de control.

## Resumen

Las curvas de Bézier se definen por sus puntos de control.

Vimos dos definiciones de curvas de Bézier:

1. Utilizando un proceso de dibujo: el algoritmo de De Casteljau.
2. Utilizando una fórmula matemática.

Buenas propiedades de las curvas de Bezier:

- Podemos dibujar líneas suaves con un ratón moviendo los puntos de control.
- Las formas complejas se pueden construir con varias curvas Bezier.

Uso:

- En gráficos por ordenador, modelado, editores gráficos vectoriales. Las fuentes están descritas por curvas de Bézier.
- En desarrollo web – para gráficos en Canvas y en formato SVG. Por cierto, los ejemplos “en vivo” de arriba están escritos en SVG. En realidad, son un solo documento SVG que recibe diferentes puntos como parámetros. Puede abrirla en una ventana separada y ver el código fuente: [demo.svg](#).
- En animación CSS para describir la trayectoria y la velocidad de la animación.

## Animaciones CSS

Las animaciones CSS permiten hacer animaciones simples sin JavaScript en absoluto.

Se puede utilizar JavaScript para controlar la animación CSS y mejorarla con un poco de código.

## Transiciones CSS

La idea de las transiciones CSS es simple. Describimos una propiedad y cómo se deberían animar sus cambios. Cuando la propiedad cambia, el navegador pinta la animación.

Es decir: todo lo que necesitamos es cambiar la propiedad, y la transición fluida la hará el navegador.

Por ejemplo, el CSS a continuación anima los cambios de `background-color` durante 3 segundos:

```
.animated {
 transition-property: background-color;
 transition-duration: 3s;
}
```

Ahora, si un elemento tiene la clase `.animated`, cualquier cambio de `background-color` es animado durante 3 segundos.

Haz clic en el botón de abajo para animar el fondo:

```
<button id="color">Haz clic en mi</button>

<style>
#color {
 transition-property: background-color;
 transition-duration: 3s;
}
</style>

<script>
color.onclick = function() {
 this.style.backgroundColor = 'red';
};
</script>
```

```
Haz clic en mí
```

Hay 4 propiedades para describir las transiciones CSS:

- `transition-property`
- `transition-duration`
- `transition-timing-function`
- `transition-delay`

Las cubriremos en un momento, por ahora tengamos en cuenta que la propiedad común `transition` permite declararlas juntas en el orden: `property duration timing-function delay`, y también animar múltiples propiedades a la vez.

Por ejemplo, este botón anima tanto `color` como `font-size`:

```
<button id="growing">Haz clic en mí</button>

<style>
#growing {
 transition: font-size 3s, color 2s;
}
</style>

<script>
growing.onclick = function() {
 this.style.fontSize = '36px';
 this.style.color = 'red';
};
</script>
```

```
Haz clic en mí
```

Ahora cubramos las propiedades de animación una por una.

## transition-property

En `transition-property` escribimos una lista de propiedades para animar, por ejemplo: `left`, `margin-left`, `height`, `color`. O podemos escribir `all`, que significa “animar todas las propiedades”.

No todas las propiedades pueden ser animadas, pero sí [la mayoría de las generalmente usadas ↗](#).

## transition-duration

En `transition-duration` podemos especificar cuánto tiempo debe durar la animación. El tiempo debe estar en [formato de tiempo CSS ↗](#): en segundos `s` o milisegundos `ms`.

## transition-delay

En `transition-delay` podemos especificar el retraso *antes* de la animación. Por ejemplo, si `transition-delay` es `1s` y `transition-duration` es `2s`, la animación comienza después de 1 segundo tras el cambio de la propiedad y la duración total será de 2 segundos.

Los valores negativos también son posibles. De esta manera la animación comienza inmediatamente, pero el punto de inicio de la animación será el del valor dado (tiempo). Por ejemplo, si `transition-delay` es `-1s` y `transition-duration` es `2s`, entonces la animación comienza desde la mitad y la duración total será de 1 segundo.

Aquí la animación cambia los números de `0` a `9` usando la propiedad CSS `translate`:

[https://plnkr.co/edit/3OpQT3blzipKDAAdP?p=preview ↗](https://plnkr.co/edit/3OpQT3blzipKDAAdP?p=preview)

La propiedad `transform` se anima así:

```
#stripe.animate {
 transform: translate(-90%);
```

```
transition-property: transform;
transition-duration: 9s;
}
```

En el ejemplo anterior, JavaScript agrega la clase `.animate` al elemento, y comienza la animación:

```
stripe.classList.add('animate');
```

También podemos comenzar “desde el medio”, desde el número exacto, p. ej. correspondiente al segundo actual, usando el negativo `transition-delay`.

Aquí, si haces clic en el dígito, comienza la animación desde el segundo actual:

[https://plnkr.co/edit/Crob6acjuQzNVyvd?p=preview ↗](https://plnkr.co/edit/Crob6acjuQzNVyvd?p=preview)

JavaScript lo hace con una línea extra:

```
stripe.onclick = function() {
 let sec = new Date().getSeconds() % 10;
 // por ejemplo, -3s aquí comienza la animación desde el 3er segundo
 stripe.style.transitionDelay = '-' + sec + 's';
 stripe.classList.add('animate');
};
```

## transition-timing-function

La función de temporización describe cómo se distribuye el proceso de animación a lo largo del tiempo. Comenzará lentamente y luego irá rápido o viceversa.

Es la propiedad más complicada a primera vista. Pero se vuelve muy simple si le dedicamos un poco de tiempo.

Esa propiedad acepta dos tipos de valores: una curva de Bézier o pasos. Comencemos por la curva, ya que se usa con más frecuencia.

### Curva de Bézier

La función de temporización se puede establecer como una [curva de Bézier](#) con 4 puntos de control que satisfacen las condiciones:

1. Primer punto de control: `(0, 0)`.
2. Último punto de control: `(1, 1)`.
3. Para los puntos intermedios, los valores de `x` deben estar en el intervalo `0..1`, `y` puede ser cualquier cosa.

La sintaxis de una curva de Bézier en CSS: `cubic-bezier(x2, y2, x3, y3)`. Aquí necesitamos especificar solo los puntos de control segundo y tercero, porque el primero está fijado a `(0, 0)` y el cuarto es `(1, 1)`.

La función de temporización determina qué tan rápido ocurre el proceso de animación.

- El eje `x` es el tiempo: `0` – el momento inicial, `1` – el último momento de `transition-duration`.
- El eje `y` especifica la finalización del proceso: `0` – el valor inicial de la propiedad, `1` – el valor final.

La variante más simple es cuando la animación es uniforme, con la misma velocidad lineal. Eso puede especificarse mediante la curva `cubic-bezier(0, 0, 1, 1)`.

Así es como se ve esa curva:



... Como podemos ver, es solo una línea recta. A medida que pasa el tiempo (`x`), la finalización (`y`) de la animación pasa constantemente de `0` a `1`.

El tren, en el ejemplo a continuación, va de izquierda a derecha con velocidad constante (haz clic en él):

[https://plnkr.co/edit/RljV9tj7atlBrDE2?p=preview ↗](https://plnkr.co/edit/RljV9tj7atlBrDE2?p=preview)

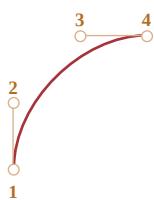
La `transition` de CSS se basa en esa curva:

```
.train {
 left: 0;
 transition: left 5s cubic-bezier(0, 0, 1, 1);
 /* el clic en un tren establece left a 450px, disparando la animación */
}
```

... ¿Y cómo podemos mostrar un tren desacelerando?

Podemos usar otra curva de Bézier: `cubic-bezier(0.0, 0.5, 0.5, 1.0)`.

La gráfica:



Como podemos ver, el proceso comienza rápido: la curva se eleva mucho, y luego más y más despacio.

Aquí está la función de temporización en acción (haz clic en el tren):

[https://plnkr.co/edit/9rlVfsxLkBooVvZI?p=preview ↗](https://plnkr.co/edit/9rlVfsxLkBooVvZI?p=preview)

CSS:

```
.train {
 left: 0;
 transition: left 5s cubic-bezier(0, .5, .5, 1);
 /* el clic en un tren establece left a 450px, disparando la animación */
}
```

Hay varias curvas incorporadas: `linear`, `ease`, `ease-in`, `ease-out` y `ease-in-out`.

La `linear` es una abreviatura de `cubic-bezier(0, 0, 1, 1)` – una línea recta, como ya vimos.

Otros nombres son abreviaturas para la siguiente `cubic-bezier`:

ease *	ease-in	ease-out	ease-in-out
(0.25, 0.1, 0.25, 1.0)	(0.42, 0, 1.0, 1.0)	(0, 0, 0.58, 1.0)	(0.42, 0, 0.58, 1.0)

\* – por defecto, si no hay una función de temporización, se utiliza `ease`.

Por lo tanto, podríamos usar `ease-out` para nuestro tren desacelerando:

```
.train {
 left: 0;
 transition: left 5s ease-out;
 /* igual que transition: left 5s cubic-bezier(0, .5, .5, 1); */
}
```

Pero se ve un poco diferente.

### Una curva de Bézier puede hacer que la animación exceda su rango.

Los puntos de control en la curva pueden tener cualquier coordenada `y`: incluso negativa o enorme. Entonces la curva de Bézier también saltaría muy bajo o alto, haciendo que la animación vaya más allá de su rango normal.

En el siguiente ejemplo, el código de animación es:

```
.train {
 left: 100px;
 transition: left 5s cubic-bezier(.5, -1, .5, 2);
 /* clic en un tren establece left a 400px */
}
```

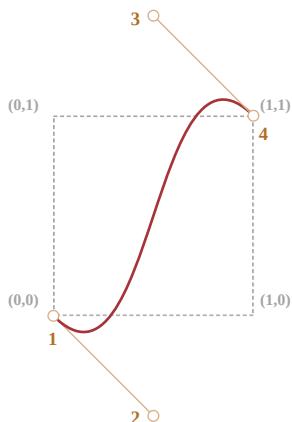
La propiedad `left` debería animarse de `100px` a `400px`.

Pero si haces clic en el tren, verás que:

- Primero, el tren va atrás: `left` llega a ser menor que `100px`.
- Luego avanza, un poco más allá de `400px`.
- Y luego de vuelve a `400px`.

<https://plnkr.co/edit/OmerUCXgNrO8u3va?p=preview>

¿Por qué sucede? es bastante obvio si miramos la gráfica de la curva de Bézier dada:



Movimos la coordenada `y` del segundo punto por debajo de cero, y para el tercer punto lo colocamos sobre `1`, de modo que la curva sale del cuadrante "regular". La `y` está fuera del rango "estándar" `0..1`.

Como sabemos, `y` mide "la finalización del proceso de animación". El valor `y = 0` corresponde al valor inicial de la propiedad y `y = 1` al valor final. Por lo tanto, los valores `y < 0` mueven la propiedad por debajo del `left` inicial y `y > 1` por encima del `left` final.

Esa es una variante "suave" sin duda. Si ponemos valores `y` como `-99` y `99`, entonces el tren saltaría mucho más fuera del rango.

Pero, ¿cómo hacer la curva de Bézier para una tarea específica? Hay muchas herramientas.

- Por ejemplo, podemos hacerlo en el sitio <http://cubic-bezier.com/>.
- El navegador también tiene soporte especial para curvas Bezier en CSS:
  1. Abre las herramientas de desarrollador con `F12` (Mac: `Cmd+Opt+I`).
  2. Selecciona la pestaña `Elementos` y presta atención al subpanel `Estilos`.
  3. Las propiedades CSS que contengan la palabra `cubic-bezier` tendrán un ícono antes de esta palabra.
  4. Haz clic en este ícono para editar la curva.

### Pasos

La función de temporización `steps(number of steps[, start/end])` permite dividir la animación en múltiples pasos.

Veamos eso en un ejemplo con dígitos.

Aquí tenemos una lista de dígitos, sin animaciones, solo como fuente:

[https://plnkr.co/edit/SHMPwMBWcOj5e7VG?p=preview ↗](https://plnkr.co/edit/SHMPwMBWcOj5e7VG?p=preview)

En el HTML, una línea de dígitos está encerrada en un div de largo fijo `<div id="digits">`:

```
<div id="digit">
 <div id="stripe">0123456789</div>
</div>
```

El div `#digit` tiene ancho fijo y un borde, entonces se ve como una ventana roja.

Haremos un temporizador: los dígitos aparecerán uno por uno, de una manera discreta.

Para lograr esto, ocultaremos el `#stripe` fuera de `#digit` usando `overflow: hidden`, y luego desplazamos el `#stripe` a la izquierda paso a paso.

Habrá 9 pasos, un paso para cada dígito:

```
#stripe.animate {
 transform: translate(-90%);
 transition: transform 9s steps(9, start);
}
```

El primer argumento de `steps(9, start)` es el número de pasos. La transformación se dividirá en 9 partes (10% cada una). El intervalo de tiempo también se divide automáticamente en 9 partes, por lo que `transition: 9s` nos da 9 segundos para toda la animación: 1 segundo por dígito.

El segundo argumento es una de dos palabras: `start` o `end`.

El `start` significa que al comienzo de la animación debemos hacer el primer paso de inmediato.

En acción:

[https://plnkr.co/edit/8YXVGHc6jf0G3Fsj?p=preview ↗](https://plnkr.co/edit/8YXVGHc6jf0G3Fsj?p=preview)

Un clic en el dígito lo cambia a `1` (el primer paso) inmediatamente, y luego cambia al comienzo del siguiente segundo.

El proceso está progresando así:

- `0s` – `-10%` (primer cambio al comienzo del primer segundo, inmediatamente)
- `1s` – `-20%`
- ...
- `8s` – `-90%`
- (el último segundo muestra el valor final).

Aquí el primer cambio fue inmediato por el `start` en el `steps`

El valor alternativo 'end' haría que el cambio se aplicara no al principio sino al final de cada segundo.

Entonces el proceso para `steps(9, end)` sería así:

- `0s` – `0` (durante el primer segundo nada cambia)
- `1s` – `-10%` (primer cambio al final del primer segundo)
- `2s` – `-20%`
- ...
- `9s` – `-90%`

Aquí está el `step(9, end)` en acción (observa la pausa antes del primer cambio de dígitos):

[https://plnkr.co/edit/A9eNaFJ1OXrOmmkd?p=preview ↗](https://plnkr.co/edit/A9eNaFJ1OXrOmmkd?p=preview)

También hay algunas formas abreviadas predefinidas para `steps(...)`:

- `step-start` – es lo mismo que `steps(1, start)`. Es decir, la animación comienza de inmediato y toma 1 paso. Entonces comienza y termina inmediatamente, como si no hubiera animación.
- `step-end` – lo mismo que `steps(1, end)`: realiza la animación en un solo paso al final de `transition-duration`.

Estos valores rara vez se usan porque no representan una verdadera animación sino un cambio de un solo paso.

## Evento transitionend

Cuando finaliza la animación CSS, se dispara el evento `transitionend`.

Es ampliamente utilizado para hacer una acción después de que se realiza la animación. También podemos unir animaciones.

Por ejemplo, el barco a continuación comienza a navegar ida y vuelta al hacer clic, cada vez más y más a la derecha:



La animación se inicia mediante la función `go` que se vuelve a ejecutar cada vez que finaliza la transición y cambia la dirección:

```
boat.onclick = function() {
 //...
 let times = 1;

 function go() {
 if (times % 2) {
 // navegar a la derecha
 boat.classList.remove('back');
 boat.style.marginLeft = 100 * times + 200 + 'px';
 } else {
 // navegar a la izquierda
 boat.classList.add('back');
 boat.style.marginLeft = 100 * times - 200 + 'px';
 }
 }

 go();

 boat.addEventListener('transitionend', function() {
 times++;
 go();
 });
};
```

El objeto de evento para `transitionend` tiene pocas propiedades específicas:

### `event.propertyName`

La propiedad que ha terminado de animarse. Puede ser bueno si animamos múltiples propiedades simultáneamente.

### `event.elapsedTime`

El tiempo (en segundos) que duró la animación, sin `transition-delay`.

## Fotogramas clave (Keyframes)

Podemos unir múltiples animaciones simples juntas usando la regla CSS `@keyframes`.

Especifica el “nombre” de la animación y las reglas: qué, cuándo y dónde animar. Luego, usando la propiedad `animation`, adjuntamos la animación al elemento y especificamos parámetros adicionales para él.

Aquí tenemos un ejemplo con explicaciones:

```

<div class="progress"></div>

<style>
 @keyframes go-left-right { /* dale un nombre: "go-left-right" */
 from { left: 0px; } /* animar desde la izquierda: 0px */
 to { left: calc(100% - 50px); } /* animar a la izquierda: 100%-50px */
 }

 .progress {
 animation: go-left-right 3s infinite alternate;
 /* aplicar la animación "go-left-right" al elemento
 duración 3 segundos
 número de veces: infinitas
 alternar la dirección cada vez
 */

 position: relative;
 border: 2px solid green;
 width: 50px;
 height: 20px;
 background: lime;
 }
</style>

```



Hay muchos artículos sobre `@keyframes` y una [especificación detallada ↗](#).

Probablemente no necesitarás `@keyframes` a menudo, a menos que todo esté en constante movimiento en tus sitios.

## Performance

La mayoría de las propiedades CSS pueden ser animadas, porque la mayoría son valores numéricos. Por ejemplo `width`, `color`, `font-size` son todas números. Cuando las animamos, el navegador cambia estos valores gradualmente grama por grama, creando un efecto suave.

Sin embargo, no todas las animaciones se verán tan suaves como quisieras, porque diferentes propiedades CSS tienen diferente costo para cambiar.

En detalles más técnicos, cuando hay un cambio de estilo, el navegador atraviesa 3 pasos para renderizar la nueva vista:

1. **Layout**: (diagrama) recalcula la geometría y posición de cada elemento, luego
2. **Paint**: (dibuja) recalcula cómo debe verse todo en sus lugares, incluyendo `background`, colores,
3. **Composite**: (render) despliega el resultado final en pixels de la pantalla, aplicando transformaciones CSS si existen.

Durante una animación CSS, este proceso se repite para cada frame. Sin embargo las propiedades CSS que nunca afectan geometría o posición, como `color`, pueden saltar el paso “Layout”. Si un `color` cambia, el navegador no recalcula geometría, va a Paint → Composite. Y hay unas pocas propiedades que saltan directo a “Composite”. Puedes encontrar la lista de propiedades CSS y cuáles estados disparan en <https://csstriggers.com> ↗.

Los cálculos pueden tomar un tiempo, especialmente en páginas con muchos elementos y diagramación compleja. Y los retrasos pueden ser notorios en muchos dispositivos, provocando “jitter”: animaciones irregulares, menos fluidas.

La animación de propiedades que salten el paso “Layout” son más rápidas. Mucho mejor si el paso “Paint” se salta también.

La propiedad `transform` es una excelente opción porque:

- CSS transform afecta el elemento objetivo como un todo (rotar, tornar, estirar, desplazar).
- CSS transform nunca afecta a los elementos vecinos.

...entonces los navegadores aplican `transform` “por encima” de “Layout” y “Paint” ya calculados, en el paso “Composite”.

En otras palabras, el navegador calcula la diagramación en la etapa Layout (tamaños, posiciones); lo dibuja con colores, backgrounds, etc., en la etapa “Paint”; y luego aplica `transform` a los elementos que lo necesitan.

Cambios (animaciones) de la propiedad `transform` nunca disparan los pasos Layout y Paint. Aún más, el navegador delega las transformaciones CSS en el acelerador gráfico (un chip especial en la CPU o placa gráfica), haciéndolas muy eficientes.

Afortunadamente la propiedad `transform` es muy poderosa. Usando `transform` en un elemento, puedes rotarlo, darlo vuelta, estirarlo o comprimirlo, desplazarlo y [mucho más ↗](#). Así que en lugar de las propiedades `left/margin-left` podemos usar `transform: translateX(...)`, o usar `transform: scale` para incrementar su tamaño, etc.

La propiedad `opacity` tampoco dispara “Layout” (también se salta “Paint” en Gecko de Mozilla). Podemos usarlo para efectos de mostrar/ocultar o desvanecer/aparecer.

Aparear `transform` con `opacity` puede usualmente resolver la mayoría de nuestras necesidades brindando animaciones vistosas y fluidas.

Aquí por ejemplo, clic en el elemento `#boat` le agrega la clase con `transform: translateX(300)` y `opacity: 0`, haciendo que se mueva `300px` a la derecha y desaparezca:

```


<style>
#boat {
 cursor: pointer;
 transition: transform 2s ease-in-out, opacity 2s ease-in-out;
}

.move {
 transform: translateX(300px);
 opacity: 0;
}
</style>
<script>
 boat.onclick = () => boat.classList.add('move');
</script>
```



Un ejemplo más complejo con `@keyframes`:

```
<h2 onclick="this.classList.toggle('animated')>click me to start / stop</h2>
<style>
.animated {
 animation: hello-goodbye 1.8s infinite;
 width: fit-content;
}
@keyframes hello-goodbye {
 0% {
 transform: translateY(-60px) rotateX(0.7turn);
 opacity: 0;
 }
 50% {
 transform: none;
 opacity: 1;
 }
 100% {
 transform: translateX(230px) rotateZ(90deg) scale(0.5);
 opacity: 0;
 }
}
</style>
```

**click me to start / stop**

## Resumen

Las animaciones CSS permiten animar, suavemente o por pasos, los cambios de una o varias propiedades CSS.

Son buenas para la mayoría de las tareas de animación. También podemos usar JavaScript para animaciones, el siguiente capítulo está dedicado a eso.

Limitaciones de las animaciones CSS en comparación con las animaciones JavaScript:

- + Cosas simples hechas simplemente.
- + Rápido y ligero para la CPU.
- Las animaciones de JavaScript son flexibles. Pueden implementar cualquier lógica de animación, como una "explosión".
- No solo cambios de propiedad. Podemos crear nuevos elementos en JavaScript para fines de animación.

En los ejemplos de este artículo animamos `font-size`, `left`, `width`, `height`, etc. En proyectos de la vida real es preferible usar `transform: scale()` y `transform: translate()` para obtener mejor performance.

La mayoría de las animaciones se pueden implementar usando CSS como se describe en este capítulo. Y el evento `transitionend` permite ejecutar JavaScript después de la animación, por lo que se integra bien con el código.

Pero en el próximo capítulo haremos algunas animaciones en JavaScript para cubrir casos más complejos.

## ✓ Tareas

### Animar un avión (CSS)

importancia: 5

Muestra la animación como en la imagen a continuación (haz clic en el avión):



- La imagen crece al hacer clic de `40x24px` a `400x240px` (10 veces más grande).
- La animación dura 3 segundos.
- Al final muestra: “¡Listo!”.
- Durante el proceso de animación, puede haber más clics en el avión. No deberían “romper” nada.

[Abrir un entorno controlado para la tarea.](#) ↗

[A solución](#)

### Animar el avión volando (CSS)

importancia: 5

Modifica la solución de la tarea anterior [Animar un avión \(CSS\)](#) para hacer que el avión crezca más que su tamaño original `400x240px` (saltar fuera), y luego vuelva a ese tamaño.

Así es como debería verse (haz clic en el avión):



Toma la solución de la tarea anterior como punto de partida.

[A solución](#)

## Círculo animado

importancia: 5

Crea una función `showCircle(cx, cy, radius)` que muestre un círculo animado creciendo.

- `cx, cy` son coordenadas relativas a la ventana del centro del círculo,
- `radius` es el radio del círculo.

Haz clic en el botón de abajo para ver cómo debería verse:

`showCircle(150, 150, 100)`

El documento fuente tiene un ejemplo de un círculo con estilos correctos, por lo que la tarea es precisamente hacer la animación correctamente.

[Abrir un entorno controlado para la tarea.](#) ↗

[A solución](#)

## Círculo animado con función de callback

En la tarea [Círculo animado](#) se muestra un círculo creciente animado.

Ahora digamos que necesitamos no solo un círculo, sino mostrar un mensaje dentro de él. El mensaje debería aparecer después de que la animación esté completa (el círculo es desarrollado completamente), de lo contrario se vería feo.

En la solución de la tarea, la función `showCircle(cx, cy, radius)` dibuja el círculo, pero no hay forma de saber cuando lo termina.

Agrega un argumento `callback`: `showCircle(cx, cy, radius, callback)` que se llamará cuando se complete la animación. El `callback` debería recibir el círculo `<div>` como argumento.

Aquí el ejemplo:

`showCircle(150, 150, 100, div => {`

```
div.classList.add('message-ball');
div.append("Hola, mundo!");
});
```

Demostración:

Pruébame

Toma la solución de la tarea [Círculo animado](#) como base.

[A solución](#)

## Animaciones JavaScript

Las animaciones de JavaScript pueden manejar cosas que CSS no puede.

Por ejemplo, moverse a lo largo de una ruta compleja, con una función de sincronización diferente a las curvas de Bézier, o una animación en un canvas.

### Usando setInterval

Una animación se puede implementar como una secuencia de frames, generalmente pequeños cambios en las propiedades de HTML/CSS.

Por ejemplo, cambiar `style.left` de `0px` a `100px` mueve el elemento. Y si lo aumentamos en `setInterval`, cambiando en `2px` con un pequeño retraso, como 50 veces por segundo, entonces se ve suave. Ese es el mismo principio que en el cine: 24 frames por segundo son suficientes para que se vea suave.

El pseudocódigo puede verse así:

```
let timer = setInterval(function() {
 if (animation complete) clearInterval(timer);
 else increase style.left by 2px
}, 20); // cambiar en 2px cada 20ms, aproximadamente 50 frames por segundo
```

Ejemplo más completo de la animación:

```
let start = Date.now(); // recordar la hora de inicio

let timer = setInterval(function() {
 // ¿Cuánto tiempo pasó desde el principio?
 let timePassed = Date.now() - start;

 if (timePassed >= 2000) {
 clearInterval(timer); // terminar la animación después de 2 segundos
 return;
 }

 // dibujar la animación en el momento timePassed
 draw(timePassed);

}, 20);

// mientras timePassed va de 0 a 2000
// left obtiene valores de 0px a 400px
function draw(timePassed) {
```

```
 train.style.left = timePassed / 5 + 'px';
}
```

Haz clic para ver la demostración:

<https://plnkr.co/edit/mpxSFrY9Z060JY3w?p=preview>

## Usando requestAnimationFrame

Imaginemos que tenemos varias animaciones ejecutándose simultáneamente.

Si las ejecutamos por separado, aunque cada una tenga `setInterval(..., 20)`, el navegador tendría que volver a pintar con mucha más frecuencia que cada `20ms`.

Eso es porque tienen un tiempo de inicio diferente, por lo que “cada 20ms” difiere entre las diferentes animaciones. Los intervalos no están alineados. Así que tendremos varias ejecuciones independientes dentro de `20ms`.

En otras palabras, esto:

```
setInterval(function() {
 animate1();
 animate2();
 animate3();
}, 20)
```

...Es más ligero que tres llamadas independientes:

```
setInterval/animate1, 20); // animaciones independientes
setInterval/animate2, 20); // en diferentes lugares del script
setInterval/animate3, 20);
```

Estos varios redibujos independientes deben agruparse para facilitar el redibujado al navegador y, por lo tanto, cargar menos CPU y verse más fluido.

Hay una cosa más a tener en cuenta. A veces, cuando el CPU está sobrecargado, o hay otras razones para volver a dibujar con menos frecuencia (como cuando la pestaña del navegador está oculta), no deberíamos ejecutarlo cada `20ms`.

Pero, ¿cómo sabemos eso en JavaScript? Hay una especificación [Sincronización de animación](#) que proporciona la función `requestAnimationFrame`. Aborda todos estos problemas y aún más.

La sintaxis:

```
let requestId = requestAnimationFrame(callback)
```

Eso programa la función `callback` para que se ejecute en el tiempo más cercano cuando el navegador quiera hacer una animación.

Si hacemos cambios en los elementos dentro de `callback`, entonces se agruparán con otros callbacks de `requestAnimationFrame` y con animaciones CSS. Así que habrá un recálculo y repintado de geometría en lugar de muchos.

El valor devuelto `requestId` se puede utilizar para cancelar la llamada:

```
// cancelar la ejecución programada del callback
cancelAnimationFrame(requestId);
```

El `callback` obtiene un argumento: el tiempo transcurrido desde el inicio de la carga de la página en microsegundos. Este tiempo también se puede obtener llamando a [performance.now\(\)](#).

Por lo general, el `callback` se ejecuta muy pronto, a menos que el CPU esté sobrecargado o la batería de la laptop esté casi descargada, o haya otra razón.

El siguiente código muestra el tiempo entre las primeras 10 ejecuciones de `requestAnimationFrame`. Por lo general, son 10-20ms:

```

<script>
 let prev = performance.now();
 let times = 0;

 requestAnimationFrame(function measure(time) {
 document.body.insertAdjacentHTML("beforeEnd", Math.floor(time - prev) + " ");
 prev = time;

 if (times++ < 10) requestAnimationFrame(measure);
 })
</script>

```

## Animación estructurada

Ahora podemos hacer una función de animación más universal basada en `requestAnimationFrame`:

```

function animate({timing, draw, duration}) {

 let start = performance.now();

 requestAnimationFrame(function animate(time) {
 // timeFraction va de 0 a 1
 let timeFraction = (time - start) / duration;
 if (timeFraction > 1) timeFraction = 1;

 // calcular el estado actual de la animación
 let progress = timing(timeFraction)

 draw(progress); // dibujar

 if (timeFraction < 1) {
 requestAnimationFrame(animate);
 }
 });
}

```

La función `animate` acepta 3 parámetros que básicamente describen la animación:

### `duration`

Tiempo total de animación. Como: `1000`.

### `timing(timeFraction)`

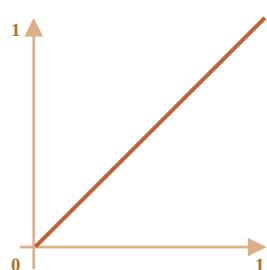
Función de sincronización, como la propiedad CSS `transition-timing-function` que obtiene la fracción de tiempo que pasó (`0` al inicio, `1` al final) y devuelve la finalización de la animación (como `y` en la curva de Bézier).

Por ejemplo, una función lineal significa que la animación continúa uniformemente con la misma velocidad:

```

function linear(timeFraction) {
 return timeFraction;
}

```



Su gráfico:

Eso es como `transition-timing-function: linear`. A continuación se muestran variantes más interesantes.

### `draw(progress)`

La función que toma el estado de finalización de la animación y la dibuja. El valor `progress=0` denota el estado inicial de la animación y `progress=1` – el estado final.

Esta es la función que realmente dibuja la animación.

Puede mover el elemento:

```
function draw(progress) {
 train.style.left = progress + 'px';
}
```

...O hacer cualquier otra cosa, podemos animar cualquier cosa, de cualquier forma.

Vamos a animar el elemento `width` de `0` a `100%` usando nuestra función.

Haz clic en el elemento de la demostración:

[https://plnkr.co/edit/W3U78xZC38RKf7Ry?p=preview ↗](https://plnkr.co/edit/W3U78xZC38RKf7Ry?p=preview)

El código para ello:

```
animate({
 duration: 1000,
 timing(timeFraction) {
 return timeFraction;
 },
 draw(progress) {
 elem.style.width = progress * 100 + '%';
 }
});
```

A diferencia de la animación CSS, aquí podemos hacer cualquier función de sincronización y cualquier función de dibujo. La función de sincronización no está limitada por las curvas de Bézier. Y `draw` puede ir más allá de las propiedades, crear nuevos elementos para la animación de fuegos artificiales o algo así.

## Funciones de sincronización

Vimos arriba la función de sincronización lineal más simple.

Veamos más de ellas. Intentaremos animaciones de movimiento con diferentes funciones de sincronización para ver cómo funcionan.

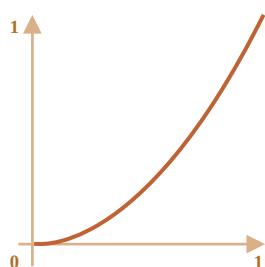
### Potencia de n

Si queremos acelerar la animación, podemos usar `progress` en la potencia `n`.

Por ejemplo, una curva parabólica:

```
function quad(timeFraction) {
 return Math.pow(timeFraction, 2)
}
```

La gráfica:

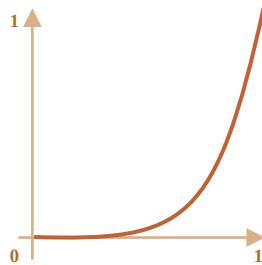


Veo en acción (haz clic para activar):



...O la curva cúbica o incluso mayor  $n$ . Aumentar la potencia hace que se acelere más rápido.

Aquí está el gráfico de `progress` en la potencia 5:



En acción:

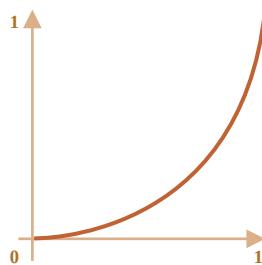


## El arco

Función:

```
function circ(timeFraction) {
 return 1 - Math.sin(Math.acos(timeFraction));
}
```

La gráfica:



## Back: tiro con arco

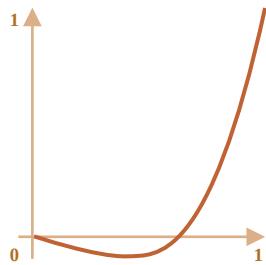
Esta función realiza el “tiro con arco”. Primero “tiramos de la cuerda del arco”, y luego “disparamos”.

A diferencia de las funciones anteriores, depende de un parámetro adicional  $x$ , el “coeficiente de elasticidad”. La distancia de “tirar de la cuerda del arco” está definida por él.

El código:

```
function back(x, timeFraction) {
 return Math.pow(timeFraction, 2) * ((x + 1) * timeFraction - x)
}
```

The graph for  $x = 1.5$ :



Para la animación lo usamos con un valor específico de `x`. Ejemplo de `x = 1.5`:



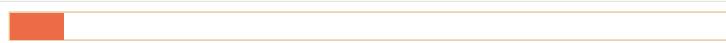
### Rebotar

Imagina que dejamos caer una pelota. Se cae, luego rebota unas cuantas veces y se detiene.

La función `bounce` hace lo mismo, pero en orden inverso: el “rebote” comienza inmediatamente. Utiliza algunos coeficientes especiales para eso:

```
function bounce(timeFraction) {
 for (let a = 0, b = 1; 1; a += b, b /= 2) {
 if (timeFraction >= (7 - 4 * a) / 11) {
 return -Math.pow((11 - 6 * a - 11 * timeFraction) / 4, 2) + Math.pow(b, 2)
 }
 }
}
```

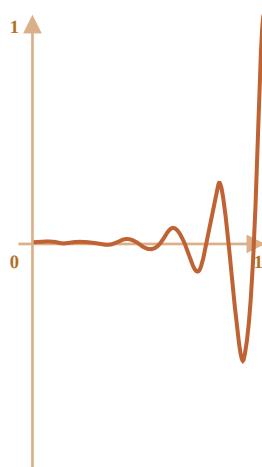
En acción:



### Animación elástica

Una función “elástica” más que acepta un parámetro adicional `x` para el “rango inicial”.

```
function elastic(x, timeFraction) {
 return Math.pow(2, 10 * (timeFraction - 1)) * Math.cos(20 * Math.PI * x / 3 * timeFraction)
}
```



La gráfica para `x=1.5`:

En acción para `x=1.5`:



### Inversión: ease\*

Entonces tenemos una colección de funciones de sincronización. Su aplicación directa se llama “easyIn”.

A veces necesitamos mostrar la animación en orden inverso. Eso se hace con la transformación “easyOut”.

### easeOut

En el modo “easyOut”, la función de sincronización se coloca en un wrapper `timingEaseOut`:

```
timingEaseOut(timeFraction) = 1 - timing(1 - timeFraction)
```

En otras palabras, tenemos una función de “transformación” `makeEaseOut` que toma una función de sincronización “regular” y devuelve el wrapper envolviéndola:

```
// acepta una función de sincronización, devuelve la variante transformada
function makeEaseOut(timing) {
 return function(timeFraction) {
 return 1 - timing(1 - timeFraction);
}
```

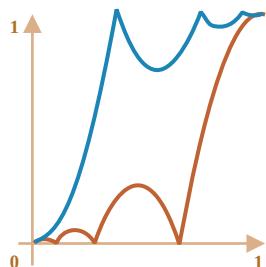
Por ejemplo, podemos tomar la función `bounce` descrita anteriormente y aplicarla:

```
let bounceEaseOut = makeEaseOut(bounce);
```

Entonces el rebote no estará al principio, sino al final de la animación. Se ve aún mejor:

<https://plnkr.co/edit/v6NSfMa0ypglZabT?p=preview>

Aquí podemos ver cómo la transformación cambia el comportamiento de la función:



Si hay un efecto de animación al principio, como rebotar, se mostrará al final.

En el gráfico anterior, el **rebote regular** tiene el color rojo y el **rebote easyOut** es azul.

- Rebote regular: el objeto rebota en la parte inferior y luego, al final, salta bruscamente hacia la parte superior.
- Después de `easyOut` – primero salta a la parte superior, luego rebota allí.

### easeInOut

También podemos mostrar el efecto tanto al principio como al final de la animación. La transformación se llama “easyInOut”.

Dada la función de tiempo, calculamos el estado de la animación de la siguiente manera:

```
if (timeFraction <= 0.5) { // primera mitad de la animación
 return timing(2 * timeFraction) / 2;
} else { // segunda mitad de la animación
 return (2 - timing(2 * (1 - timeFraction))) / 2;
}
```

El código wrapper:

```
function makeEaseInOut(timing) {
 return function(timeFraction) {
 if (timeFraction < .5)
 return timing(2 * timeFraction) / 2;
 else
 return (2 - timing(2 * (1 - timeFraction))) / 2;
```

```

 }
}

bounceEaseInOut = makeEaseInOut(bounce);

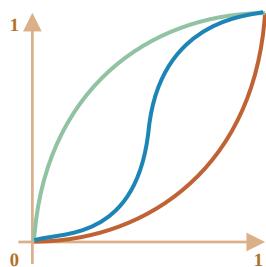
```

En acción, `bounceEaseInOut`:

[https://plnkr.co/edit/Vh0Q3ShC7IOHEyQN?p=preview ↗](https://plnkr.co/edit/Vh0Q3ShC7IOHEyQN?p=preview)

La transformación “`easyInOut`” une dos gráficos en uno: `easyIn` (regular) para la primera mitad de la animación y `easyOut` (invertido) – para la segunda parte.

El efecto se ve claramente si comparamos las gráficas de `easyIn`, `easyOut` y `easyInOut` de la función de sincronización `circ`:



- Rojo es la variante regular de `circ` (`easeIn`).
- Verde – `easeOut`.
- Azul – `easeInOut`.

Como podemos ver, el gráfico de la primera mitad de la animación es el `easyIn` reducido y la segunda mitad es el `easyOut` reducido. Como resultado, la animación comienza y termina con el mismo efecto.

## “Dibujar” más interesante

En lugar de mover el elemento podemos hacer otra cosa. Todo lo que necesitamos es escribir la función `draw` adecuada.

Aquí está la escritura de texto animada “rebotando”:

[https://plnkr.co/edit/pzNmYk0GHZZgqXhw?p=preview ↗](https://plnkr.co/edit/pzNmYk0GHZZgqXhw?p=preview)

## Resumen

Para animaciones que CSS no puede manejar bien, o aquellas que necesitan un control estricto, JavaScript puede ayudar. Las animaciones de JavaScript deben implementarse a través de `requestAnimationFrame`. Ese método integrado permite configurar una función callback para que se ejecute cuando el navegador esté preparando un repaintido. Por lo general, es muy pronto, pero el tiempo exacto depende del navegador.

Cuando una página está en segundo plano, no se repinta en absoluto, por lo que el callback no se ejecutará: la animación se suspenderá y no consumirá recursos. Eso es genial.

Aquí está la función auxiliar `animate` para configurar la mayoría de las animaciones:

```

function animate({timing, draw, duration}) {
 let start = performance.now();

 requestAnimationFrame(function animate(time) {
 // timeFraction va de 0 a 1
 let timeFraction = (time - start) / duration;
 if (timeFraction > 1) timeFraction = 1;

 // calcular el estado actual de la animación
 let progress = timing(timeFraction);

 draw(progress); // dibujar

 if (timeFraction < 1) {

```

```
 requestAnimationFrame/animate);
}

});
```

Opciones:

- `duration` – el tiempo total de animación en ms.
- `timing` – la función para calcular el progreso de la animación. Obtiene una fracción de tiempo de 0 a 1, devuelve el progreso de la animación, generalmente de 0 a 1.
- `draw` – la función para dibujar la animación.

Seguramente podríamos mejorarlo, agregar más campanas y silbidos, pero las animaciones de JavaScript no se aplican a diario. Se utilizan para hacer algo interesante y no estándar. Por lo tanto, querrás agregar las funciones que necesitas cuando las necesites.

Las animaciones JavaScript pueden utilizar cualquier función de sincronización. Cubrimos muchos ejemplos y transformaciones para hacerlos aún más versátiles. A diferencia de CSS, aquí no estamos limitados a las curvas de Bézier.

Lo mismo ocurre con `draw`: podemos animar cualquier cosa, no solo propiedades CSS.

## ✓ Tareas

### Animar la pelota que rebota

importancia: 5

Haz una pelota que rebote. Haz clic para ver cómo debería verse:



[Abrir un entorno controlado para la tarea.](#) ↗

[A solución](#)

### Animar la pelota rebotando hacia la derecha

importancia: 5

Haz que la pelota rebote hacia la derecha. Así:



Escribe el código de la animación. La distancia a la izquierda es `100px`.

Toma la solución de la tarea anterior [Animar la pelota que rebota](#) como fuente.

## Componentes Web

Los componentes web son un conjunto de estándares para crear componentes autónomos: elementos HTML personalizados con sus propias propiedades y métodos, DOM encapsulado y estilos.

### Desde la altura orbital

En esta sección se describe un conjunto de normas modernas para los “web components”.

En la actualidad, estos estándares están en desarrollo. Algunas características están bien apoyadas e integradas en el standard moderno HTML/DOM, mientras que otras están aún en fase de borrador. Puedes probar algunos ejemplos en cualquier navegador, Google Chrome es probablemente el que más actualizado esté con estas características. Suponemos que eso se debe a que los compañeros de Google están detrás de muchas de las especificaciones relacionadas.

### Lo que es común entre...

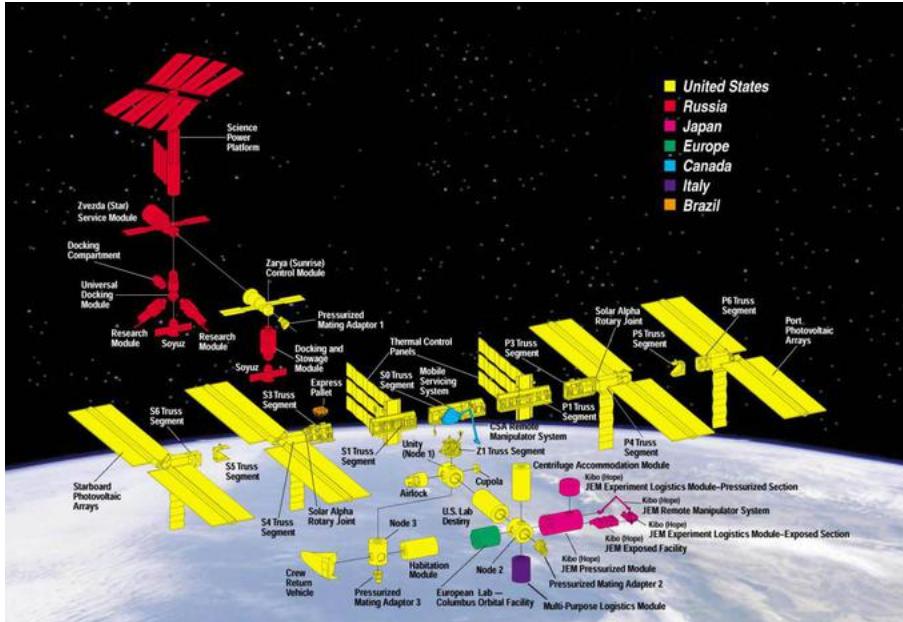
La idea del componente completo no es nada nuevo. Se usa en muchos frameworks y en otros lugares.

Antes de pasar a los detalles de implementación, echemos un vistazo a este gran logro de la humanidad:



Esa es la Estación Espacial Internacional (EEI).

Y así es como se ha montado (aproximadamente):



### La Estación Espacial Internacional:

- Está formada por muchos componentes.
- Cada componente, a su vez, tiene muchos detalles más pequeños en su interior.
- Los componentes son muy complejos, mucho más complicados que la mayoría de los sitios web.
- Los componentes han sido desarrollados internacionalmente, por equipos de diferentes países, que hablan diferentes idiomas.

...Y esta cosa vuela, ¡mantiene a los humanos vivos en el espacio!

¿Cómo se crean dispositivos tan complejos?

¿Qué principios podríamos tomar prestados para que nuestro desarrollo sea fiable y escalable a ese nivel? ¿O, al menos, cerca de él?

### Arquitectura de componentes

La regla más conocida para desarrollar software complejo es: no hacer software complejo.

Si algo se vuelve complejo – divídalo en partes más simples y conéctalas de la manera más obvia.

**Un buen arquitecto es el que puede hacer lo complejo simple.**

Podemos dividir la interfaz de usuario en componentes visuales: cada uno de ellos tiene su propio lugar en la página, puede "hacer" una tarea bien descrita, y está separado de los demás.

Echemos un vistazo a un sitio web, por ejemplo Twitter.

Naturalmente está dividido en componentes:

The screenshot shows the Twitter homepage with several numbered callouts highlighting different components:

1. Top navigation bar with icons for notifications, messages, and search, followed by a 'Tweet' button.
2. User profile card for Ilya Kantor (@iliakan) showing tweets (240), following (35), and followers (2,263).
3. 'Who to follow' sidebar with links to Node.js, GitHub, and Brackets.
4. 'What's happening?' feed header.
5. A tweet from Andrea Giammarchi (@WebReflection) about the status of web extension submissions.
6. A retweet from Jason Karns (@jasonkarns) about buying a mechanical keyboard.
7. A retweet from Guillermo Álvarez (@zeithq) about learning to use Styled JSX in Next.js.

1. Navegación superior.
2. Información usuario.
3. Sugerencias de seguimiento.
4. Envío de formulario.
5. (y también 6, 7) – mensajes.

Los componentes pueden tener subcomponentes, p.ej. los mensajes pueden ser parte de un componente “lista de mensajes” de nivel superior. Una imagen de usuario en sí puede ser un componente, y así sucesivamente.

¿Cómo decidimos qué es un componente? Eso viene de la intuición, la experiencia y el sentido común. Normalmente es una entidad visual separada que podemos describir en términos de lo que hace y cómo interactúa con la página. En el caso anterior, la página tiene bloques, cada uno de ellos juega su propio papel, es lógico crear esos componentes.

Un componente tiene:

- Su propia clase de JavaScript.
- La estructura DOM, gestionada únicamente por su clase, el código externo no accede a ella (principio de “encapsulación”).
- Estilos CSS, aplicados al componente.
- API: eventos, métodos de clase etc, para interactuar con otros componentes.

Una vez más, todo el asunto del “componente” no es nada especial.

Existen muchos frameworks y metodologías de desarrollos para construirlos, cada uno con sus propias características y reglas. Normalmente, se utilizan clases y convenciones CSS para proporcionar la “sensación de componente” – alcance de CSS y encapsulación de DOM.

“Web components” proporcionan capacidades de navegación incorporadas para eso, así que ya no tenemos que emularlos.

- [Custom elements ↗](#) – para definir elementos HTML personalizados.
- [Shadow DOM ↗](#) – para crear un DOM interno para el componente, oculto a los demás componentes.
- [CSS Scoping ↗](#) – para declarar estilos que sólo se aplican dentro del Shadow DOM del componente.
- [Event retargeting ↗](#) y otras cosas menores para hacer que los componentes se ajusten mejor al desarrollo.

En el próximo capítulo entraremos en detalles en los “Custom Elements” – la característica fundamental y bien soportada de los componentes web, buena por sí misma.

## Elementos personalizados

Podemos crear elementos HTML personalizados con nuestras propias clases; con sus propios métodos, propiedades, eventos y demás.

Una vez que definimos el elemento personalizado, podemos usarlo a la par de elementos HTML nativos.

Esto es grandioso, porque el diccionario HTML es rico, pero no infinito. No hay `<aletas-faciles>`, `<gira-carrusel>`, `<bella-descarga>` ... Solo piensa en cualquier otra etiqueta que puedas necesitar.

Podemos definirlos con una clase especial, y luego usarlos como si siempre hubieran sido parte del HTML.

Hay dos clases de elementos personalizados:

1. **Elementos personalizados autónomos** – son elementos “todo-nuevo”, extensiones de la clase abstracta `HTMLElement`.
2. **Elementos nativos personalizados** – son extensiones de elementos nativos, por ejemplo un botón personalizado basado en `HTMLButtonElement`.

Primero cubriremos los elementos autónomos, luego pasaremos a la personalización de elementos nativos.

Para crear un elemento personalizado, necesitamos decirle al navegador varios detalles acerca de él: cómo mostrarlo, qué hacer cuando el elemento es agregado o quitado de la página, etc.

Eso se logra creando una clase con métodos especiales. Es fácil, son unos pocos métodos y todos ellos son opcionales.

Este es el esquema con la lista completa:

```
class MyElement extends HTMLElement {
 constructor() {
 super();
 // elemento creado
 }
}
```

```

}

connectedCallback() {
 // el navegador llama a este método cuando el elemento es agregado al documento
 // (puede ser llamado varias veces si un elemento es agregado y quitado repetidamente)
}

disconnectedCallback() {
 // el navegador llama a este método cuando el elemento es quitado del documento
 // (puede ser llamado varias veces si un elemento es agregado y quitado repetidamente)
}

static get observedAttributes() {
 return /* array de nombres de atributos a los que queremos monitorear por cambios */;
}

attributeChangedCallback(name, oldValue, newValue) {
 // es llamado cuando uno de los atributos listados arriba es modificado
}

adoptedCallback() {
 // es llamado cuando el elemento es movido a un nuevo documento
 // (ocurre en document.adoptNode, muy raramente usado)
}

// puede haber otros métodos y propiedades de elemento
}

```

Después de ello, necesitamos registrar el elemento:

```
// hacer saber al navegador que <my-element> es servido por nuestra nueva clase
customElements.define("my-element", MyElement);
```

A partir de ello, para cada elemento HTML con la etiqueta `<my-element>` se crea una instancia de `MyElement` y se llaman los métodos mencionados. También podemos insertarlo con JavaScript: `document.createElement('my-element')`.

**i Los nombres de los elementos personalizados deben incluir un guion -**

Los elementos personalizados deben incluir un guion corto `-` en su nombre. Por ejemplo, `my-element` y `super-button` son nombres válidos, pero `myelement` no lo es.

Esto se hace para asegurar que no haya conflicto de nombres entre los elementos nativos y los personalizados.

## Ejemplo: “time-formatted”

Ya existe un elemento `<time>` en HTML para presentar fecha y hora, pero este no hace ningún formateo por sí mismo.

Construyamos el elemento `<time-formatted>` que muestre la hora en un bonito formato y reconozca la configuración de lengua local:

```

<script>
class TimeFormatted extends HTMLElement { // (1)

 connectedCallback() {
 let date = new Date(this.getAttribute('datetime') || Date.now());

 this.innerHTML = new Intl.DateTimeFormat("default", {
 year: this.getAttribute('year') || undefined,
 month: this.getAttribute('month') || undefined,
 day: this.getAttribute('day') || undefined,
 hour: this.getAttribute('hour') || undefined,
 minute: this.getAttribute('minute') || undefined,
 second: this.getAttribute('second') || undefined,
 timeZoneName: this.getAttribute('time-zone-name') || undefined,
 }).format(date);
 }
}

```

```

customElements.define("time-formatted", TimeFormatted); // (2)
</script>

<!-- (3) -->
<time-formatted datetime="2019-12-01"
 year="numeric" month="long" day="numeric"
 hour="numeric" minute="numeric" second="numeric"
 time-zone-name="short"
></time-formatted>

```

December 1, 2019, 3:00:00 AM GMT+3

1. La clase tiene un solo método, `connectedCallback()`, que es llamado por el navegador cuando se agrega el elemento `<time-formatted>` a la página o cuando el analizador HTML lo detecta. Este método usa el formateador de datos nativo [Intl.DateTimeFormat](#), bien soportado por los navegadores, para mostrar una agradable hora formateada.
2. Necesitamos registrar nuestro nuevo elemento con `customElements.define(tag, class)`.
3. Y podremos usarlo por doquier.

#### i Actualización de elementos personalizados

Si el navegador encuentra algún `<time-formatted>` antes de `customElements.define`, no es un error. Pero el elemento es todavía desconocido, como cualquier etiqueta no estándar.

Tal elemento “undefined” puede ser estilizado con el selector CSS `:not(:defined)`.

Una vez que `customElement.define` es llamado, estos elementos son “actualizados”: para cada elemento, una nueva instancia de `TimeFormatted` es creada y `connectedCallback` es llamado. Se vuelven `:defined`.

Para obtener información acerca de los elementos personalizados, tenemos los métodos:

- `customElements.get(name)` – devuelve la clase del elemento personalizado con el `name` dado,
- `customElements.whenDefined(name)` – devuelve una promesa que se resuelve (sin valor) cuando un elemento personalizado con el `name` dado se vuelve `defined`.

#### i Renderizado en `connectedCallback`, no en el constructor

En el ejemplo de arriba, el contenido del elemento es renderizado (construido) en `connectedCallback`.

¿Por qué no en el `constructor`?

La razón es simple: cuando el `constructor` es llamado, es aún demasiado pronto. El elemento es creado, pero el navegador aún no procesó ni asignó atributos en este estado, entonces las llamadas a `getAttribute` devolverían `null`. Así que no podemos renderizar ahora.

Por otra parte, si lo piensas, es más adecuado en términos de performance: demorar el trabajo hasta que realmente se lo necesite.

El `connectedCallback` se dispara cuando el elemento es agregado al documento. No apenas agregado a otro elemento como hijo, sino cuando realmente se vuelve parte de la página. Así podemos construir un DOM separado, crear elementos y prepararlos para uso futuro. Ellos serán realmente renderizados una vez que estén dentro de la página.

## Observando atributos

En la implementación actual de `<time-formatted>`, después de que el elemento fue renderizado, cambios posteriores en sus atributos no tendrán ningún efecto. Eso es extraño para un elemento HTML, porque cuando cambiamos un atributo (como en `a.href`) esperamos que dicho cambio sea visible de inmediato. Corrijamos esto.

Podemos observar atributos suministrando la lista de ellos al getter estático `observedAttributes()`. Cuando esos atributos son modificados, se dispara `attributeChangedCallback`. No se dispara para los atributos no incluidos en la lista, por razones de performance.

A continuación, el nuevo `<time-formatted>` que se actualiza cuando los atributos cambian:

```

<script>
class TimeFormatted extends HTMLElement {
 render() { // (1)

```

```

let date = new Date(this.getAttribute('datetime') || Date.now());

this.innerHTML = new Intl.DateTimeFormat("default", {
 year: this.getAttribute('year') || undefined,
 month: this.getAttribute('month') || undefined,
 day: this.getAttribute('day') || undefined,
 hour: this.getAttribute('hour') || undefined,
 minute: this.getAttribute('minute') || undefined,
 second: this.getAttribute('second') || undefined,
 timeZoneName: this.getAttribute('time-zone-name') || undefined,
}).format(date);
}

connectedCallback() { // (2)
 if (!this.rendered) {
 this.render();
 this.rendered = true;
 }
}

static get observedAttributes() { // (3)
 return ['datetime', 'year', 'month', 'day', 'hour', 'minute', 'second', 'time-zone-name'];
}

attributeChangedCallback(name, oldValue, newValue) { // (4)
 this.render();
}

}

customElements.define("time-formatted", TimeFormatted);
</script>

<time-formatted id="elem" hour="numeric" minute="numeric" second="numeric"></time-formatted>

<script>
setInterval(() => elem.setAttribute('datetime', new Date()), 1000); // (5)
</script>

```

5:12:43 AM

1. La lógica de renderizado fue movida al método ayudante `render()`.
2. Lo llamamos una vez cuando el elemento es insertado en la página.
3. Al cambiar un atributo listado en `observedAttributes()`, se dispara `attributeChangedCallback`.
4. ...y se re-renderiza el elemento.
5. Como resultado, ahora podemos crear un reloj dinámico con facilidad.

## Orden de renderizado

Cuando el “parser” construye el DOM, los elementos son procesados uno tras otro, padres antes que hijos. Por ejemplo si tenemos `<outer><inner></inner></outer>`, el elemento `<outer>` es creado y conectado al DOM primero, y luego `<inner>`.

Esto lleva a consecuencias importantes para los elementos personalizados.

Por ejemplo, si un elemento personalizado trata de acceder a `innerHTML` en `connectedCallback`, no obtiene nada:

```

<script>
customElements.define('user-info', class extends HTMLElement {

 connectedCallback() {
 alert(this.innerHTML); // vacío (*)
 }
});
</script>

<user-info>John</user-info>

```

Si lo ejecutas, el `alert` estará vacío.

Esto es porque no hay hijos en aquel estadio, pues el DOM no está finalizado. Se conectó el elemento personalizado `<user-info>` y está por proceder con sus hijos, pero no lo hizo aún.

Si queremos pasar información al elemento personalizado, podemos usar atributos. Estos están disponibles inmediatamente.

O, si realmente necesitamos acceder a los hijos, podemos demorar el acceso a ellos con un `setTimeout` de tiempo cero.

Esto funciona:

```
<script>
customElements.define('user-info', class extends HTMLElement {

 connectedCallback() {
 setTimeout(() => alert(this.innerHTML)); // John (*)
 }
});

</script>

<user-info>John</user-info>
```

Ahora el `alert` en la línea (\*) muestra "John" porque lo corremos asincrónicamente, después de que el armado HTML está completo. Podemos procesar los hijos si lo necesitamos y finalizar la inicialización.

Por otro lado, la solución tampoco es perfecta. Si los elementos anidados también usan `setTimeout` para inicializarse, entonces van a la cola: el `setTimeout` externo se dispara primero y luego el interno.

Como consecuencia, el elemento externo termina la inicialización antes que el interno.

Demostrémoslo con un ejemplo:

```
<script>
customElements.define('user-info', class extends HTMLElement {
 connectedCallback() {
 alert(`#${this.id} connected.`);
 setTimeout(() => alert(`#${this.id} initialized.`));
 }
});
</script>

<user-info id="outer">
 <user-info id="inner"></user-info>
</user-info>
```

Orden de salida:

1. outer conectado.
2. inner conectado.
3. outer inicializado.
4. inner inicializado.

Claramente vemos que el elemento finaliza su inicialización (3) antes que el interno (4).

No existe un callback nativo que se dispare después de que los elementos anidados estén listos. Si es necesario, podemos implementarlo nosotros mismos. Por ejemplo, los elementos internos pueden disparar eventos como `initialized`, y los externos pueden escucharlos para reaccionar a ellos.

## Elementos nativos personalizados

Los elementos nuevos que creamos, tales como `<time-formatted>`, no tienen ninguna semántica asociada. Para los motores de búsqueda son desconocidos, y los dispositivos de accesibilidad tampoco pueden manejarlos.

Pero estas cosas son importantes. Por ejemplo, un motor de búsqueda podría estar interesado en saber que realmente mostramos la hora. y si hacemos una clase especial de botón, ¿por qué no reusar la funcionalidad ya existente de `<button>` ?

Podemos extender y personalizar elementos HTML nativos, heredando desde sus clases.

Por ejemplo, los botones son instancias de `HTMLButtonElement`, construyamos sobre ello.

1. Extender `HTMLButtonElement` con nuestra clase:

```
class HelloButton extends HTMLButtonElement { /* métodos de elemento personalizado */ }
```

2. Ponemos el tercer argumento de `customElements.define`, el cual especifica la etiqueta:

```
customElements.define('hello-button', HelloButton, {extends: 'button'});
```

Puede haber diferentes etiquetas que comparten la misma clase DOM, por eso se necesita especificar `extends`.

3. Por último, para usar nuestro elemento personalizado, insertamos una etiqueta común `<button>`, pero le agregamos `is="hello-button"`:

```
<button is="hello-button">...</button>
```

El ejemplo completo:

```
<script>
// El botón que dice "hello" al hacer clic
class HelloButton extends HTMLButtonElement {
 constructor() {
 super();
 this.addEventListener('click', () => alert("Hello!"));
 }
}

customElements.define('hello-button', HelloButton, {extends: 'button'});
</script>

<button is="hello-button">Click me</button>

<button is="hello-button" disabled>Disabled</button>
```



Nuestro nuevo botón extiende el 'button' nativo. Así mantenemos los mismos estilos y características estándar, como por ejemplo el atributo `disabled`.

## Referencias

- HTML estándar vivo: <https://html.spec.whatwg.org/#custom-elements> ↗.
- Compatibilidad: <https://caniuse.com/#feat=custom-elementsv1> ↗.

## Resumen

Los elementos personalizados pueden ser de dos tipos:

1. “Autónomos” – son etiquetas nuevas, se crean extendiendo `HTMLElement`.

Esquema de definición:

```
class MyElement extends HTMLElement {
 constructor() { super(); /* ... */ }
 connectedCallback() { /* ... */ }
 disconnectedCallback() { /* ... */ }
 static get observedAttributes() { return /* ... */; }
 attributeChangedCallback(name, oldValue, newValue) { /* ... */ }
 adoptedCallback() { /* ... */ }
}
customElements.define('my-element', MyElement);
/* <my-element> */
```

2. “Elementos nativos personalizados” – se crean extendiendo elementos ya existentes.

Requiere un argumento más `.define`, y `is="..."` en HTML:

```
class MyButton extends HTMLElement { /*...*/ }
customElements.define('my-button', MyElement, {extends: 'button'});
/* <button is="my-button"> */
```

Los elementos personalizados tienen muy buen soporte entre los navegadores. Existe un polyfill <https://github.com/webcomponents/polyfills/tree/master/packages/webcomponentsjs>.

## ✓ Tareas

### Elemento reloj dinámico

Ya tenemos un elemento `<time-formatted>` para mostrar la hora agradablemente formateada.

Crea el elemento `<live-timer>` para mostrar la hora actual:

1. Internamente debe usar `<time-formatted>`, no duplicar su funcionalidad.
2. Actualiza (`tic!`) cada segundo.
3. Por cada tic, se debe generar un evento personalizado llamado `tick` con la fecha actual en `event.detail` (ver artículo [Envío de eventos personalizados](#)).

Uso:

```
<live-timer id="elem"></live-timer>

<script>
 elem.addEventListener('tick', event => console.log(event.detail));
</script>
```

Demo:

5:12:43 AM

[Abrir un entorno controlado para la tarea.](#)

[A solución](#)

## Shadow DOM

Shadow DOM sirve para el encapsulamiento. Le permite a un componente tener su propio árbol DOM oculto, que no puede ser accedido por accidente desde el documento principal, puede tener reglas de estilo locales, y más.

### Shadow DOM incorporado

¿Alguna vez pensó cómo los controles complejos del navegador se crean y se les aplica estilo?

Tales como `<input type="range">`:



El navegador usa DOM/CSS internamente para dibujarlos. Esa estructura DOM normalmente está oculta para nosotros, pero podemos verla con herramientas de desarrollo. Por ejemplo, en Chrome, necesitamos habilitar la opción "Show user agent shadow DOM" en las herramientas de desarrollo.

Entonces `<input type="range">` se ve algo así:

```

▼<input type="range"> == $0
 ▼#shadow-root (user-agent)
 ▼<div>
 ▼<div pseudo="-webkit-slider-runnable-track" id="track">
 <div id="thumb"></div>
 </div>
 </div>
</input>

```

Lo que ves bajo `#shadow-root` se llama “shadow DOM”.

No podemos obtener los elementos de shadow DOM incorporados con llamadas normales a JavaScript o selectores. Estos no son hijos normales sino una poderosa técnica de encapsulamiento.

En el ejemplo de abajo podemos ver un útil atributo `pseudo`. No es estándar, existe por razones históricas. Podemos usarlo para aplicar estilo a subelementos con CSS como aquí:

```

<style>
/* hace el control deslizable rojo */
input::-webkit-slider-runnable-track {
 background: red;
}
</style>

<input type="range">

```



De nuevo: `pseudo` no es un atributo estándar. Cronológicamente, los navegadores primero comenzaron a experimentar con estructuras DOM internas para implementar controles, y luego, con el tiempo, fue estandarizado shadow DOM que nos permite, a nosotros desarrolladores, hacer algo similar.

Seguidamente usaremos el moderno estándar shadow DOM cubierto en la [especificación DOM ↗](#).

## Shadow tree (árbol oculto)

Un elemento DOM puede tener dos tipos de subárboles DOM:

1. Light tree – un subárbol normal, hecho de hijos HTML. Todos los subárboles vistos en capítulos previos eran “light”.
2. Shadow tree – un subárbol shadow DOM, no reflejado en HTML, oculto a la vista.

Si un elemento tiene ambos, el navegador solamente construye el árbol shadow. Pero también podemos establecer un tipo de composición entre árboles shadow y light. Veremos los detalles en el capítulo [Shadow DOM slots, composición](#).

El árbol shadow puede ser usado en elementos personalizados para ocultar los componentes internos y aplicarles estilos locales.

Por ejemplo, este elemento `<show-hello>` oculta su DOM interno en un shadow tree:

```

<script>
customElements.define('show-hello', class extends HTMLElement {
 connectedCallback() {
 const shadow = this.attachShadow({mode: 'open'});
 shadow.innerHTML = `<p>
 Hello, ${this.getAttribute('name')}
 </p>`;
 }
});
</script>

<show-hello name="John"></show-hello>

```

Hello, John

Así es como el DOM resultante se ve en las herramientas de desarrollador de Chrome, todo el contenido está bajo `#shadow-root`:

```
▼<show-hello name="John"> == $0
 ▼#shadow-root (open)
 | <p>Hello, John!</p>
 </show-hello>
```

Primero, el llamado a `elem.attachShadow({mode: ...})` crea un árbol shadow.

Hay dos limitaciones:

1. Podemos crear solamente una raíz shadow por elemento.
2. `elem` debe ser: o bien un elemento personalizado, o uno de: "article", "aside", "blockquote", "body", "div", "footer", "h1... h6", "header", "main" "nav", "p", "section", o "span". Otros elementos, como `<img>`, no pueden contener un árbol shadow.

La opción `mode` establece el nivel de encapsulamiento. Debe tener uno de estos dos valores:

- "open" – Abierto: la raíz shadow está disponible como `elem.shadowRoot`.  
Todo código puede acceder el árbol shadow de `elem`.
- "closed" – Cerrado: `elem.shadowRoot` siempre es `null`.

Solamente podemos acceder al shadow DOM por medio de la referencia devuelta por `attachShadow` (y probablemente oculta dentro de un class). Árboles shadow nativos del navegador, tales como `<input type="range">`, son "closed". No hay forma de accederlos.

La raíz [shadow root](#), devuelta por `attachShadow`, es como un elemento: podemos usar `innerHTML` o métodos DOM tales como `append` para llenarlo.

El elemento con una raíz shadow es llamado "shadow tree host" (anfitrión de árbol shadow), y está disponible como la propiedad `host` de shadow root:

```
// asumimos {mode: "open"}, de otra forma elem.shadowRoot sería null
alert(elem.shadowRoot.host === elem); // true
```

## Encapsulamiento

Shadow DOM está fuertemente delimitado del documento principal "main document":

1. Los elementos Shadow DOM no son visibles para `querySelector` desde el DOM visible (light DOM). En particular, los elementos Shadow DOM pueden tener ids en conflicto con aquellos en el DOM visible. Estos deben ser únicos solamente dentro del árbol shadow.
2. El Shadow DOM tiene stylesheets propios. Las reglas de estilo del exterior DOM no se le aplican.

Por ejemplo:

```
<style>
 /* document style no será aplicado al árbol shadow dentro de #elem (1) */
 p { color: red; }
</style>

<div id="elem"></div>

<script>
 elem.attachShadow({mode: 'open'});
 // el árbol shadow tiene su propio style (2)
 elem.shadowRoot.innerHTML =
 <style> p { font-weight: bold; } </style>
 <p>Hello, John!</p>
 ;
 // <p> solo es visible en consultas "query" dentro del árbol shadow (3)
 alert(document.querySelectorAll('p').length); // 0
 alert(elem.shadowRoot.querySelectorAll('p').length); // 1
</script>
```

1. El estilo del documento no afecta al árbol shadow.
2. ...Pero el estilo interno funciona.
3. Para obtener los elementos en el árbol shadow, debemos buscarlos (query) desde dentro del árbol.

## Referencias

- DOM: <https://dom.spec.whatwg.org/#shadow-trees>
- Compatibilidad: <https://caniuse.com/#feat=shadowdomv1>
- Shadow DOM es mencionado en muchas otras especificaciones, por ejemplo [DOM Parsing](#) especifica que el shadow root tiene `innerHTML`.

## Resumen

El Shadow DOM es una manera de crear un DOM de componentes locales.

1. `shadowRoot = elem.attachShadow({mode: open|closed})` – crea shadow DOM para `elem`. Si `mode="open"`, será accesible con la propiedad `elem.shadowRoot`.
2. Podemos llenar `shadowRoot` usando `innerHTML` u otros métodos DOM.

Los elementos de Shadow DOM:

- Tienen su propio espacio de ids,
- Son invisibles a los selectores JavaScript desde el documento principal tales como `querySelector`,
- Usan style solo desde dentro del árbol shadow, no desde el documento principal.

El Shadow DOM, si existe, es construido por el navegador en lugar del DOM visible llamado “light DOM” (hijo regular). En el capítulo [Shadow DOM slots, composición](#) veremos cómo se componen.

## Elemento template

El elemento incorporado `<template>` sirve como almacenamiento para plantillas de markup de HTML. El navegador ignora su contenido, solo verifica la validez de la sintaxis, pero podemos acceder a él y usarlo en JavaScript para crear otros elementos.

En teoría, podríamos crear cualquier elemento invisible en algún lugar de HTML par fines de almacenamiento de HTML markup. ¿Qué hay de especial en `<template>`?

En primer lugar, su contenido puede ser cualquier HTML válido, incluso si normalmente requiere una etiqueta adjunta adecuada.

Por ejemplo, podemos poner una fila de tabla `<tr>`:

```
<template>
 <tr>
 <td>Contenidos</td>
 </tr>
</template>
```

Normalmente, si intentamos poner `<tr>` dentro, digamos, de un `<div>`, el navegador detecta la estructura DOM como inválida y la “arregla”, y añade un `<table>` alrededor. Eso no es lo que queremos. Sin embargo, `<template>` mantiene exactamente lo que ponemos allí.

También podemos poner estilos y scripts dentro de `<template>`:

```
<template>
 <style>
 p { font-weight: bold; }
 </style>
 <script>
 alert("Hola");
 </script>
</template>
```

El navegador considera al contenido `<template>` “fuera del documento”: Los estilos no son aplicados, los scripts no son ejecutados, `<video autoplay>` no es ejecutado, etc.

El contenido cobra vida (estilos aplicados, scripts, etc) cuando los insertamos dentro del documento.

## Insertando template

El contenido template está disponible en su propiedad `content` como un [DocumentFragment](#): un tipo especial de nodo DOM.

Podemos tratarlo como a cualquier otro nodo DOM, excepto por una propiedad especial: cuando lo insertamos en algún lugar, sus hijos son insertados en su lugar.

Por ejemplo:

```
<template id="tmpl1">
 <script>
 alert("Hola");
 </script>
 <div class="message">¡Hola mundo!</div>
</template>

<script>
 let elem = document.createElement('div');

 // Clona el contenido de la plantilla para reutilizarlo múltiples veces
 elem.append(tmpl.content.cloneNode(true));

 document.body.append(elem);
 // Ahora el script de <template> se ejecuta
</script>
```

Reescribamos un ejemplo de Shadow DOM del capítulo anterior usando `<template>`:

```
<template id="tmpl1">
 <style> p { font-weight: bold; } </style>
 <p id="message"></p>
</template>

<div id="elem">Haz clic sobre mí</div>

<script>
 elem.onclick = function() {
 elem.attachShadow({mode: 'open'});

 elem.shadowRoot.append(tmpl.content.cloneNode(true)); // (*)

 elem.shadowRoot.getElementById('message').innerHTML = "¡Saludos desde las sombras!";
 };
</script>
```

Haz clic sobre mí

En la línea `(*)`, cuando clonamos e insertamos `tmpl.content` como su [DocumentFragment](#), sus hijos (`<style>`, `<p>`) se insertan en su lugar.

Ellos forman el shadow DOM:

```
<div id="elem">
 #shadow-root
 <style> p { font-weight: bold; } </style>
 <p id="message"></p>
</div>
```

## Resumen

Para resumir:

- El contenido `<template>` puede ser cualquier HTML sintácticamente correcto.
- El contenido `<template>` es considerado “fuera del documento”, para que no afecte a nada.
- Podemos acceder a `template.content` desde JavaScript, y clonarlo para reusarlo en un nuevo componente.

La etiqueta `<template>` es bastante única, ya que:

- El navegador comprueba la sintaxis HTML dentro de él (lo opuesto a usar una plantilla string dentro de un script).
- ...Pero aún permite el uso de cualquier etiqueta HTML de alto nivel, incluso aquellas que no tienen sentido sin un envoltorio adecuado (por ej. `<tr>`).
- El contenido se vuelve interactivo cuando es insertado en el documento: los scripts se ejecutan, `<video autoplay>` se reproduce, etc.

El elemento `<template>` no ofrece ningún mecanismo de iteración, enlazamiento de datos o sustitución de variables, pero podemos implementar los que están por encima.

## Shadow DOM slots, composición

Muchos tipos de componentes; como pestañas, menús, galerías de imágenes, etc., necesitan renderizar contenido.

Al igual que el `<select>` nativo del navegador espera elementos de `<option>`, nuestros `<custom-tabs>` pueden esperar que se pase el contenido real de la pestaña. Y un `<custom-menu>` puede esperar elementos de menú.

El código que hace uso de `<custom-menu>` puede verse así:

```
<custom-menu>
 <title>Menú de dulces</title>
 <item>Paletas</item>
 <item>Tostada de frutas</item>
 <item>Magdalenas</item>
</custom-menu>
```

...Entonces nuestro componente debería renderizar correctamente, como un agradable menú con un título y elementos dados, manejar eventos de menú, etc.

¿Cómo implementarlo?

Podríamos intentar analizar el contenido del elemento y copiar y reorganizar dinámicamente los nodos del DOM. Esto es posible, pero si estamos moviendo elementos al shadow DOM, entonces los estilos CSS del documento no se aplican allí, por lo que se puede perder el estilo visual. También eso requiere algo de programación.

Afortunadamente, no tenemos que hacerlo. Shadow DOM soporta elementos `<slot>`, que se llenan automáticamente con el contenido del light DOM.

## Slots con nombres

Veamos cómo funcionan los slots en un ejemplo simple.

Aquí, el shadow DOM `<user-card>` proporciona dos slots, que se llenan desde el light DOM:

```
<script>
customElements.define('user-card', class extends HTMLElement {
 connectedCallback() {
 this.attachShadow({mode: 'open'});
 this.shadowRoot.innerHTML = `
 <div>Nombre:
 <slot name="username"></slot>
 </div>
 <div>Cumpleaños:
 <slot name="birthday"></slot>
 </div>
 `;
 }
});
</script>

<user-card>
 John Smith
 01.01.2001
</user-card>
```

```
Nombre: John Smith
Cumpleaños: 01.01.2001
```

En el shadow DOM, `<slot name="X">` define un “punto de inserción”, un lugar donde se renderizan los elementos con `slot="X"`.

Luego, el navegador realiza la “composición”: toma elementos del light DOM y los renderiza en los slots correspondientes del shadow DOM. Al final, tenemos exactamente lo que queremos: un componente que se puede llenar con datos.

Aquí está la estructura del DOM después del script, sin tener en cuenta la composición:

```
<user-card>
 #shadow-root
 <div>Nombre:
 <slot name="username"></slot>
 </div>
 <div>Cumpleaños:
 <slot name="birthday"></slot>
 </div>
 John Smith
 01.01.2001
</user-card>
```

Creamos el shadow DOM, así que aquí está, en `#shadow-root`. Ahora el elemento tiene ambos, light DOM y shadow DOM.

Para fines de renderizado, para cada `<slot name="...">` en el shadow DOM, el navegador busca `slot="..."` con el mismo nombre en el light DOM. Estos elementos se renderizan dentro de los slots:

```
<user-card>
 #shadow-root
 <div>Name:
 <slot name="username"></slot>
 </div>
 <div>Birthday:
 <slot name="birthday"></slot>
 </div>
 John Smith
 01.01.2001
</user-card>
```

El resultado se llama “flattened DOM” (DOM aplanado):

```
<user-card>
 #shadow-root
 <div>Nombre:
 <slot name="username">
 <!-- el elemento eslotead se inserta en el slot -->
 John Smith
 </slot>
 </div>
 <div>Cumpleaños:
 <slot name="birthday">
 01.01.2001
 </slot>
 </div>
</user-card>
```

...Pero el flattened DOM existe solo para fines de procesamiento y manejo de eventos. Es una especie de “virtual DOM”. Así se muestran las cosas. Pero los nodos del documento en realidad no se mueven!

Eso se puede comprobar fácilmente si ejecutamos `querySelectorAll`: los nodos todavía están en sus lugares.

```
// light DOM los nodos siguen en el mismo lugar, en `<user-card>`
alert(document.querySelectorAll('user-card span').length); // 2
```

Entonces, el flattened DOM se deriva del shadow DOM insertando slots. El navegador lo renderiza y lo usa para la herencia de estilo, la propagación de eventos (más sobre esto más adelante). Pero JavaScript todavía ve el documento “tal cual”, antes de acoplarlo.

### ⚠ Solo los nodos hijos de alto nivel pueden tener el atributo slot="..."

El atributo `slot = " ... "` solo es válido para los hijos directos del shadow host (en nuestro ejemplo, el elemento `<user-card>`). Para los elementos anidados, se ignora.

Por ejemplo, el segundo `<span>` aquí se ignora (ya que no es un elemento hijo de nivel superior de `<user-card>`):

```
<user-card>
 John Smith
 <div>
 <!-- slot no válido, debe ser hijo directo de user-card -->
 01.01.2001
 </div>
</user-card>
```

Si hay varios elementos en el light DOM con el mismo nombre de slot, se añaden al slot, uno tras otro.

Por ejemplo, este:

```
<user-card>
 John
 Smith
</user-card>
```

Este flattened DOM con dos elementos en `<slot name="username">`:

```
<user-card>
 #shadow-root
 <div>Nombre:
 <slot name="username">
 John
 Smith
 </slot>
 </div>
 <div>Cumpleaños:
 <slot name="birthday"></slot>
 </div>
</user-card>
```

## Slot con contenido alternativo

Si ponemos algo dentro de un `<slot>`, se convierte en el contenido alternativo, “predeterminado”. El navegador lo muestra si no tiene un equivalente en el Light DOM desde donde llenarlo.

Por ejemplo, en esta parte del shadow DOM, se representa Anónimo si no hay `slot="username"` en el light DOM.

```
<div>Name:
 <slot name="username">anónimo</slot>
</div>
```

## Slot predeterminado: el primero sin nombre

El primer `<slot>` en el shadow DOM que no tiene un nombre es un slot “predeterminado”. Obtiene todos los nodos del light DOM que no están ubicados en otro lugar.

Por ejemplo, agreguemos el slot predeterminado a nuestro `<user-card>` que muestra toda la información sin slotear sobre el usuario:

```
<script>
```

```

customElements.define('user-card', class extends HTMLElement {
 connectedCallback() {
 this.attachShadow({mode: 'open'});
 this.shadowRoot.innerHTML = `
 <div>Nombre:
 <slot name="username"></slot>
 </div>
 <div>Cumpleaños:
 <slot name="birthday"></slot>
 </div>
 <fieldset>
 <legend>Otra información</legend>
 <slot></slot>
 </fieldset>
 `;
 }
});

</script>

<user-card>
 <div>Me gusta nadar.</div>
 John Smith
 01.01.2001
 <div>...Y jugar volleyball también!</div>
</user-card>

```

Nombre: John Smith  
Cumpleaños: 01.01.2001

Otra información

Me gusta nadar.  
...Y jugar volleyball también!

Todo el contenido del light DOM sin slotear entra en el conjunto de campos “Otra información”.

Los elementos se agregan a un slot uno tras otro, por lo que ambas piezas de información sin slotear se encuentran juntas en el slot predeterminado.

El flattened DOM se ve así:

```

<user-card>
 #shadow-root
 <div>Nombre:
 <slot name="username">
 John Smith
 </slot>
 </div>
 <div>Cumpleaños:
 <slot name="birthday">
 01.01.2001
 </slot>
 </div>
 <fieldset>
 <legend>Otra información</legend>
 <slot>
 <div>Me gusta nadar.</div>
 <div>...Y jugar volleyball también!</div>
 </slot>
 </fieldset>
 </user-card>

```

## Ejemplo de menú

Ahora volvamos al `<custom-menu>`, mencionado al principio del capítulo.

Podemos usar slots para distribuir elementos.

Aquí está el marcado para `<custom-menu>`:

```

<custom-menu>
 Menú de dulces
 <li slot="item">Paletas

```

```

<li slot="item">Tostada de frutas
<li slot="item">Magdalenas
</custom-menu>

```

La plantilla del shadow DOM con los slots adecuados:

```

<template id="tmpl">
 <style> /* estilos del menu */ </style>
 <div class="menu">
 <slot name="title"></slot>
 <slot name="item"></slot>
 </div>
</template>

```

1. `<span slot="title">` entra en `<slot name="title">`.
2. Hay muchos `<li slot="item">` en el `<custom-menu>`, pero solo un `<slot name="item">` en la plantilla. Así que todos esos `<li slot="item">` se añaden a `<slot name="item">` uno tras otro, formando así la lista.

El flattened DOM se convierte en:

```

<custom-menu>
 #shadow-root
 <style> /* estilos del menu */ </style>
 <div class="menu">
 <slot name="title">
 Menú de dulces
 </slot>

 <slot name="item">
 <li slot="item">Paletas
 <li slot="item">Tostada de frutas
 <li slot="item">Magdalenas
 </slot>

 </div>
</custom-menu>

```

Uno podría notar que, en un DOM válido, `<li>` debe ser un hijo directo de `<ul>`. Pero esto es flattened DOM, describe cómo se representa el componente, tal cosa sucede naturalmente aquí.

Solo necesitamos agregar un manejador de `click` para abrir/cerrar la lista, y el `<custom-menu>` está listo:

```

customElements.define('custom-menu', class extends HTMLElement {
 connectedCallback() {
 this.attachShadow({mode: 'open'});

 // tmpl es la plantilla del shadow DOM (arriba)
 this.shadowRoot.append(tmpl.content.cloneNode(true));

 // no podemos seleccionar nodos del light DOM, así que manejemos los clics en el slot
 this.shadowRoot.querySelector('slot[name="title"]').onclick = () => {
 // abrir/cerrar el menú
 this.shadowRoot.querySelector('.menu').classList.toggle('closed');
 };
 }
});

```

Aquí está la demostración completa:

```

Candy menu
Lollipop
Fruit Toast
Cup Cake

```

Por supuesto, podemos agregarle más funcionalidad: eventos, métodos, etc.

## Actualizar slots

¿Qué pasa si el código externo quiere agregar/eliminar elementos de menú dinámicamente?

**El navegador monitorea los slots y actualiza la representación si se agregan/eliminan elementos sloteados.**

Además, como los nodos del light DOM no se copian, sino que simplemente se renderizan en los slots, los cambios dentro de ellos se hacen visibles de inmediato.

Así que no tenemos que hacer nada para actualizar el renderizado. Pero si el código del componente quiere saber acerca de los cambios del slot, entonces el evento `slotchange` está disponible.

Por ejemplo, aquí el elemento del menú se inserta dinámicamente después de 1 segundo y el título cambia después de 2 segundos.:

```
<custom-menu id="menu">
 Menú de dulces
</custom-menu>

<script>
customElements.define('custom-menu', class extends HTMLElement {
 connectedCallback() {
 this.attachShadow({mode: 'open'});
 this.shadowRoot.innerHTML = `<div class="menu">
 <slot name="title"></slot>
 <slot name="item"></slot>
 </div>`;

 // shadowRoot no puede tener controladores de eventos, por lo que se usa el primer hijo
 this.shadowRoot.firstChild.addEventListener('slotchange',
 e => alert("slotchange: " + e.target.name)
);
 }
});

setTimeout(() => {
 menu.insertAdjacentHTML('beforeEnd', '<li slot="item">Paletas')
}, 1000);

setTimeout(() => {
 menu.querySelector('[slot="title"]').innerHTML = "Nuevo menú";
}, 2000);
</script>
```

La representación del menú se actualiza cada vez sin nuestra intervención...

Hay dos eventos `slotchange` aquí:

1. En la inicialización:

`slotchange: title` se dispara inmediatamente, cuando el `slot="title"` desde el light DOM entra en el slot correspondiente.

2. Despues de 1 segundo:

`slotchange: item` se activa, cuando se agrega un nuevo `<li slot="item">`.

Observa que no hay ningún evento `slotchange` después de 2 segundos, cuando se modifica el contenido de `slot = "title"`. Eso es porque no hay cambio en el slot. Modificamos el contenido dentro del elemento eslotead, eso es otra cosa.

Si quisieramos rastrear las modificaciones internas del Light DOM desde JavaScript, eso también es posible usando un mecanismo más genérico: [MutationObserver](#).

## Slot API

Finalmente, mencionemos los métodos JavaScript relacionados con los slots.

Como hemos visto antes, JavaScript busca en el DOM “real”, sin aplanar. Pero, si el shadow tree tiene `{mode: 'open'}`, podemos averiguar qué elementos hay asignados a un slot y, viceversa, averiguar el slot por el elemento dentro de él:

- `node.assignedSlot` – retorna el elemento `<slot>` al que está asignado el `nodo`.

- `slot.assignedNodes({flatten: true/false})` – Nodos DOM, asignados al slot. La opción `flatten` es `false` por defecto. Si se establece explícitamente a `true`, entonces mira más profundamente en el flattened DOM, retornando slots anidadas en caso de componentes anidados y el contenido de respaldo si ningún node está asignado.
- `slot.assignedElements({flatten: true/false})` – Elementos DOM, asignados al slot (igual que arriba, pero solo nodos de elementos).

Estos métodos son útiles cuando no solo necesitamos mostrar el contenido esloteadoo, sino también rastrearlo en JavaScript.

Por ejemplo, si el componente `<custom-menu>` quiere saber qué muestra, entonces podría rastrear `slotchange` y obtener los elementos de `slot.assignedElements`:

```

<custom-menu id="menu">
 Menú de dulces
 <li slot="item">Paletas
 <li slot="item">Tostada de frutas
</custom-menu>

<script>
customElements.define('custom-menu', class extends HTMLElement {
 items = []

 connectedCallback() {
 this.attachShadow({mode: 'open'});
 this.shadowRoot.innerHTML = `<div class="menu">
 <slot name="title"></slot>
 <slot name="item"></slot>
 </div>`;

 // se activa cuando cambia el contenido del slot
 this.shadowRoot.firstChild.addEventListener('slotchange', e => {
 let slot = e.target;
 if (slot.name == 'item') {
 this.items = slot.assignedElements().map(elem => elem.textContent);
 alert("Items: " + this.items);
 }
 });
 }
});

// se actualizan después de 1 segundo
setTimeout(() => {
 menu.insertAdjacentHTML('beforeEnd', '<li slot="item">Magdalenas')
}, 1000);
</script>

```

## Resumen

Por lo general, si un elemento tiene shadow DOM, no se muestra su light DOM. Los slots permiten mostrar elementos del light DOM en lugares específicos del shadow DOM.

Hay dos tipos de slots:

- Named slots: `<slot name="X">...</slot>` – consigue los light children con `slot="X"`.
- Default slot: el primer `<slot>` sin un nombre (los slots siguientes sin nombre se ignoran) – obtiene light children sin slotear.
- Si hay muchos elementos para el mismo slot, se añaden uno tras otro.
- El contenido del elemento `<slot>` se utiliza como respaldo. Se muestra si no hay light children para el slot.

El proceso de renderizar elementos sloteados dentro de sus slots se llama “composición”. El resultado se denomina “flattened DOM”.

La composición no mueve realmente los nodos, desde el punto de vista de JavaScript, el DOM sigue siendo el mismo.

JavaScript puede acceder a los slots mediante estos métodos:

- `slot.assignedNodes/Elements()` – retorna nodos/elementos dentro del slot .
- `node.assignedSlot` – la propiedad inversa, retorna el slot por un nodo.

Si queremos saber, podemos rastrear el contenido de los slots usando:

- `slotchange` event – se activa la primera vez que se llena un slot, y en cualquier operación de agregar/quitar/reemplazar del elemento esloeteado, pero no sus hijos. El slot es `event.target`.
- [MutationObserver](#) para profundizar en el contenido del slot, observar los cambios en su interior.

Ahora que, como sabemos cómo mostrar elementos del light DOM en el shadow DOM, veamos cómo diseñarlos correctamente. La regla básica es que los elementos shadow se diseñan en el interior y los elementos light se diseñan afuera, pero hay notables excepciones.

Veremos los detalles en el próximo capítulo.

## Estilo Shadow DOM

Shadow DOM puede incluir las etiquetas `<style>` y `<link rel="stylesheet" href="...">`. En este último caso, las hojas de estilo se almacenan en la caché HTTP, por lo que no se vuelven a descargar para varios de los componentes que usan la misma plantilla.

Como regla general, los estilos locales solo funcionan dentro del shadow tree, y los estilos de documentos funcionan fuera de él. Pero hay pocas excepciones.

### `:host`

El selector `:host` permite seleccionar el shadow host (el elemento que contiene el shadow tree).

Por ejemplo, estamos creando un elemento `<custom-dialog>` que debería estar centrado. Para eso necesitamos diseñar el elemento `<custom-dialog>`.

Eso es exactamente lo que `:host` hace:

```
<template id="tmp1">
<style>
 /* el estilo se aplicará desde el interior al elemento de diálogo personalizado */
 :host {
 position: fixed;
 left: 50%;
 top: 50%;
 transform: translate(-50%, -50%);
 display: inline-block;
 border: 1px solid red;
 padding: 10px;
 }
</style>
<slot></slot>
</template>

<script>
customElements.define('custom-dialog', class extends HTMLElement {
 connectedCallback() {
 this.attachShadow({mode: 'open'}).append(tmp1.content.cloneNode(true));
 }
});
</script>

<custom-dialog>
 Hello!
</custom-dialog>
```



The screenshot shows a browser window with a single element: a centered dialog box containing the text "Hello!". The dialog box has a thin red border and is positioned in the center of the page. The background is white, and there are no other elements visible.

Hello!

## Cascada

El shadow host (`<custom-dialog>` en sí) reside en el light DOM, por lo que se ve afectado por las reglas de CSS del documento.

Si hay una propiedad con estilo tanto en el `:host` localmente, y en el documento, entonces el estilo del documento tiene prioridad.

Por ejemplo, si en el documento tenemos:

```
<style>
custom-dialog {
 padding: 0;
}
</style>
```

...Entonces el `<custom-dialog>` estaría sin padding.

Es muy conveniente, ya que podemos configurar estilos de componentes “predeterminados” en su regla `:host`, y luego sobreescribirlos fácilmente en el documento.

La excepción es cuando una propiedad local está etiquetada como `!important`. Para tales propiedades, los estilos locales tienen prioridad.

## `:host(selector)`

Igual que `:host`, pero se aplica solo si el shadow host coincide con el `selector`.

Por ejemplo, nos gustaría centrar el `<custom-dialog>` solo si tiene el atributo `centered`:

```
<template id="tmpl1">
 <style>
 :host([centered]) {
 position: fixed;
 left: 50%;
 top: 50%;
 transform: translate(-50%, -50%);
 border-color: blue;
 }

 :host {
 display: inline-block;
 border: 1px solid red;
 padding: 10px;
 }
 </style>
 <slot></slot>
</template>

<script>
customElements.define('custom-dialog', class extends HTMLElement {
 connectedCallback() {
 this.attachShadow({mode: 'open'}).append(tmpl1.content.cloneNode(true));
 }
});
</script>

<custom-dialog centered>
 ¡Centrado!
</custom-dialog>

<custom-dialog>
 No centrado.
</custom-dialog>
```

No centrado.

¡Centrado!

Ahora los estilos de centrado adicionales solo se aplican al primer diálogo: `<custom-dialog centered>`.

Para resumir, podemos usar la familia de selectores `:host` para aplicar estilos al elemento principal del componente. Estos estilos (a menos que sea `!important`) pueden ser sobreescritos por el documento.

## **Estilo de contenido eslotado(cuando un elemento ha sido insertado en un slot, se dice que fue eslotado por su término en inglés slotted)**

Ahora consideremos la situación con los slots.

Los elementos eslotados vienen del light DOM, por lo que usan estilos del documento. Los estilos locales no afectan al contenido de los elementos eslotados.

En el siguiente ejemplo, el elemento eslotado `<span>` está en bold, según el estilo del documento, pero no toma el `background` del estilo local:

```
<style>
 span { font-weight: bold }
</style>

<user-card>
 <div slot="username">John Smith</div>
</user-card>

<script>
customElements.define('user-card', class extends HTMLElement {
 connectedCallback() {
 this.attachShadow({mode: 'open'});
 this.shadowRoot.innerHTML = `
 <style>
 span { background: red; }
 </style>
 Name: <slot name="username"></slot>
 `;
 }
});
</script>
```

Name:  
**John Smith**

El resultado es bold, pero no red.

Si queremos aplicar estilos a elementos eslotados en nuestro componente, hay dos opciones.

Primero, podemos aplicarle el estilo al elemento `<slot>` en sí mismo y confiar en la herencia CSS:

```
<user-card>
 <div slot="username">John Smith</div>
</user-card>

<script>
customElements.define('user-card', class extends HTMLElement {
 connectedCallback() {
 this.attachShadow({mode: 'open'});
 this.shadowRoot.innerHTML = `
 <style>
 slot[name="username"] { font-weight: bold; }
 </style>
 Name: <slot name="username"></slot>
 `;
 }
});
</script>
```

Name:  
**John Smith**

Aquí `<p>John Smith</p>` se vuelve bold, porque la herencia CSS está en efecto entre el `<slot>` y su contenido. Pero en el propio CSS no todas las propiedades se heredan.

Otra opción es usar la pseudoclase `::slotted(selector)`. Coincide con elementos en función de 2 condiciones.

1. Eso es un elemento eslotado, que viene del light DOM. El nombre del slot no importa. Cualquier elemento eslotado, pero solo el elemento en si, no sus hijos.
2. El elemento coincide con el selector .

En nuestro ejemplo, `::slotted(div)` selecciona exactamente `<div slot="username">`, pero no sus hijos:

```

<user-card>
 <div slot="username">
 <div>John Smith</div>
 </div>
</user-card>

<script>
customElements.define('user-card', class extends HTMLElement {
 connectedCallback() {
 this.attachShadow({mode: 'open'});
 this.shadowRoot.innerHTML = `
 <style>
 ::slotted(div) { border: 1px solid red; }
 </style>
 Name: <slot name="username"></slot>
 `;
 }
});
</script>

```

Name:  
John Smith

Tenga en cuenta, que el selector `::slotted` no puede descender más en el slot. Estos selectores no son válidos:

```

::slotted(div span) {
 /* nuestro slotted <div> no coincide con esto */
}

::slotted(div) p {
 /* No puede entrar en light DOM */
}

```

También, `::slotted` solo se puede utilizar en CSS. No podemos usarlo en `querySelector`.

## CSS hooks con propiedades personalizadas

¿Cómo diseñamos los elementos internos de un componente del documento principal?

Selectores como `:host` aplican reglas al elemento `<custom-dialog>` o `<user-card>`, ¿pero cómo aplicar estilos a elementos del shadow DOM dentro de ellos?

No hay ningún selector que pueda afectar directamente a los estilos del shadow DOM del documento. Pero así como exponemos métodos para interactuar con nuestro componente, podemos exponer variables CSS (propiedades CSS personalizadas) para darle estilo.

**Existen propiedades CSS personalizadas en todos los niveles, tanto en light como shadow.**

Por ejemplo, en el shadow DOM podemos usar la variable CSS `--user-card-field-color` para dar estilo a los campos, y en el documento exterior establecer su valor:

```

<style>
 .field {
 color: var(--user-card-field-color, black);
 /* si --user-card-field-color no está definido, usar color negro */
 }
</style>
<div class="field">Name: <slot name="username"></slot></div>
<div class="field">Birthday: <slot name="birthday"></slot></div>

```

Entonces, podemos declarar esta propiedad en el documento exterior para `<user-card>`:

```

user-card {
 --user-card-field-color: green;
}

```

Las propiedades personalizadas CSS atraviesan el shadow DOM, son visibles en todas partes, por lo que la regla interna `.field` hará uso de ella.

Aquí está el ejemplo completo:

```
<style>
 user-card {
 --user-card-field-color: green;
 }
</style>

<template id="tmp1">
 <style>
 .field {
 color: var(--user-card-field-color, black);
 }
 </style>
 <div class="field">Name: <slot name="username"></slot></div>
 <div class="field">Birthday: <slot name="birthday"></slot></div>
</template>

<script>
customElements.define('user-card', class extends HTMLElement {
 connectedCallback() {
 this.attachShadow({mode: 'open'});
 this.shadowRoot.append(document.getElementById('tmp1').content.cloneNode(true));
 }
});
</script>

<user-card>
 John Smith
 01.01.2001
</user-card>
```

Name: John Smith  
Birthday: 01.01.2001

## Resumen

Shadow DOM puede incluir estilos, como `<style>` o `<link rel="stylesheet">`.

Los estilos locales pueden afectar:

- shadow tree,
- shadow host con pseudoclases `:host` y `:host()`,
- elementos eslotados (provenientes de light DOM), `::slotted(selector)` permite seleccionar elementos eslotados, pero no a sus hijos.

Los estilos de documentos pueden afectar:

- shadow host (ya que vive en el documento exterior)
- elementos eslotados y su contenido (ya que eso también está en el documento exterior)

Cuando las propiedades CSS entran en conflicto, normalmente los estilos del documento tienen prioridad, a menos que la propiedad esté etiquetada como `!important`. Entonces, los estilos locales tienen prioridad.

Las propiedades CSS personalizadas atraviesan el shadow DOM. Se utilizan como “hooks” para aplicar estilos al componente:

1. El componente utiliza una propiedad CSS personalizada para aplicar estilos a elementos clave, como `var(--component-name-title, <default value>)`.
2. El autor del componente publica estas propiedades para los desarrolladores, son tan importantes como otros métodos de componentes públicos.
3. Cuando un desarrollador desea aplicar un estilo a un título, asigna la propiedad CSS `--component-name-title` para el shadow host o superior.
4. ¡Beneficio!

## Shadow DOM y eventos

La idea detrás del shadow tree es encapsular los detalles internos de implementación de un componente.

Digamos que ocurre un evento click dentro de un shadow DOM del componente `<user-card>`. Pero los scripts en el documento principal no tienen idea acerca del interior del shadow DOM, especialmente si el componente es de una librería de terceros.

Entonces, para mantener los detalles encapsulados, el navegador *redirige* el evento.

**Los eventos que ocurren en el shadow DOM tienen el elemento host como objetivo cuando son atrapados fuera del componente.**

Un ejemplo simple:

```
<user-card></user-card>

<script>
customElements.define('user-card', class extends HTMLElement {
 connectedCallback() {
 this.attachShadow({mode: 'open'});
 this.shadowRoot.innerHTML = `<p>
 <button>Click me</button>
 </p>`;
 this.shadowRoot.firstChild.onclick =
 e => alert("Inner target: " + e.target.tagName);
 }
});

document.onclick =
 e => alert("Outer target: " + e.target.tagName);
</script>
```

Si haces clic en el botón, los mensajes son:

1. Inner target: `BUTTON` – el manejador de evento interno obtiene el objetivo correcto, el elemento dentro del shadow DOM.
2. Outer target: `USER-CARD` – el manejador de evento del documento obtiene el shadow host como objetivo.

Tener la “redirección de eventos” es muy bueno, porque el documento externo no necesita tener conocimiento acerca del interior del componente. Desde su punto de vista, el evento ocurrió sobre `<user-card>`.

**No hay redirección si el evento ocurre en un elemento eslotado (slot element), que físicamente se aloja en el “light DOM”, el DOM visible.**

Por ejemplo, si un usuario hace clic en `<span slot="username">` en el ejemplo siguiente, el objetivo del evento es precisamente ese elemento `span` para ambos manejadores, shadow y light.

```
<user-card id="userCard">
 John Smith
</user-card>

<script>
customElements.define('user-card', class extends HTMLElement {
 connectedCallback() {
 this.attachShadow({mode: 'open'});
 this.shadowRoot.innerHTML = `<div>
 Name: <slot name="username"></slot>
 </div>`;

 this.shadowRoot.firstChild.onclick =
 e => alert("Inner target: " + e.target.tagName);
 }
});

userCard.onclick = e => alert(`Outer target: ${e.target.tagName}`);
</script>
```

**Name:** John Smith

Si un clic ocurre en "John Smith", el target es `<span slot="username">` para ambos manejadores: el interno y el externo. Es un elemento del light DOM, entonces no hay redirección.

Por otro lado, si el clic ocurre en un elemento originalmente del shadow DOM, ej. en `<b>Name</b>`, entonces, como se propaga hacia fuera del shadow DOM, su `event.target` se reestablece a `<user-card>`.

## Propagación, `event.composedPath()`

Para el propósito de propagación de eventos, es usado un "flattened DOM" (DOM aplanado, fusión de light y shadow).

Así, si tenemos un elemento eslotado y un evento ocurre dentro, entonces se propaga hacia arriba a `<slot>` y más allá.

La ruta completa del destino original "event target", con todos sus elementos shadow, puede ser obtenida usando `event.composedPath()`. Como podemos ver del nombre del método, la ruta se toma después de la composición.

En el ejemplo de arriba, el "flattened DOM" es:

```
<user-card id="userCard">
 #shadow-root
 <div>
 Name:
 <slot name="username">
 John Smith
 </slot>
 </div>
</user-card>
```

Entonces, para un clic sobre `<span slot="username">`, una llamada a `event.composedPath()` devuelve un array: `[span, slot, div, shadow-root, user-card, body, html, document, window]`. Que es precisamente la cadena de padres desde el elemento target en el flattened DOM, después de la composición.

**⚠ Los detalles del árbol Shadow solo son provistos en árboles con `{mode: 'open'}`**

Si el árbol shadow fue creado con `{mode: 'closed'}`, la ruta compuesta comienza desde el host: `user-card` en adelante.

Este principio es similar a otros métodos que trabajan con el shadow DOM. El interior de árboles cerrados está completamente oculto.

## `event.composed`

La mayoría de los eventos se propagan exitosamente a través de los límites de un shadow DOM. Hay unos pocos eventos que no.

Esto está gobernado por la propiedad `composed` del objeto de evento. Si es `true`, el evento cruza los límites. Si no, solamente puede ser capturado dentro del shadow DOM.

Vemos en la [especificación UI Events ↗](#) que la mayoría de los eventos tienen `composed: true`:

- `blur`, `focus`, `focusin`, `focusout`,
- `click`, `dblclick`,
- `mousedown`, `mouseup` `mousemove`, `mouseout`, `mouseover`,
- `wheel`,
- `beforeinput`, `input`, `keydown`, `keyup`.

Todos los eventos de toque y puntero también tienen `composed: true`.

Algunos eventos tienen `composed: false`:

- `mouseenter`, `mouseleave` (que no se propagan en absoluto),
- `load`, `unload`, `abort`, `error`,
- `select`,
- `slotchange`.

Estos eventos solo pueden ser capturados dentro del mismo DOM, donde reside el evento target.

## Eventos personalizados

Cuando enviamos eventos personalizados, necesitamos establecer ambas propiedades `bubbles` y `composed` a `true` para que se propague hacia arriba y afuera del componente.

Por ejemplo, aquí creamos `div#inner` en el shadow DOM de `div#outer` y disparamos dos eventos en él. Solo el que tiene `composed: true` logra salir hacia el documento:

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

<script>
outer.attachShadow({mode: 'open'});

let inner = document.createElement('div');
outer.shadowRoot.append(inner);

/*
div(id=outer)
 #shadow-dom
 div(id=inner)
*/
document.addEventListener('test', event => alert(event.detail));

inner.dispatchEvent(new CustomEvent('test', {
 bubbles: true,
 composed: true,
 detail: "composed"
}));

inner.dispatchEvent(new CustomEvent('test', {
 bubbles: true,
 composed: false,
 detail: "not composed"
}));
</script>
```

## Resumen

Los eventos solo cruzan los límites de shadow DOM si su bandera `composed` se establece como `true`.

La mayoría de los eventos nativos tienen `composed: true`, tal como se describe en las especificaciones relevantes:

- Eventos UI <https://www.w3.org/TR/uievents>.
- Eventos Touch <https://w3c.github.io/touch-events>.
- Eventos Pointer <https://www.w3.org/TR/pointerevents>.
- ...y así.

Algunos eventos nativos que tienen `composed: false`:

- `mouseenter`, `mouseleave` (que tampoco se propagan),
- `load`, `unload`, `abort`, `error`,
- `select`,
- `slotchange`.

Estos eventos solo pueden ser capturados en elementos dentro del mismo DOM.

Si enviamos un evento personalizado `CustomEvent`, debemos establecer explícitamente `composed: true`.

Tenga en cuenta que en caso de componentes anidados, un shadow DOM puede estar anidado dentro de otro. En ese caso los eventos se propagan a través de los límites de todos los shadow DOM. Entonces, si se pretende que un evento sea solo para el componente inmediato que lo encierra, podemos enviarlo también en el shadow host y establecer `composed: false`. Entonces saldrá al shadow DOM del componente, pero no se propagará hacia un DOM de mayor nivel.

## Expresiones Regulares

Las expresiones regulares son una forma poderosa de hacer búsqueda y reemplazo de cadenas.

## Patrones y banderas (flags)

Las expresiones regulares son patrones que proporcionan una forma poderosa de buscar y reemplazar texto.

En JavaScript, están disponibles a través del objeto [RegExp](#), además de integrarse en métodos de cadenas.

## Expresiones Regulares

Una expresión regular (también "regexp", o simplemente "reg") consiste en un *patrón* y *banderas* opcionales.

Hay dos sintaxis que se pueden usar para crear un objeto de expresión regular.

La sintaxis "larga":

```
regexp = new RegExp("patrón", "banderas");
```

Y el "corto", usando barras `"/"`:

```
regexp = /pattern/; // sin banderas
regexp = /pattern/gmi; // con banderas g,m e i (para ser cubierto pronto)
```

Las barras `/ . . . /` le dicen a JavaScript que estamos creando una expresión regular. Juegan el mismo papel que las comillas para las cadenas.

En ambos casos, `regexp` se convierte en una instancia de la clase incorporada `RegExp`.

La principal diferencia entre estas dos sintaxis es que el patrón que utiliza barras `/ . . . /` no permite que se inserten expresiones (como los literales de plantilla de cadena con  `${ . . . }` ). Son completamente estáticos.

Las barras se utilizan cuando conocemos la expresión regular en el momento de escribir el código, y esa es la situación más común. Mientras que `new RegExp`, se usa con mayor frecuencia cuando necesitamos crear una expresión regular "sobre la marcha" a partir de una cadena generada dinámicamente. Por ejemplo:

```
let tag = prompt("¿Qué etiqueta quieres encontrar?", "h2");
igual que /<h2>/ si respondió "h2" en el mensaje anterior
```

## Banderas

Las expresiones regulares pueden usar banderas que afectan la búsqueda.

Solo hay 6 de ellas en JavaScript:

**i**

Con esta bandera, la búsqueda no distingue entre mayúsculas y minúsculas: no hay diferencia entre `A` y `a` (consulte el ejemplo a continuación).

**g**

Con esta bandera, la búsqueda encuentra todas las coincidencias, sin ella, solo se devuelve la primera coincidencia.

**m**

Modo multilínea (cubierto en el capítulo [Modo multilínea de anclas ^ \\$, bandera "m"](#)).

**s**

Habilita el modo "dotall", que permite que un punto `.` coincida con el carácter de línea nueva `\n` (cubierto en el capítulo [Clases de caracteres](#)).

**u**

Permite el soporte completo de Unicode. La bandera permite el procesamiento correcto de pares sustitutos. Más del tema en el capítulo [Unicode: bandera "u" y clase \p{...}](#).

## y

Modo “adhesivo”: búsqueda en la posición exacta del texto (cubierto en el capítulo Indicador adhesivo “y”, buscando en una posición.)

### Colores

A partir de aquí, el esquema de color es:

- regexp – red
- cadena (donde buscamos) – blue
- resulta – green

## Buscando: str.match

Como se mencionó anteriormente, las expresiones regulares se integran con los métodos de cadena.

El método `str.match(regex)` busca todas las coincidencias de `regex` en la cadena `str`.

Tiene 3 modos de trabajo:

1. Si la expresión regular tiene la bandera `g`, devuelve un arreglo de todas las coincidencias:

```
let str = "We will, we will rock you";
alert(str.match(/we/gi)); // We,we (un arreglo de 2 subcadenas que coinciden)
```

Tenga en cuenta que tanto `We` como `we` se encuentran, porque la bandera `i` hace que la expresión regular no distinga entre mayúsculas y minúsculas.

2. Si no existe dicha bandera, solo devuelve la primera coincidencia en forma de arreglo, con la coincidencia completa en el índice `0` y algunos detalles adicionales en las propiedades:

```
let str = "We will, we will rock you";
let result = str.match(/we/i); // sin la bandera g

alert(result[0]); // We (1ra coincidencia)
alert(result.length); // 1

// Detalles:
alert(result.index); // 0 (posición de la coincidencia)
alert(result.input); // We will, we will rock you (cadena fuente)
```

El arreglo puede tener otros índices, además de `0` si una parte de la expresión regular está encerrada entre paréntesis. Cubriremos eso en el capítulo [Grupos de captura](#).

3. Y, finalmente, si no hay coincidencias, se devuelve `null` (no importa si hay una bandera `g` o no).

Este es un matiz muy importante. Si no hay coincidencias, no recibimos un arreglo vacío, sino que recibimos `null`. Olvidar eso puede conducir a errores, por ejemplo:

```
let matches = "JavaScript".match(/HTML/); // = null

if (!matches.length) { // Error: No se puede leer la propiedad 'length' de null
 alert("Error en la línea anterior");
}
```

Si queremos que el resultado sea siempre un arreglo, podemos escribirlo de esta manera:

```
let matches = "JavaScript".match(/HTML/) || [];
if (!matches.length) {
 alert("Sin coincidencias"); // ahora si trabaja
}
```

## Reemplazando: str.replace

El método `str.replace(regexp, replacement)` reemplaza las coincidencias encontradas usando `regexp` en la cadena `str` con `replacement` (todas las coincidencias si está la bandera `g`, de lo contrario, solo la primera).

Por ejemplo:

```
// sin la bandera g
alert("We will, we will".replace(/we/i, "I")); // I will, we will

// con la bandera g
alert("We will, we will".replace(/we/ig, "I")); // I will, I will
```

El segundo argumento es la cadena de `replacement`. Podemos usar combinaciones de caracteres especiales para insertar fragmentos de la coincidencia:

Símbolos	Acción en la cadena de reemplazo
<code>\$&amp;</code>	inserta toda la coincidencia
<code>\$`</code>	inserta una parte de la cadena antes de la coincidencia
<code>\$'</code>	inserta una parte de la cadena después de la coincidencia
<code>\$n</code>	si <code>n</code> es un número de 1-2 dígitos, entonces inserta el contenido de los paréntesis <code>n</code> -ésimo, más del tema en el capítulo <a href="#">Grupos de captura</a>
<code>\$&lt;name&gt;</code>	inserta el contenido de los paréntesis con el <code>nombre</code> dado, más del tema en el capítulo <a href="#">Grupos de captura</a>
<code>\$\$</code>	inserta el carácter <code>\$</code>

Un ejemplo con `$&`:

```
alert("Me gusta HTML".replace(/HTML/, "$& y JavaScript")); // Me gusta HTML y JavaScript
```

## Pruebas: regexp.test

El método `regexp.test(str)` busca al menos una coincidencia, si se encuentra, devuelve `true`, de lo contrario `false`.

```
let str = "Me gusta JavaScript";
let regexp = /GUSTA/i;

alert(regexp.test(str)); // true
```

Más adelante en este capítulo estudiaremos más expresiones regulares, exploraremos más ejemplos y también conoceremos otros métodos.

La información completa sobre métodos se proporciona en el artículo No se encontró el artículo "regexp-method".

## Resumen

- Una expresión regular consiste en un patrón y banderas opcionales: `g`, `i`, `m`, `u`, `s`, `y`.
- Sin banderas y símbolos especiales (que estudiaremos más adelante), la búsqueda por expresión regular es lo mismo que una búsqueda de subcadena.
- El método `str.match(regexp)` busca coincidencias: devuelve todas si hay una bandera `g`, de lo contrario, solo la primera.
- El método `str.replace(regexp, replacement)` reemplaza las coincidencias encontradas usando `regexp` con `replacement`: devuelve todas si hay una bandera `g`, de lo contrario solo la primera.
- El método `regexp.test(str)` devuelve `true` si hay al menos una coincidencia, de lo contrario, devuelve `false`.

## Clases de caracteres

Considera una tarea práctica: tenemos un número de teléfono como "+7(903)-123-45-67", y debemos convertirlo en número puro: 79031234567.

Para hacerlo, podemos encontrar y eliminar cualquier cosa que no sea un número. La clase de caracteres pueden ayudar con eso.

Una *clase de caracteres* es una notación especial que coincide con cualquier símbolo de un determinado conjunto.

Para empezar, explóremos la clase "dígito". Está escrito como \d y corresponde a "cualquier dígito".

Por ejemplo, busquemos el primer dígito en el número de teléfono:

```
let str = "+7(903)-123-45-67";
let regexp = /\d/;
alert(str.match(regexp)); // 7
```

Sin la bandera (flag) g, la expresión regular solo busca la primera coincidencia, es decir, el primer dígito \d.

Agreguemos la bandera g para encontrar todos los dígitos:

```
let str = "+7(903)-123-45-67";
let regexp = /\d/g;
alert(str.match(regexp)); // array de coincidencias: 7,9,0,3,1,2,3,4,5,6,7
// hagamos el número de teléfono de solo dígitos:
alert(str.match(regexp).join('')); // 79031234567
```

Esa fue una clase de caracteres para los dígitos. También hay otras.

Las más usadas son:

### \d ("d" es de dígito)

Un dígito: es un carácter de 0 a 9.

### \s ("s" es un espacio)

Un símbolo de espacio: incluye espacios, tabulaciones \t, líneas nuevas \n y algunos otros caracteres raros, como \v, \f y \r.

### \w ("w" es carácter de palabra)

Un carácter de palabra es: una letra del alfabeto latino o un dígito o un guion bajo \_. Las letras no latinas (como el cirílico o el hindú) no pertenecen al \w.

Por ejemplo, \d\s\w significa un "dígito" seguido de un "carácter de espacio" seguido de un "carácter de palabra", como 1 a .

**Una expresión regular puede contener símbolos regulares y clases de caracteres.**

Por ejemplo, CSS\d coincide con una cadena CSS con un dígito después:

```
let str = "¿Hay CSS4?";
let regexp = /CSS\d/
alert(str.match(regexp)); // CSS4
```

También podemos usar varias clases de caracteres:

```
alert("Me gusta HTML5!".match(/\s\w\w\w\w\d/)); // ' HTML5'
```

La coincidencia (cada clase de carácter de la expresión regular tiene el carácter resultante correspondiente):

## Clases inversas

Para cada clase de caracteres existe una “clase inversa”, denotada con la misma letra, pero en mayúscula.

El “inverso” significa que coincide con todos los demás caracteres, por ejemplo:

\D

Sin dígitos: cualquier carácter excepto `\d`, por ejemplo, una letra.

\S

Sin espacio: cualquier carácter excepto `\s`, por ejemplo, una letra.

\W

Sin carácter de palabra: cualquier cosa menos `\w`, por ejemplo, una letra no latina o un espacio.

Al comienzo del capítulo vimos cómo hacer un número de teléfono solo de números a partir de una cadena como `+7(903)-123-45-67`: encontrar todos los dígitos y unirlos.

```
let str = "+7(903)-123-45-67";
alert(str.match(/\d/g).join('')); // 79031234567
```

Una forma alternativa y más corta es usar el patrón sin dígito `\D` para encontrarlos y eliminarlos de la cadena:

```
let str = "+7(903)-123-45-67";
alert(str.replace(/\D/g, "")); // 79031234567
```

## Un punto es “cualquier carácter”

El patrón punto (`.`) es una clase de caracteres especial que coincide con “cualquier carácter excepto una nueva línea”.

Por ejemplo:

```
alert("Z".match(/\./)); // Z
```

O en medio de una expresión regular:

```
let regexp = /CS.4/;
alert("CSS4".match(regexp)); // CSS4
alert("CS-4".match(regexp)); // CS-4
alert("CS 4".match(regexp)); // CS 4 (el espacio también es un carácter)
```

Tenga en cuenta que un punto significa “cualquier carácter”, pero no la “ausencia de un carácter”. Debe haber un carácter para que coincida:

```
alert("CS4".match(/CS.4/)); // null, no coincide porque no hay caracteres entre S y 4
```

## Punto es igual a la bandera “s” que literalmente retorna cualquier carácter

Por defecto, `punto` no coincide con el carácter de línea nueva `\n`.

Por ejemplo, la expresión regular `A.B` coincide con `A`, y luego `B` con cualquier carácter entre ellos, excepto una línea nueva `\n`:

```
alert("A\nB".match(/A.B/)); // null (sin coincidencia)
```

Hay muchas situaciones en las que nos gustaría que *punto* signifique literalmente “cualquier carácter”, incluida la línea nueva.

Eso es lo que hace la bandera `s`. Si una expresión regular la tiene, entonces `.` coincide literalmente con cualquier carácter:

```
alert("A\nB".match(/A.B/s)); // A\nB (coincide!)
```

#### No soportado en IE

La bandera `s` no está soportada en IE.

Afortunadamente, hay una alternativa, que funciona en todas partes. Podemos usar una expresión regular como `[\s\S]` para que coincida con “cualquier carácter”. (Este patrón será cubierto en el artículo [Conjuntos y rangos \[...\]](#)).

```
alert("A\nB".match(/A[\s\S]B/)); // A\nB (coincide!)
```

El patrón `[\s\S]` literalmente dice: “con carácter de espacio O sin carácter de espacio”. En otras palabras, “cualquier cosa”. Podríamos usar otro par de clases complementarias, como `[\d\D]`, eso no importa. O incluso `[^]`, que significa que coincide con cualquier carácter excepto nada.

También podemos usar este truco si queremos ambos tipos de “puntos” en el mismo patrón: el patrón actual `.` comportándose de la manera regular (“sin incluir una línea nueva”), y la forma de hacer coincidir “cualquier carácter” con el patrón `[\s\S]` o similar.

#### Presta atención a los espacios

Por lo general, prestamos poca atención a los espacios. Para nosotros, las cadenas `1-5` y `1 - 5` son casi idénticas.

Pero si una expresión regular no tiene en cuenta los espacios, puede que no funcione.

Intentemos encontrar dígitos separados por un guión:

```
alert("1 - 5".match(/\d-\d/)); // null, sin coincidencia!
```

Vamos a arreglarlo agregando espacios en la expresión regular `\d - \d`:

```
alert("1 - 5".match(/\d - \d/)); // 1 - 5, funciona ahora
// o podemos usar la clase \s:
alert("1 - 5".match(/\d\s-\s\d/)); // 1 - 5, tambien funciona
```

#### Un espacio es un carácter. Igual de importante que cualquier otro carácter.

No podemos agregar o eliminar espacios de una expresión regular y esperar que funcione igual.

En otras palabras, en una expresión regular todos los caracteres importan, los espacios también.

## Resumen

Existen las siguientes clases de caracteres:

- `\d` – dígitos.
- `\D` – sin dígitos.
- `\s` – símbolos de espacio, tabulaciones, líneas nuevas.
- `\S` – todo menos `\s`.
- `\w` – letras latinas, dígitos, guion bajo `'_'`.
- `\W` – todo menos `\w`.
- `.` – cualquier carácter, si la expresión regular usa la bandera `'s'`, de otra forma cualquiera excepto **línea nueva** `\n`.

...¡Pero eso no es todo!

La codificación Unicode, utilizada por JavaScript para las cadenas, proporciona muchas propiedades para los caracteres, como: a qué idioma pertenece la letra (si es una letra), es un signo de puntuación, etc.

Se pueden hacer búsquedas usando esas propiedades. Y se requiere la bandera `u`, analizada en el siguiente artículo.

## Unicode: bandera "u" y clase \p{...}

JavaScript utiliza [codificación Unicode ↗](#) para las cadenas. La mayoría de los caracteres están codificados con 2 bytes, esto permite representar un máximo de 65536 caracteres.

Ese rango no es lo suficientemente grande como para codificar todos los caracteres posibles, es por eso que algunos caracteres raros se codifican con 4 bytes, por ejemplo como `X` (X matemática) o `☺` (una sonrisa), algunos sinogramas, etc.

Aquí los valores unicode de algunos caracteres:

Carácter	Unicode	conteo de Bytes en unicode
a	0x0061	2
=	0x2248	2
χ	0x1d4b3	4
ຍ	0x1d4b4	4
☺	0x1f604	4

Entonces los caracteres como `a` e `=` ocupan 2 bytes, mientras que los códigos para `χ`, `ຍ` y `☺` son más largos, tienen 4 bytes.

Hace mucho tiempo, cuando se creó el lenguaje JavaScript, la codificación Unicode era más simple: no había caracteres de 4 bytes. Por lo tanto, algunas características del lenguaje aún los manejan incorrectamente.

Por ejemplo, aquí `length` interpreta que hay dos caracteres:

```
alert('☺'.length); // 2
alert('χ'.length); // 2
```

...Pero podemos ver que solo hay uno, ¿verdad? El punto es que `length` maneja 4 bytes como dos caracteres de 2 bytes. Eso es incorrecto, porque debe considerarse como uno solo (el llamado “par sustituto”, puede leer sobre ellos en el artículo [Strings](#)).

Por defecto, las expresiones regulares manejan los “caracteres largos” de 4 bytes como un par de caracteres de 2 bytes cada uno. Y, como sucede con las cadenas, eso puede conducir a resultados extraños. Lo veremos un poco más tarde, en el artículo No se encontró el artículo “regexp-character-sets-and-range”.

A diferencia de las cadenas, las expresiones regulares tienen la bandera `u` que soluciona tales problemas. Con dicha bandera, una expresión regular maneja correctamente los caracteres de 4 bytes. Y podemos usar la búsqueda de propiedades Unicode, que veremos a continuación.

## Propiedades Unicode \p{...}

Cada carácter en Unicode tiene varias propiedades. Describen a qué “categoría” pertenece el carácter, contienen información diversa al respecto.

Por ejemplo, si un carácter tiene la propiedad `Letter`, significa que pertenece a un alfabeto (de cualquier idioma). Y la propiedad `Number` significa que es un dígito: tal vez árabe o chino, y así sucesivamente.

Podemos buscar caracteres por su propiedad, usando `\p{...}`. Para usar `\p{...}`, una expresión regular debe usar también `u`.

Por ejemplo, `\p{Letter}` denota una letra en cualquiera de los idiomas. También podemos usar `\p{L}`, ya que `L` es un alias de `Letter`. Casi todas las propiedades tienen alias cortos.

En el ejemplo a continuación se encontrarán tres tipos de letras: inglés, georgiano y coreano.

```
let str = "A ბ ㄱ";
alert(str.match(/\p{L}/gu)); // A,ბ,ㄱ
alert(str.match(/\p{L}/g)); // null (sin coincidencia, como no hay bandera "u")
```

Estas son las principales categorías y subcategorías de caracteres:

- Letter (Letra) `L`:
  - lowercase (minúscula) `Ll`,
  - modifier (modificador) `Lm`,
  - titlecase (capitales) `Lt`,
  - uppercase (mayúscula) `Lu`,
  - other (otro) `Lo`.
- Number (número) `N`:
  - decimal digit (dígito decimal) `Nd`,
  - letter number (número de letras) `Nl`,
  - other (otro) `No`.
- Punctuation (puntuación) `P`:
  - connector (conector) `Pc`,
  - dash (guión) `Pd`,
  - initial quote (comilla inicial) `Pi`,
  - final quote (comilla final) `Pf`,
  - open (abre) `Ps`,
  - close (cierra) `Pe`,
  - other (otro) `Po`.
- Mark (marca) `M` (acentos etc):
  - spacing combining (combinación de espacios) `Mc`,
  - enclosing (encerrado) `Me`,
  - non-spacing (sin espaciado) `Mn`.
- Symbol (símbolo) `S`:
  - currency (moneda) `Sc`,
  - modifier (modificador) `Sk`,
  - math (matemática) `Sm`,
  - other (otro) `So`.
- Separator (separador) `Z`:
  - line (línea) `Zl`,
  - paragraph (párrafo) `Zp`,
  - space (espacio) `Zs`.
- Other (otros) `C`:
  - control `Cc`,
  - format (formato) `Cf`,
  - not assigned (sin asignación) `Cn`,
  - private use (uso privado) `Co`,
  - surrogate (sustituto) `Cs`.

Entonces, por ejemplo si necesitamos letras en minúsculas, podemos escribir `\p{Ll}`, signos de puntuación: `\p{P}` y así sucesivamente.

También hay otras categorías derivadas, como:

- `Alphabetic` (alfabético) (`Alpha`), incluye letras `L`, más números de letras `Nl` (por ejemplo, `XII` – un carácter para el número romano 12), y otros símbolos `Other_Alphabetic` (`OAlpha`).
- `Hex_Digit` incluye dígitos hexadecimales: `0-9`, `a-f`.
- ...Y así.

Unicode admite muchas propiedades diferentes, la lista completa es muy grande, estas son las referencias:

- Lista de todas las propiedades por carácter: <https://unicode.org/cldr/utility/character.jsp> (enlace no disponible).
- Lista de caracteres por propiedad: <https://unicode.org/cldr/utility/list-unicodeset.jsp> (enlace no disponible)
- Alias cortos para propiedades: <https://www.unicode.org/Public/UCD/latest/ucd/PropertyValueAliases.txt> (enlace no disponible)

- Aquí una base completa de caracteres Unicode en formato de texto, con todas las propiedades: <https://www.unicode.org/Public/UCD/latest/ucd/>.

### Ejemplo: números hexadecimales

Por ejemplo, busquemos números hexadecimales, escritos como `xFF` donde `F` es un dígito hexadecimal (0...9 o A...F).

Un dígito hexadecimal se denota como `\p{Hex_Digit}`:

```
let regexp = /x\p{Hex_Digit}\p{Hex_Digit}/u;
alert("número: xAF".match(regexp)); // xAF
```

### Ejemplo: sinogramas chinos

Busquemos sinogramas chinos.

Hay una propiedad Unicode `Script` (un sistema de escritura), que puede tener un valor: `Cyrillic`, `Greek`, `Arabic`, `Han` (chino), etc. [lista completa](#).

Para buscar caracteres de un sistema de escritura dado, debemos usar `Script=<value>`, por ejemplo para letras cirílicas: `\p{sc=Cyrillic}`, para sinogramas chinos: `\p{sc=Han}`, y así sucesivamente:

```
let regexp = /\p{sc=Han}/gu; // devuelve sinogramas chinos
let str = `Hello Привет 你好 123_456`;
alert(str.match(regexp)); // 你,好
```

### Ejemplo: moneda

Los caracteres que denotan una moneda, como `$`, `€`, `¥`, tienen la propiedad unicode `\p{Currency_Symbol}`, el alias corto: `\p{Sc}`.

Usémoslo para buscar precios en el formato “moneda, seguido de un dígito”:

```
let regexp = /\p{Sc}\d/gu;
let str = `Precios: $2, €1, ¥9`;
alert(str.match(regexp)); // $2,€1,¥9
```

Más adelante, en el artículo [Cuantificadores +, \\*, ? y {n}](#) veremos cómo buscar números que contengan muchos dígitos.

## Resumen

La bandera `u` habilita el soporte de Unicode en expresiones regulares.

Eso significa dos cosas:

1. Los caracteres de 4 bytes se manejan correctamente: como un solo carácter, no dos caracteres de 2 bytes.
2. Las propiedades Unicode se pueden usar en las búsquedas: `\p{...}`.

Con las propiedades Unicode podemos buscar palabras en determinados idiomas, caracteres especiales (comillas, monedas), etc.

## Anclas: inicio ^ y final \$ de cadena

Los patrones caret (del latín carece) `^` y dólar `$` tienen un significado especial en una expresión regular. Se llaman “anclas”.

El patrón caret `^` coincide con el principio del texto y dólar `$` con el final.

Por ejemplo, probemos si el texto comienza con `Mary`:

```
let str1 = "Mary tenía un corderito";
alert(/^Mary/.test(str1)); // true
```

El patrón `^Mary` significa: "inicio de cadena y luego Mary".

Similar a esto, podemos probar si la cadena termina con `nieve` usando `nieve$`:

```
let str1 = "su vellón era blanco como la nieve";
alert(/nieve$/.test(str1)); // true
```

En estos casos particulares, en su lugar podríamos usar métodos de cadena `beginWith/endsWith`. Las expresiones regulares deben usarse para pruebas más complejas.

## Prueba para una coincidencia completa

Ambos anclajes `^...$` se usan juntos a menudo para probar si una cadena coincide completamente con el patrón. Por ejemplo, para verificar si la entrada del usuario está en el formato correcto.

Verifiquemos si una cadena está o no en formato de hora `12:34`. Es decir: dos dígitos, luego dos puntos y luego otros dos dígitos.

En el idioma de las expresiones regulares eso es `\d\d:\d\d`:

```
let goodInput = "12:34";
let badInput = "12:345";

let regexp = /^\\d\\d:\\d\\d$/;
alert(regexp.test(goodInput)); // true
alert(regexp.test(badInput)); // false
```

La coincidencia para `\d\d:\d\d` debe comenzar exactamente después del inicio de texto `^`, y seguido inmediatamente, el final `$`.

Toda la cadena debe estar exactamente en este formato. Si hay alguna desviación o un carácter adicional, el resultado es falso.

Las anclas se comportan de manera diferente si la bandera `m` está presente. Lo veremos en el próximo artículo.

### Las anclas tienen "ancho cero"

Las anclas `^` y `$` son pruebas. Ellas tienen ancho cero.

En otras palabras, no coinciden con un carácter, sino que obligan al motor regexp a verificar la condición (inicio/fin de texto).

## Tareas

### Regexp `^$`

¿Qué cadena coincide con el patrón `^$`?

[A solución](#)

### Modo multilínea de anclas `^ $`, bandera "m"

El modo multilínea está habilitado por el indicador `m`.

Solo afecta el comportamiento de `^` y `$`.

En el modo multilínea, coinciden no solo al principio y al final de la cadena, sino también al inicio/final de la línea.

### Buscando al inicio de línea `^`

En el siguiente ejemplo, el texto tiene varias líneas. El patrón `/^\\d/gm` toma un dígito desde el principio de cada línea:

```
let str = `1er lugar: Winnie
```

```
2do lugar: Piglet
3er lugar: Eeyore`;

console.log(str.match(/\^d/gm)); // 1, 2, 3
```

Sin la bandera `m` solo coincide el primer dígito:

```
let str = `1er lugar: Winnie
2do lugar: Piglet
3er lugar: Eeyore`;

console.log(str.match(/\^d/g)); // 1
```

Esto se debe a que, de forma predeterminada, un caret `\^` solo coincide al inicio del texto y en el modo multilínea, al inicio de cualquier línea.

**i Por favor tome nota:**

“Inicio de una línea” significa formalmente “inmediatamente después de un salto de línea”: la prueba `\^` en modo multilínea coincide en todas las posiciones precedidas por un carácter de línea nueva `\n`.

Y al comienzo del texto.

## Buscando al final de la línea \$

El signo de dólar `$` se comporta de manera similar.

La expresión regular `\d$` encuentra el último dígito en cada línea

```
let str = `Winnie: 1
Piglet: 2
Eeyore: 3`;

console.log(str.match(/\d$/gm)); // 1,2,3
```

Sin la bandera `m`, dólar `$` solo coincidiría con el final del texto completo, por lo que solo se encontraría el último dígito.

**i Por favor tome nota:**

“Fin de una línea” significa formalmente “inmediatamente antes de un salto de línea”: la prueba `$` en el modo multilínea coincide en todas las posiciones seguidas por un carácter de línea nueva `\n`.

Y al final del texto.

## Buscando \n en lugar de ^ \$

Para encontrar una línea nueva, podemos usar no solo las anclas `\^` y `$`, sino también el carácter de línea nueva `\n`.

¿Cuál es la diferencia? Veamos un ejemplo.

Buscamos `\d\n` en lugar de `\d$`:

```
let str = `Winnie: 1
Piglet: 2
Eeyore: 3`;

console.log(str.match(/\d\n/g)); // 1\n,2\n
```

Como podemos ver, hay 2 coincidencias en lugar de 3.

Esto se debe a que no hay una línea nueva después de `3` (sin embargo, hay un final de texto, por lo que coincide con `$`).

Otra diferencia: ahora cada coincidencia incluye un carácter de línea nueva `\n`. A diferencia de las anclas `\^` `$`, que solo prueban la condición (inicio/final de una línea), `\n` es un carácter, por lo que se hace parte del resultado.

Entonces, un `\n` en el patrón se usa cuando necesitamos encontrar caracteres de línea nueva, mientras que las anclas se usan para encontrar algo “al principio/al final” de una línea.

## Límite de palabra: `\b`

Un límite de palabra `\b` es una prueba, al igual que `^` y `$`.

Cuando el motor regex (módulo de programa que implementa la búsqueda de expresiones regulares) se encuentra con `\b`, comprueba que la posición en la cadena es un límite de palabra.

Hay tres posiciones diferentes que califican como límites de palabras:

- Al comienzo de la cadena, si el primer carácter de cadena es un carácter de palabra `\w`.
- Entre dos caracteres en la cadena, donde uno es un carácter de palabra `\w` y el otro no.
- Al final de la cadena, si el último carácter de la cadena es un carácter de palabra `\w`.

Por ejemplo, la expresión regular `\bJava\b` se encontrará en `Hello, Java!`, donde `Java` es una palabra independiente, pero no en `Hello, JavaScript!`.

```
alert("Hello, Java!".match(/\bJava\b/)); // Java
alert("Hello, JavaScript!".match(/\bJava\b/)); // null
```

En la cadena `Hello, Java!` las flechas que se muestran corresponden a `\b`, ver imagen:

Entonces, coincide con el patrón `\bHello\b`, porque:

1. Al comienzo de la cadena coincide con la primera prueba: `\b`.
2. Luego coincide con la palabra `Hello`.
3. Luego, la prueba `\b` vuelve a coincidir, ya que estamos entre `o` y una coma.

El patrón `\bHello\b` también coincidiría. Pero no `\bHel\b` (porque no hay límite de palabras después de `l`) y tampoco `Java!\b` (porque el signo de exclamación no es un carácter común `\w`, entonces no hay límite de palabras después de eso).

```
alert("Hello, Java!".match(/\bHello\b/)); // Hello
alert("Hello, Java!".match(/\bJava\b/)); // Java
alert("Hello, Java!".match(/\bHell\b/)); // null (sin coincidencia)
alert("Hello, Java!".match(/\bJava!\b/)); // null (sin coincidencia)
```

Podemos usar `\b` no solo con palabras, sino también con dígitos.

Por ejemplo, el patrón `\b\d\d\b` busca números independientes de 2 dígitos. En otras palabras, busca números de 2 dígitos que están rodeados por caracteres diferentes de `\w`, como espacios o signos de puntuación (o texto de inicio/final).

```
alert("1 23 456 78".match(/\b\d\d\b/g)); // 23,78
alert("12,34,56".match(/\b\d\d\b/g)); // 12,34,56
```

### ⚠️ El límite de palabra `\b` no funciona para alfabetos no latinos

La prueba de límite de palabra `\b` verifica que debe haber un `\w` en un lado de la posición y “no `\w`”- en el otro lado.

Pero `\w` significa una letra latina `a-z` (o un dígito o un guion bajo), por lo que la prueba no funciona para otros caracteres, p.ej.: letras cirílicas o jeroglíficos.

## ✔️ Tareas

### Encuentra la hora

La hora tiene un formato: `horas:minutos`. Tanto las horas como los minutos tienen dos dígitos, como `09:00`.

Haz una expresión regular para encontrar el tiempo en la cadena: `Desayuno a las 09:00 en la habitación 123:456.`

P.D.: En esta tarea todavía no hay necesidad de verificar la corrección del tiempo, por lo que `25:99` también puede ser un resultado válido.

P.P.D.: La expresión regular no debe coincidir con `123:456`.

## A solución

## Escapando, caracteres especiales

Como hemos visto, una barra invertida `\` se usa para denotar clases de caracteres, p.ej. `\d`. Por lo tanto, es un carácter especial en expresiones regulares (al igual que en las cadenas regulares).

También hay otros caracteres especiales que tienen un significado especial en una expresión regular, tales como `[ ] { }` `( ) \ ^ $ . | ? * +`. Se utilizan para hacer búsquedas más potentes.

No intentes recordar la lista: pronto nos ocuparemos de cada uno de ellos por separado y los recordarás fácilmente.

## Escapando

Digamos que queremos encontrar literalmente un punto. No “cualquier carácter”, sino solo un punto.

Para usar un carácter especial como uno normal, agrégalo con una barra invertida: `\.`.

A esto se le llama “escape de carácter”.

Por ejemplo:

```
alert("Capítulo 5.1".match(/\d\.\d/)); // 5.1 (Coincide!)
alert("Capítulo 511".match(/\d\.\d/)); // null (buscando un punto real \.)
```

Los paréntesis también son caracteres especiales, por lo que si los buscamos, deberíamos usar `\()`. El siguiente ejemplo busca una cadena `"g()"`:

```
alert("función g()".match(/g\(\)/)); // "g()"
```

Si estamos buscando una barra invertida `\`, como es un carácter especial tanto en cadenas regulares como en expresiones regulares, debemos duplicarlo.

```
alert("1\\2".match(/\\\/)); // '\\'
```

## Una barra

Un símbolo de barra `' / '` no es un carácter especial, pero en JavaScript se usa para abrir y cerrar expresiones regulares: `/...pattern.../`, por lo que también debemos escaparlo.

Así es como se ve la búsqueda de una barra `' / '`:

```
alert("/" .match(/\//)); // '/'
```

Por otro lado, si no estamos usando `/.../`, pero creamos una expresión regular usando `new RegExp`, entonces no necesitamos escaparla:

```
alert("/" .match(new RegExp("/"))); // encuentra /
```

## new RegExp

Si estamos creando una expresión regular con `new RegExp`, entonces no tenemos que escapar la barra `/`, pero sí otros caracteres especiales.

Por ejemplo, considere esto:

```
let regexp = new RegExp("\d.\d");
alert("Capítulo 5.1".match(regexp)); // null
```

En uno de los ejemplos anteriores funcionó la búsqueda con `/\d.\d/`, pero `new RegExp ("\\d\\.\\d")` no funciona, ¿por qué?

La razón es que las barras invertidas son “consumidas” por una cadena. Como podemos recordar, las cadenas regulares tienen sus propios caracteres especiales, como `\n`, y se usa una barra invertida para escapar esos caracteres especiales de cadena.

Así es como se percibe “`\d.\d`”:

```
alert("\d.\d"); // d.d
```

Las comillas de cadena “consumen” barras invertidas y las interpretan como propias, por ejemplo:

- `\n` – se convierte en un carácter de línea nueva,
- `\u1234` – se convierte en el carácter Unicode con dicho código,
- ...Y cuando no hay un significado especial: como `\d` o `\z`, entonces la barra invertida simplemente se elimina.

Así que `new RegExp` toma una cadena sin barras invertidas. ¡Por eso la búsqueda no funciona!

Para solucionarlo, debemos duplicar las barras invertidas, porque las comillas de cadena convierten `\\"` en `\`:

```
let regStr = "\\\d\\\\.\\d";
alert(regStr); // \d.\d (ahora está correcto)

let regexp = new RegExp(regStr);

alert("Capítulo 5.1".match(regexp)); // 5.1
```

## Resumen

- Para buscar literalmente caracteres especiales `[ \ ^ $ . | ? * + ( ) ]`, se les antepone una barra invertida `\` (“escaparlos”).
- Se debe escapar `/` si estamos dentro de `/.../` (pero no dentro de `new RegExp`).
- Al pasar una cadena a `new RegExp`, se deben duplicar las barras invertidas `\\"`, porque las comillas de cadena consumen una.

## Conjuntos y rangos [...]

Varios caracteres o clases de caracteres entre corchetes `[ ... ]` significa “buscar cualquier carácter entre los datos”.

## Conjuntos

Por ejemplo, `[eao]` significa cualquiera de los 3 caracteres: `'a'`, `'e'`, o `'o'`.

A esto se le llama *conjunto*. Los conjuntos se pueden usar en una expresión regular junto con los caracteres normales:

```
// encontrar [t ó m], y luego "op"
alert("Mop top".match(/tm]op/gi)); // "Mop", "top"
```

Tenga en cuenta que aunque hay varios caracteres en el conjunto, corresponden exactamente a un carácter en la coincidencia.

Entonces, en el siguiente ejemplo no hay coincidencias:

```
// encuentra "V", luego [o ó i], luego "la"
alert("Voila".match(/V[oi]la/)); // null, sin coincidencias
```

El patrón busca:

- `V`,
- después *una* de las letras `[oi]`,
- después `la`.

Entonces habría una coincidencia para `Vola` o `Vila`.

## Rangos

Los corchetes también pueden contener *rangos de caracteres*.

Por ejemplo, `[a-z]` es un carácter en el rango de `a` a `z`, y `[0-5]` es un dígito de `0` a `5`.

En el ejemplo a continuación, estamos buscando `"x"` seguido de dos dígitos o letras de `A` a `F`:

```
alert("Excepción 0xAF".match(/x[0-9A-F][0-9A-F]/g)); // xAF
```

Aquí `[0-9A-F]` tiene dos rangos: busca un carácter que sea un dígito de `0` a `9` o una letra de `A` a `F`.

Si también queremos buscar letras minúsculas, podemos agregar el rango `a-f`: `[0-9A-Fa-f]`. O se puede agregar la bandera `i`.

También podemos usar clases de caracteres dentro de los `[...]`.

Por ejemplo, si quisiéramos buscar un carácter de palabra `\w` o un guion `-`, entonces el conjunto es `[\w-]`.

También es posible combinar varias clases, p.ej.: `[\s\d]` significa “un carácter de espacio o un dígito”.

**💡 Las clases de caracteres son abreviaturas (o atajos) para ciertos conjuntos de caracteres.**

Por ejemplo:

- `\d` – es lo mismo que `[0-9]`,
- `\w` – es lo mismo que `[a-zA-Z0-9_]`,
- `\s` – es lo mismo que `[\t\n\v\f\r]`, además de otros caracteres de espacio raros de unicode.

## Ejemplo: multi-idioma \w

Como la clase de caracteres `\w` es una abreviatura de `[a-zA-Z0-9_]`, no puede coincidir con sinogramas chinos, letras cirílicas, etc.

Podemos escribir un patrón más universal, que busque caracteres de palabra en cualquier idioma. Eso es fácil con las propiedades unicode: `[\p{Alpha}\p{M}\p{Nd}\p{Pc}\p{Join_C}]`.

Decifrémolo. Similar a `\w`, estamos creando un conjunto propio que incluye caracteres con las siguientes propiedades unicode:

- `Alfabético (Alpha)` – para letras,
- `Marca (M)` – para acentos,
- `Numero_Decimal (Nd)` – para dígitos,
- `Conector_Puntuación (Pc)` – para guion bajo `'_'` y caracteres similares,
- `Control_Unión (Join_C)` – dos códigos especiales `200c` and `200d`, utilizado en ligaduras, p.ej. en árabe.

Un ejemplo de uso:

```
let regexp = /[^\p{Alpha}\p{M}\p{Nd}\p{Pc}\p{Join_C}]/gu;
let str = `Hola 你好 12`;
```

```
// encuentra todas las letras y dígitos:
alert(str.match(regexp)); // H,o,l,a,你,好,1,2
```

Por supuesto, podemos editar este patrón: agregar propiedades unicode o eliminarlas. Las propiedades Unicode se cubren con más detalle en el artículo [Unicode: bandera "u" y clase \p{...}](#).

### ⚠️ Las propiedades Unicode no son soportadas por IE

Las propiedades Unicode `\p{...}` no se implementaron en IE. Si realmente las necesitamos, podemos usar la biblioteca [XRegExp ↗](#).

O simplemente usa rangos de caracteres en el idioma de tu interés, p.ej. `[a-я]` para letras cirílicas.

## Excluyendo rangos

Además de los rangos normales, hay rangos “excluyentes” que se parecen a `[^...]`.

Están denotados por un carácter caret `^` al inicio y coinciden con cualquier carácter *excepto los dados*.

Por ejemplo:

- `[^aeyo]` – cualquier carácter excepto 'a', 'e', 'y' u 'o'.
- `[^0-9]` – cualquier carácter excepto un dígito, igual que `\D`.
- `[^\s]` – cualquiere carácter sin espacio, igual que `\S`.

El siguiente ejemplo busca cualquier carácter, excepto letras, dígitos y espacios:

```
alert("alice15@gmail.com".match(/[^d\sA-Z]/gi)); // @ y .
```

## Escapando dentro de corchetes [...]

Por lo general, cuando queremos encontrar exactamente un carácter especial, necesitamos escaparlo con `\.`. Y si necesitamos una barra invertida, entonces usamos `\\\`, y así sucesivamente.

Entre corchetes podemos usar la gran mayoría de caracteres especiales sin escaparlos:

- Los símbolos `.` + ( ) nunca necesitan escape.
- Un guion `-` no se escapa al principio ni al final (donde no define un rango).
- Un carácter caret `^` solo se escapa al principio (donde significa exclusión).
- El corchete de cierre `]` siempre se escapa (si se necesita buscarlo).

En otras palabras, todos los caracteres especiales están permitidos sin escapar, excepto cuando significan algo entre corchetes.

Un punto `.` dentro de corchetes significa solo un punto. El patrón `[.,]` Buscaría uno de los caracteres: un punto o una coma.

En el siguiente ejemplo, la expresión regular `[-( ).^+]` busca uno de los caracteres - ( ) . ^+ :

```
// no es necesario escaparlos
let regexp = /[-().^+]/g;

alert("1 + 2 - 3".match(regexp)); // Coincide +, -
```

...Pero si decides escaparlos “por si acaso”, no habría daño:

```
// Todo escapado
let regexp = /[\\-\\\\(\\\\)\\\\.\\\\^\\\\+]/g;

alert("1 + 2 - 3".match(regexp)); // funciona también: +, -
```

## Rangos y la bandera (flag) “u”

Si hay pares sustitutos en el conjunto, se requiere la flag `u` para que funcionen correctamente.

Por ejemplo, busquemos `[XY]` en la cadena `X`:

```
alert('X'.match(/XY/)); // muestra un carácter extraño, como [?]
// (la búsqueda se realizó incorrectamente, se devolvió medio carácter)
```

El resultado es incorrecto porque, por defecto, las expresiones regulares “no saben” sobre pares sustitutos.

El motor de expresión regular piensa que la cadena `[XY]` no son dos, sino cuatro caracteres:

1. mitad izquierda de `X` (1),
2. mitad derecha de `X` (2),
3. mitad izquierda de `Y` (3),
4. mitad derecha de `Y` (4).

Sus códigos se pueden mostrar ejecutando:

```
for(let i = 0; i < 'XY'.length; i++) {
 alert('XY'.charCodeAt(i)); // 55349, 56499, 55349, 56500
};
```

Entonces, el ejemplo anterior encuentra y muestra la mitad izquierda de `X`.

Si agregamos la flag `u`, entonces el comportamiento será correcto:

```
alert('X'.match(/XY/u)); // X
```

Ocurre una situación similar cuando se busca un rango, como `[X-Y]`.

Si olvidamos agregar la flag `u`, habrá un error:

```
'X'.match(/[X-Y]/); // Error: Expresión regular inválida
```

La razón es que sin la bandera `u` los pares sustitutos se perciben como dos caracteres, por lo que `[X-Y]` se interpreta como `[<55349><56499>-<55349><56500>]` (cada par sustituto se reemplaza con sus códigos). Ahora es fácil ver que el rango `56499-55349` es inválido: su código de inicio `56499` es mayor que el último `55349`. Esa es la razón formal del error.

Con la bandera `u` el patrón funciona correctamente:

```
// buscar caracteres desde X a Z
alert('Y'.match(/X-Z/u)); // Y
```

## ✓ Tareas

### Java[^script]

Tenemos una regexp `/Java[^script]/`.

¿Coincide con algo en la cadena `Java`? ¿Y en la cadena `JavaScript`?

[A solución](#)

### Encuentra la hora como hh:mm o hh-mm

La hora puede estar en el formato `horas:minutos` u `horas-minutos`. Tanto las horas como los minutos tienen 2 dígitos: `09:00` ó `21-30`.

Escribe una regexp que encuentre la hora:

```
let regexp = /tu regexp/g;
alert("El desayuno es a las 09:00. La cena es a las 21-30".match(regexp)); // 09:00, 21-30
```

En esta tarea asumimos que el tiempo siempre es correcto, no hay necesidad de filtrar cadenas malas como "45:67". Más tarde nos ocuparemos de eso también.

## A solución

## Cuantificadores +, \*, ? y {n}

Digamos que tenemos una cadena como `+7 (903) -123-45-67` y queremos encontrar todos los números en ella. Pero contrastando el ejemplo anterior, no estamos interesados en un solo dígito, sino en números completos: `7, 903, 123, 45, 67`.

Un número es una secuencia de 1 o más dígitos `\d`. Para marcar cuántos necesitamos, podemos agregar un *cuantificador*.

### Cantidad {n}

El cuantificador más simple es un número entre llaves: `{n}`.

Se agrega un cuantificador a un carácter (o a una clase de caracteres, o a un conjunto `[ . . . ]`, etc) y especifica cuántos necesitamos.

Tiene algunas formas avanzadas, veamos los ejemplos:

#### El recuento exacto: `{5}`

`\d{5}` Denota exactamente 5 dígitos, igual que `\d\d\d\d\d`.

El siguiente ejemplo busca un número de 5 dígitos:

```
alert("Tengo 12345 años de edad".match(/\d{5}/)); // "12345"
```

Podemos agregar `\b` para excluir números largos: `\b\d{5}\b`.

#### El rango: `{3,5}`, coincide 3-5 veces

Para encontrar números de 3 a 5 dígitos, podemos poner los límites en llaves: `\d{3,5}`

```
alert("No tengo 12, sino 1234 años de edad".match(/\d{3,5}/)); // "1234"
```

Podemos omitir el límite superior

Luego, una regexp `\d{3,}` busca secuencias de dígitos de longitud 3 o más:

```
alert("No tengo 12, sino, 345678 años de edad".match(/\d{3,}/)); // "345678"
```

Volvamos a la cadena `+7(903)-123-45-67`.

Un número es una secuencia de uno o más dígitos continuos. Entonces la expresión regular es `\d{1,}`:

```
let str = "+7(903)-123-45-67";
let numbers = str.match(/\d{1,}/g);
alert(numbers); // 7,903,123,45,67
```

## Abreviaciones

Hay abreviaciones para los cuantificadores más usados:

+

Significa “uno o más”, igual que `{1,}`.

Por ejemplo, `\d+` busca números:

```
let str = "+7(903)-123-45-67";
alert(str.match(/\d+/g)); // 7, 903, 123, 45, 67
```

?

Significa “cero o uno”, igual que `{0, 1}`. En otras palabras, hace que el símbolo sea opcional.

Por ejemplo, el patrón `ou?r` busca `o` seguido de cero o uno `u`, y luego `r`.

Entonces, `colou?r` encuentra ambos `color` y `colour`:

```
let str = "¿Debo escribir color o colour?";
alert(str.match(/colou?r/g)); // color, colour
```

\*

Significa “cero o más”, igual que `{0, }`. Es decir, el carácter puede repetirse muchas veces o estar ausente.

Por ejemplo, `\d0*` busca un dígito seguido de cualquier número de ceros (puede ser muchos o ninguno):

```
alert("100 10 1".match(/\d0*/g)); // 100, 10, 1
```

Compáralo con `+` (uno o más):

```
alert("100 10 1".match(/\d0+/g)); // 100, 10
// 1 no coincide, ya que 0+ requiere al menos un cero
```

## Más ejemplos

Los cuantificadores se usan con mucha frecuencia. Sirven como el “bloque de construcción” principal de expresiones regulares complejas, así que veamos más ejemplos.

**Regexp para fracciones decimales (un número con coma flotante):** `\d+\.\d+`

En acción:

```
alert("0 1 12.345 7890".match(/\d+\.\d+/g)); // 12.345
```

**Regexp para una “etiqueta HTML de apertura sin atributos”, tales como `<span>` o `<p>`.**

1. La más simple: `/<[a-z]+>/i`

```
alert("<body> ... </body>".match(/<[a-z]+>/gi)); // <body>
```

La regexp busca el carácter `'<'` seguido de una o más letras latinas, y el carácter `'>'`.

2. Mejorada: `/<[a-z][a-z0-9]*>/i`

De acuerdo al estándar, el nombre de una etiqueta HTML puede tener un dígito en cualquier posición excepto al inicio, tal como `<h1>`.

```
alert("<h1>Hola!</h1>".match(/<[a-z][a-z0-9]*>/gi)); // <h1>
```

**Regexp para “etiquetas HTML de apertura o cierre sin atributos”:** `/<\/?[a-z][a-z0-9]*>/i`

Agregamos una barra opcional `/?` cerca del comienzo del patrón. Se tiene que escapar con una barra diagonal inversa, de lo contrario, JavaScript pensaría que es el final del patrón.

```
alert("<h1>Hola!</h1>".match(/<\w? [a-z][a-z0-9]*>/gi)); // <h1>, </h1>
```

### Para hacer más precisa una regexp, a menudo necesitamos hacerla más compleja

Podemos ver una regla común en estos ejemplos: cuanto más precisa es la expresión regular, es más larga y compleja.

Por ejemplo, para las etiquetas HTML debemos usar una regexp más simple: `<\w+>`. Pero como HTML tiene normas estrictas para los nombres de etiqueta, `<[a-z][a-z0-9]*>` es más confiable.

¿Podemos usar `<\w+>` o necesitamos `<[a-z][a-z0-9]*>`?

En la vida real, ambas variantes son aceptables. Depende de cuán tolerantes podamos ser a las coincidencias "adicionales" y si es difícil o no eliminarlas del resultado por otros medios.

## Tareas

### ¿Cómo encontrar puntos suspensivos "...?"

importancia: 5

Escriba una regexp para encontrar puntos suspensivos: 3 (¿o más?) puntos en una fila.

Revísalo:

```
let regexp = /tu regexp/g;
alert("Hola!... ¿Cómo vas?....".match(regexp)); // ...,
```

### A solución

### Regexp para colores HTML

Escribe una regexp para encontrar colores HTML escritos como `#ABCDEF`: primero `#` y luego 6 caracteres hexadecimales.

Un ejemplo de uso:

```
let regexp = /...tu regexp...
let str = "color:#121212; background-color:#AA00ef bad-colors:f#fddee #fd2 #12345678";
alert(str.match(regexp)) // #121212,#AA00ef
```

P.D. En esta tarea no necesitamos otro formato de color como `#123` o `rgb(1, 2, 3)`, etc.

### A solución

### Cuantificadores codiciosos y perezosos

Los cuantificadores son muy simples a primera vista, pero de hecho pueden ser complicados.

Debemos entender muy bien cómo funciona la búsqueda si planeamos buscar algo más complejo que `/\d+/`.

Tomemos la siguiente tarea como ejemplo.

Tenemos un texto y necesitamos reemplazar todas las comillas " . ." con comillas latinas: « . ». En muchos países los tipógrafos las prefieren.

Por ejemplo: "Hola, mundo" debe convertirse en «Hola, mundo». Existen otras comillas, como „Witaj, Świecie!” (Polaco) o 「你好, 世界」 (Chino), pero para nuestra tarea elegimos « . ».

Lo primero que debe hacer es ubicar las cadenas entre comillas, y luego podemos reemplazarlas.

Una expresión regular como `"/ . +/"` (una comilla, después algo, luego otra comilla) Puede parecer una buena opción, ¡pero no lo es!

Vamos a intentarlo:

```

let regexp = /.+/"g;
let str = 'una "bruja" y su "escoba" son una';
alert(str.match(regexp)); // "bruja" y su "escoba"

```

...¡Podemos ver que no funciona según lo previsto!

En lugar de encontrar dos coincidencias "bruja" y "escoba", encuentra una: "bruja" y su "escoba".

Esto se puede describir como "la codicia es la causa de todo mal".

## Búsqueda codiciosa

Para encontrar una coincidencia, el motor de expresión regular utiliza el siguiente algoritmo:

- Para cada posición en la cadena
  - Prueba si el patrón coincide en esta posición.
  - Si no hay coincidencia, ir a la siguiente posición.

Estas palabras comunes no son tan obvias para determinar por qué la regexp falla, así que elaboraremos el funcionamiento de la búsqueda del patrón ".+".

1. El primer carácter del patrón es una comilla doble "".

El motor de expresión regular intenta encontrarla en la posición cero de la cadena fuente una "bruja" y su "escoba" son una, pero hay una u allí, por lo que inmediatamente no hay coincidencia.

Entonces avanza: va a la siguiente posición en la cadena fuente y prueba encontrar el primer carácter del patrón allí, falla de nuevo, y finalmente encuentra la comilla doble en la 3ra posición:



a "witch" and her "broom" is one

2. La comilla doble es detectada, y después el motor prueba encontrar una coincidencia para el resto del patrón. Prueba ver si el resto de la cadena objetivo satisface a ".+".

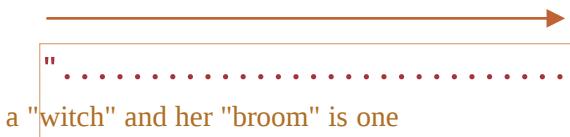
En nuestro caso el próximo carácter de patrón es . (un punto). Que denota "cualquier carácter excepto línea nueva", entonces la próxima letra de la cadena encaja 'w' :



a "witch" and her "broom" is one

3. Entonces el punto (.) se repite por el cuantificador ".+". El motor de expresión regular agrega a la coincidencia un carácter uno después de otro.

...¿Hasta cuando? Todos los caracteres coinciden con el punto, entonces se detiene hasta que alcanza el final de la cadena:



4. Ahora el motor finalizó el ciclo de ".+" y prueba encontrar el próximo carácter del patrón. El cual es la comilla doble "".

Pero hay un problema: la cadena ha finalizado, ¡no hay más caracteres!

El motor de expresión regular comprende que procesó demasiados ".+" y *reinicia* la cadena.

En otras palabras, acorta la coincidencia para el cuantificador en un carácter:



Ahora se supone que `.+` finaliza un carácter antes del final de la cadena e intenta hacer coincidir el resto del patrón desde esa posición.

Si hubiera comillas doble allí, entonces la búsqueda terminaría, pero el último carácter es `'a'`, por lo que no hay coincidencia.

5. ...Entonces el motor disminuye el número de repeticiones de `.+` en uno:



Las comillas dobles `""` no coinciden con `'n'`.

6. El motor continua reiniciando la lectura de la cadena: decrementa el contador de repeticiones para `'.+'` hasta que el resto del patrón (en nuestro caso `'''`) coincide:



7. La coincidencia está completa.

8. Entonces la primera coincidencia es "bruja" y su "escoba". Si la expresión regular tiene la bandera `g`, entonces la búsqueda continuará desde donde termina la primera coincidencia. No hay más comillas dobles en el resto de la cadena `son una`, entonces no hay más resultados.

Probablemente no es lo que esperábamos, pero así es como funciona.

**En el modo codicioso (por defecto) un carácter cuantificado se repite tantas veces como sea posible.**

El motor de regexp agrega a la coincidencia tantos caracteres como pueda abarcar el patrón `.+`, y luego los abrevia uno por uno si el resto del patrón no coincide.

En nuestro caso queremos otra cosa. Es entonces donde el modo perezoso puede ayudar.

## Modo perezoso

El modo perezoso de los cuantificadores es lo opuesto del modo codicioso. Eso significa: "repita el mínimo número de veces".

Podemos habilitarlo poniendo un signo de interrogación `?!` después del cuantificador, entonces tendríamos `*?` o `+?` o incluso `??` para `'?'`.

Aclarando las cosas: generalmente un signo de interrogación `?` es un cuantificador por si mismo (cero o uno), pero si se agrega *después de otro cuantificador (o incluso el mismo)* toma otro significado, alterna el modo de coincidencia de codicioso a perezoso.

La regexp `/" .+?"/g` funciona como se esperaba: encuentra "bruja" y "escoba":

```
let regexp = /".+?"/g;
let str = 'una "bruja" y su "escoba" son una';
alert(str.match(regexp)); // "bruja", "escoba"
```

Para comprender claramente el cambio, rastreemos la búsqueda paso a paso.

1. El primer paso es el mismo: encuentra el inicio del patrón `''''` en la 5ta posición:



2. El siguiente paso también es similar: el motor encuentra una coincidencia para el punto `'.'`:



3. Y ahora la búsqueda es diferente. Porque tenemos el modo perezoso activado en `+?`, el motor no prueba coincidir un punto una vez más, se detiene y prueba coincidir el resto del patrón (`''''`) ahora mismo :



Si hubiera comillas dobles allí, entonces la búsqueda terminaría, pero hay una `'r'`, entonces no hay coincidencia.

4. Después el motor de expresión regular incrementa el número de repeticiones para el punto y prueba una vez más:



Falla de nuevo. Después el número de repeticiones es incrementado una y otra vez...

5. ...Hasta que se encuentre una coincidencia para el resto del patrón:



6. La próxima búsqueda inicia desde el final de la coincidencia actual y produce un resultado más:



En este ejemplo vimos cómo funciona el modo perezoso para `+?`. Los cuantificadores `*?` y `??` funcionan de manera similar, el motor regexp incrementa el número de repeticiones solo si el resto del patrón no coincide en la posición dada.

**La pereza solo está habilitada para el cuantificador con `?`.**

Otros cuantificadores siguen siendo codiciosos.

Por ejemplo:

```
alert("123 456".match(/\d+ \d+?/)); // 123 4
```

- El patrón `\d+` intenta hacer coincidir tantos dígitos como sea posible (modo codicioso), por lo que encuentra `123` y se detiene, porque el siguiente carácter es un espacio `' '`.
- Luego hay un espacio en el patrón, coincide.
- Después hay un `\d+?`. El cuantificador está en modo perezoso, entonces busca un dígito `4` y trata de verificar si el resto del patrón coincide desde allí.  
...Pero no hay nada en el patrón después de `\d+?`.

El modo perezoso no repite nada sin necesidad. El patrón terminó, así que terminamos. Tenemos una coincidencia `123 4`.

### Optimizaciones

Los motores modernos de expresiones regulares pueden optimizar algoritmos internos para trabajar más rápido. Estos trabajan un poco diferente del algoritmo descrito.

Pero para comprender como funcionan las expresiones regulares y construirlas, no necesitamos saber nada al respecto. Solo se usan internamente para optimizar cosas.

Las expresiones regulares complejas son difíciles de optimizar, por lo que la búsqueda también puede funcionar exactamente como se describe.

## Enfoque alternativo

Con las regexps, por lo general hay muchas formas de hacer la misma cosa.

En nuestro caso podemos encontrar cadenas entre comillas sin el modo perezoso usando la regexp `"[^"]+"`:

```
let regexp = /"[^"]+"/g;
let str = 'una "bruja" y su "escoba" son una';
alert(str.match(regexp)); // "bruja", "escoba"
```

La regexp `"[^"]+"` devuelve el resultado correcto, porque busca una comilla doble `" "` seguida por uno o más caracteres no comilla doble `[^"]`, y luego la comilla doble de cierre.

Cuando la máquina de regexp busca el carácter no comilla `[^"]+` se detiene la repetición cuando encuentra la comilla doble de cierre, y terminamos.

Nótese, ¡esta lógica no reemplaza al cuantificador perezoso!

Es solo diferente. Hay momentos en que necesitamos uno u otro.

### Veamos un ejemplo donde los cuantificadores perezosos fallan y la variante funciona correctamente.

Por ejemplo, queremos encontrar enlaces en la forma `<a href="..." class="doc">`, con cualquier `href`.

¿Cuál expresión regular usamos?

La primera idea podría ser: `/<a href=".*" class="doc">/g`.

Veámoslo:

```
let str = '......';
let regexp = /<a href=".*" class="doc"/g;

// ¡Funciona!
alert(str.match(regexp)); //
```

Funcionó. Pero veamos ¿que pasa si hay varios enlaces en el texto?

```
let str = '...... ...';
let regexp = /<a href=".*" class="doc"/g;

// ¡Vaya! ¡Dos enlaces en una coincidencia!
alert(str.match(regexp)); // ...
```

Ahora el resultado es incorrecto por la misma razón del ejemplo de la bruja. El cuantificador `.*` toma demasiados caracteres.

La coincidencia se ve así:

```

...
```

Modifiquemos el patrón haciendo el cuantificador perezoso: `.*?`:

```
let str = '...... ...';
let regexp = //g;

// ¡Funciona!
alert(str.match(regexp)); // ,
```

Ahora parece funcionar, hay dos coincidencias:

```

...
```

...Pero probemos ahora con una entrada de texto adicional:

```
let str = '...... <p style="" class="doc">...';
let regexp = //g;

// ¡Coincidencia incorrecta!
alert(str.match(regexp)); // ... <p style="" class="doc">
```

Ahora falla. La coincidencia no solo incluye el enlace, sino también mucho texto después, incluyendo `<p . . .>`.

¿Por qué?

Eso es lo que está pasando:

1. Primero la regexp encuentra un enlace inicial `<a href="`.
2. Después busca para el patrón `.*?`: toma un carácter (¡perezosamente!), verifica si hay una coincidencia para `" class="doc">` (ninguna).
3. Después toma otro carácter dentro de `.*?`, y así... hasta que finalmente alcanza a `" class="doc">`.

Pero el problema es que: eso ya está más allá del enlace `<a . . .>`, en otra etiqueta `<p>`. No es lo que queremos.

Esta es la muestra de la coincidencia alineada con el texto:

```

... <p style="" class="doc">
```

Entonces, necesitamos un patrón que busque `<a href="...algo..." class="doc">`, pero ambas variantes, codiciosa y perezosa, tienen problemas.

La variante correcta puede ser: `href="[^"]*"`. Esta tomará todos los caracteres dentro del atributo `href` hasta la comilla doble más cercana, justo lo que necesitamos.

Un ejemplo funcional:

```
let str1 = '...... <p style="" class="doc">...';
let str2 = '...... ...';
let regexp = //g;

// ¡Funciona!
alert(str1.match(regexp)); // null, sin coincidencia, eso es correcto
alert(str2.match(regexp)); // ,
```

## Resumen

Los cuantificadores tienen dos modos de funcionamiento:

### Codicioso

Por defecto el motor de expresión regular prueba repetir el carácter cuantificado tantas veces como sea posible. Por ejemplo, `\d+` consume todos los posibles dígitos. Cuando es imposible consumir más (no hay más dígitos o es el fin de la cadena), entonces continúa hasta coincidir con el resto del patrón. Si no hay coincidencia entonces se decrementa el número de repeticiones (reinicios) y prueba de nuevo.

### Perezoso

Habilitado por el signo de interrogación `?` después de un cuantificador. El motor de regexp prueba la coincidencia para el resto del patrón antes de cada repetición del carácter cuantificado.

Como vimos, el modo perezoso no es una “panacea” de la búsqueda codiciosa. Una alternativa es una búsqueda codiciosa refinada, con exclusiones, como en el patrón `"[^"]+"`.

## ✓ Tareas

---

### Una coincidencia para /d+? d+?/

¿Cuál es la coincidencia aquí?

```
alert("123 456".match(/\d+? \d+?/g)); // ?
```

[A solución](#)

---

### Encuentra el comentario HTML

Encuentra todos los comentarios HTML en el texto:

```
let regexp = /your regexp/g;

let str = `... <!-- Mi -- comentario
prueba --> .. <!----> ..`;

alert(str.match(regexp)); // '<!-- Mi -- comentario \n prueba -->', '<!---->'
```

[A solución](#)

---

### Encontrar las etiquetas HTML

Crear una expresión regular para encontrar todas las etiquetas HTML (de apertura y cierre) con sus atributos.

Un ejemplo de uso:

```
let regexp = /tu regexp/g;

let str = '<> <input type="radio" checked> ';

alert(str.match(regexp)); // '', '<input type="radio" checked>', ''
```

Asumimos que los atributos de etiqueta no deben contener `<` ni `>` (dentro de comillas dobles también), esto simplifica un poco las cosas.

[A solución](#)

---

## Grupos de captura

Una parte de un patrón se puede incluir entre paréntesis `( . . . )`. Esto se llama “grupo de captura”.

Esto tiene dos resultados:

1. Permite obtener una parte de la coincidencia como un elemento separado en la matriz de resultados.
2. Si colocamos un cuantificador después del paréntesis, se aplica a los paréntesis en su conjunto.

## Ejemplos

Veamos cómo funcionan los paréntesis en los ejemplos.

### Ejemplo: gogogo

Sin paréntesis, el patrón `go+` significa el carácter `g`, seguido por `o` repetido una o más veces. Por ejemplo, `goooo` o `oooooooooo`.

Los paréntesis agrupan los caracteres juntos, por lo tanto `(go)+` significa `go`, `gogo`, `gogogo` etcétera.

```
alert('Gogogo now!'.match(/(go)+/ig)); // "Gogogo"
```

### Ejemplo: dominio

Hagamos algo más complejo: una expresión regular para buscar un dominio de sitio web.

Por ejemplo:

```
mail.com
users.mail.com
smith.users.mail.com
```

Como podemos ver, un dominio consta de palabras repetidas, un punto después de cada una excepto la última.

En expresiones regulares eso es `(\w+\. )+\w+`:

```
let regexp = /(\w+\.)+\w+/g;

alert("site.com my.site.com".match(regexp)); // site.com,my.site.com
```

La búsqueda funciona, pero el patrón no puede coincidir con un dominio con un guión, por ejemplo, `my-site.com`, porque el guión no pertenece a la clase `\w`.

Podemos arreglarlo al reemplazar `\w` con `[\w-]` en cada palabra excepto el último: `( [\w- ]+\. )+\w+`.

### Ejemplo: email

El ejemplo anterior puede ser extendido. Podemos crear una expresión regular para emails en base a esto.

El formato de email es: `name@domain`. Cualquier palabra puede ser el nombre, guiones y puntos están permitidos. En expresiones regulares esto es `[ - . \w ] +`.

El patrón:

```
let regexp = /[- . \w] +@[- . \w -] +\.[- . \w -] +/g;

alert("my@mail.com @ his@site.com.uk".match(regexp)); // my@mail.com, his@site.com.uk
```

Esa expresión regular no es perfecta, pero sobre todo funciona y ayuda a corregir errores de escritura accidentales. La única verificación verdaderamente confiable para un correo electrónico solo se puede realizar enviando una carta.

## Contenido del paréntesis en la coincidencia (match)

Los paréntesis están numerados de izquierda a derecha. El buscador memoriza el contenido que coincide con cada uno de ellos y permite obtenerlo en el resultado.

El método `str.match(regexp)`, si `regexp` no tiene indicador (flag) `g`, busca la primera coincidencia y lo devuelve como un array:

1. En el índice `0`: la coincidencia completa.
2. En el índice `1`: el contenido del primer paréntesis.

3. En el índice `2` : el contenido del segundo paréntesis.

4. ...etcétera...

Por ejemplo, nos gustaría encontrar etiquetas HTML `<. *>`, y procesarlas. Sería conveniente tener el contenido de la etiqueta (lo que está dentro de los ángulos), en una variable por separado.

Envolvamos el contenido interior en paréntesis, de esta forma: `<( . *?)>`.

Ahora obtendremos ambos, la etiqueta entera `<h1>` y su contenido `h1` en el array resultante:

```
let str = '<h1>Hello, world!</h1>';

let tag = str.match(/<(.*?)>/);

alert(tag[0]); // <h1>
alert(tag[1]); // h1
```

## Grupos anidados

Los paréntesis pueden ser anidados. En este caso la numeración también va de izquierda a derecha.

Por ejemplo, al buscar una etiqueta en `<span class="my">` tal vez nos pueda interesar:

1. El contenido de la etiqueta como un todo: `span class="my"`.
2. El nombre de la etiqueta: `span`.
3. Los atributos de la etiqueta: `class="my"`.

Agreguemos paréntesis: `<(( [a-z]+)\s*([>]*))>`.

Así es cómo se enumeran (izquierda a derecha, por el paréntesis de apertura):

```
span class="my"
1 _____
<(([a-z]+)\s* ([>]*)) >
 2 _____ 3 _____
 span class="my"
```

En acción:

```
let str = '';

let regexp = /<(([a-z]+)\s*([>]*))>/;

let result = str.match(regexp);
alert(result[0]); //
alert(result[1]); // span
alert(result[2]); // [a-z]+
alert(result[3]); // \s*
```

El índice cero de `result` siempre contiene la coincidencia completa.

Luego los grupos, numerados de izquierda a derecha por un paréntesis de apertura. El primer grupo se devuelve como `result[1]`. Aquí se encierra todo el contenido de la etiqueta.

Luego en `result[2]` va el grupo desde el segundo paréntesis de apertura (`[a-z]+` – nombre de etiqueta, luego en `result[3]` la etiqueta: `( [>]* )`).

El contenido de cada grupo en el string:

```
span class="my"
1 _____
<(([a-z]+)\s* ([>]*)) >
 2 _____ 3 _____
 span class="my"
```

## Grupos opcionales

Incluso si un grupo es opcional y no existe en la coincidencia (p.ej. tiene el cuantificador `( . . . )?`), el elemento array `result` correspondiente está presente y es igual a `undefined`.

Por ejemplo, consideremos la expresión regular `a(z)?(c)?`. Busca "a" seguida por opcionalmente "z", seguido por "c" opcionalmente.

Si lo ejecutamos en el string con una sola letra `a`, entonces el resultado es:

```
let match = 'a'.match(/a(z)?(c)?/);

alert(match.length); // 3
alert(match[0]); // a (coincidencia completa)
alert(match[1]); // undefined
alert(match[2]); // undefined
```

El array tiene longitud de `3`, pero todos los grupos están vacíos.

Y aquí hay una coincidencia más compleja para el string `ac`:

```
let match = 'ac'.match(/a(z)?(c)?/)

alert(match.length); // 3
alert(match[0]); // ac (coincidencia completa)
alert(match[1]); // undefined, ¿porque no hay nada para (z)?
alert(match[2]); // c
```

La longitud del array es permanente: `3`. Pero no hay nada para el grupo `(z)?`, por lo tanto el resultado es `["ac", undefined, "c"]`.

## Buscar todas las coincidencias con grupos: `matchAll`

 **matchAll** es un nuevo método, polyfill puede ser necesario

El método `matchAll` no es compatible con antiguos navegadores.

Un polyfill puede ser requerido, tal como [https://github.com/ljharb/String.prototype.matchAll ↗](https://github.com/ljharb/String.prototype.matchAll).

Cuando buscamos todas las coincidencias (flag `g`), el método `match` no devuelve contenido para los grupos.

Por ejemplo, encontraremos todas las etiquetas en un string:

```
let str = '<h1> <h2>';

let tags = str.match(/<.*?>/g);

alert(tags); // <h1>,<h2>
```

El resultado es un array de coincidencias, pero sin detalles sobre cada uno de ellos. Pero en la práctica normalmente necesitamos contenidos de los grupos de captura en el resultado.

Para obtenerlos tenemos que buscar utilizando el método `str.matchAll(regexp)`.

Fue incluido a JavaScript mucho después de `match`, como su versión "nueva y mejorada".

Al igual que `match`, busca coincidencias, pero hay 3 diferencias:

1. No devuelve un array sino un objeto iterable.
2. Cuando está presente el indicador `g`, devuelve todas las coincidencias como un array con grupos.
3. Si no hay coincidencias, no devuelve `null` sino un objeto iterable vacío.

Por ejemplo:

```
let results = '<h1> <h2>'.matchAll(/<.*?>/gi);

// results - no es un array, sino un objeto iterable
alert(results); // [object RegExp String Iterator]
```

```

alert(results[0]); // undefined (*)

results = Array.from(results); // lo convertamos en array

alert(results[0]); // <h1>,h1 (1er etiqueta)
alert(results[1]); // <h2>,h2 (2da etiqueta)

```

Como podemos ver, la primera diferencia es muy importante, como se demuestra en la línea `(* )`. No podemos obtener la coincidencia como `results[0]`, porque ese objeto no es pseudo array. Lo podemos convertir en un `Array` real utilizando `Array.from`. Hay más detalles sobre pseudo arrays e iterables en el artículo. [Iterables](#).

No se necesita `Array.from` si estamos iterando sobre los resultados:

```

let results = '<h1> <h2>'.matchAll(/<(.+?)>/gi);

for(let result of results) {
 alert(result);
 // primer alert: <h1>,h1
 // segundo: <h2>,h2
}

```

...O utilizando desestructurización:

```

let [tag1, tag2] = '<h1> <h2>'.matchAll(/<(.+?)>/gi);

```

Cada coincidencia devuelta por `matchAll` tiene el mismo formato que el devuelto por `match` sin el flag `g`: es un array con propiedades adicionales `index` (coincide índice en el string) e `input` (fuente string):

```

let results = '<h1> <h2>'.matchAll(/<(.+?)>/gi);

let [tag1, tag2] = results;

alert(tag1[0]); // <h1>
alert(tag1[1]); // h1
alert(tag1.index); // 0
alert(tag1.input); // <h1> <h2>

```

### **i** ¿Por qué el resultado de `matchAll` es un objeto iterable y no un array?

¿Por qué el método está diseñado de esa manera? La razón es simple – por la optimización.

El llamado a `matchAll` no realiza la búsqueda. En cambio devuelve un objeto iterable, en un principio sin los resultados. La búsqueda es realizada cada vez que iteramos sobre ella, es decir, en el bucle.

Por lo tanto, se encontrará tantos resultados como sea necesario, no más.

Por ejemplo, posiblemente hay 100 coincidencias en el texto, pero en un bucle `for .. of` encontramos 5 de ellas: entonces decidimos que es suficiente y realizamos un `break`. Así el buscador no gastará tiempo buscando otras 95 coincidencias.

## Grupos con nombre

Es difícil recordar a los grupos por su número. Para patrones simples, es factible, pero para los más complejos, contar los paréntesis es inconveniente. Tenemos una opción mucho mejor: poner nombres entre paréntesis.

Eso se hace poniendo `?<name>` inmediatamente después del paréntesis de apertura.

Por ejemplo, busquemos una fecha en el formato “año-mes-día”:

```

let dateRegexp = /(?<year>[0-9]{4})-(?<month>[0-9]{2})-(?<day>[0-9]{2})/;
let str = "2019-04-30";

let groups = str.match(dateRegexp).groups;

alert(groups.year); // 2019

```

```
alert(groups.month); // 04
alert(groups.day); // 30
```

Como puedes ver, los grupos residen en la propiedad `.groups` de la coincidencia.

Para buscar todas las fechas, podemos agregar el flag `g`.

También vamos a necesitar `matchAll` para obtener coincidencias completas, junto con los grupos:

```
let dateRegexp = /(?<year>[0-9]{4})-(?<month>[0-9]{2})-(?<day>[0-9]{2})/g;

let str = "2019-10-30 2020-01-01";

let results = str.matchAll(dateRegexp);

for(let result of results) {
 let {year, month, day} = result.groups;

 alert(`#${day}.${month}.${year}`);
 // primer alert: 30.10.2019
 // segundo: 01.01.2020
}
```

## Grupos de captura en reemplazo

El método `str.replace(regexp, replacement)` que reemplaza todas las coincidencias con `regexp` en `str` nos permite utilizar el contenido de los paréntesis en el string `replacement`. Esto se hace utilizando `$n`, donde `n` es el número de grupo.

Por ejemplo,

```
let str = "John Bull";
let regexp = /(\w+) (\w+)/;

alert(str.replace(regexp, '$2, $1')); // Bull, John
```

Para los paréntesis con nombre la referencia será `$<name>`.

Por ejemplo, volvamos a darle formato a las fechas desde “year-month-day” a “day.month.year”:

```
let regexp = /(?<year>[0-9]{4})-(?<month>[0-9]{2})-(?<day>[0-9]{2})/g;

let str = "2019-10-30, 2020-01-01";

alert(str.replace(regexp, '$<day>.$<month>.$<year>'));
// 30.10.2019, 01.01.2020
```

## Grupos que no capturan con ?:

A veces necesitamos paréntesis para aplicar correctamente un cuantificador, pero no queremos su contenido en los resultados.

Se puede excluir un grupo agregando `?:` al inicio.

Por ejemplo, si queremos encontrar `(go)+`, pero no queremos el contenido del paréntesis (`go`) como un ítem separado del array, podemos escribir: `(?:go)+`.

En el ejemplo de arriba solamente obtenemos el nombre `John` como un miembro separado de la coincidencia:

```
let str = "Gogogo John!";

// ?: excluye 'go' de la captura
let regexp = /(?:go)+ (\w+)/i;

let result = str.match(regexp);

alert(result[0]); // Gogogo John (coincidencia completa)
```

```
alert(result[1]); // John
alert(result.length); // 2 (no hay más ítems en el array)
```

## Resumen

Los paréntesis agrupan una parte de la expresión regular, de modo que el cuantificador se aplique a ella como un todo.

Los grupos de paréntesis se numeran de izquierda a derecha y, opcionalmente, se pueden nombrar con `(?<name>...)`.

El contenido, emparejado por un grupo, se puede obtener en los resultados:

- El método `str.match` devuelve grupos de captura únicamente sin el indicador (flag) `g`.
- El método `str.matchAll` siempre devuelve grupos de captura.

Si el paréntesis no tiene nombre, entonces su contenido está disponible en el array de coincidencias por su número. Los paréntesis con nombre también están disponibles en la propiedad `groups`.

También podemos utilizar el contenido del paréntesis en el string de reemplazo de `str.replace`: por el número `$n` o el nombre `$<name>`.

Un grupo puede ser excluido de la enumeración al agregar `?:` en el inicio. Eso se usa cuando necesitamos aplicar un cuantificador a todo el grupo, pero no lo queremos como un elemento separado en el array de resultados. Tampoco podemos hacer referencia a tales paréntesis en el string de reemplazo.

## ✓ Tareas

### Verificar dirección MAC

La [Dirección MAC](#) de una interfaz de red consiste en 6 números hexadecimales de dos dígitos separados por dos puntos.

Por ejemplo: `'01:32:54:67:89:AB'`.

Escriba una expresión regular que verifique si una cadena es una Dirección MAC.

Uso:

```
let regexp = /your regexp/;

alert(regexp.test('01:32:54:67:89:AB')); // true

alert(regexp.test('0132546789AB')); // false (sin dos puntos)

alert(regexp.test('01:32:54:67:89')); // false (5 números, necesita 6)

alert(regexp.test('01:32:54:67:89:ZZ')); // false (ZZ al final)
```

### A solución

### Encuentra el color en el formato #abc o #abcdef

Escriba una expresión regular que haga coincidir los colores en el formato `#abc` o `#abcdef`. Esto es: `#` seguido por 3 o 6 dígitos hexadecimales.

Ejemplo del uso:

```
let regexp = /your regexp/g;

let str = "color: #3f3; background-color: #AA00ef; and: #abcd";

alert(str.match(regexp)); // #3f3 #AA00ef
```

P.D. Esto debe ser exactamente 3 o 6 dígitos hexadecimales. Valores con 4 dígitos, tales como `#abcd`, no deben coincidir.

### A solución

## Encuentre todos los números

Escribe una expresión regular que busque todos los números decimales, incluidos los enteros, con el punto flotante y los negativos.

Un ejemplo de uso:

```
let regexp = /your regexp/g;
let str = "-1.5 0 2 -123.4.";
alert(str.match(regexp)); // -1.5, 0, 2, -123.4
```

A solución

## Analizar una expresión:

Una expresión aritmética consta de 2 números y un operador entre ellos, por ejemplo:

- `1 + 2`
- `1.2 * 3.4`
- `-3 / -6`
- `-2 - 2`

El operador es uno de estos: `"+"`, `"+"`, `"*"` o `"/"`.

Puede haber espacios adicionales al principio, al final o entre las partes.

Crea una función `parse(expr)` que tome una expresión y devuelva un array de 3 ítems:

1. El primer número.
2. El operador.
3. El segundo número.

Por ejemplo:

```
let [a, op, b] = parse("1.2 * 3.4");

alert(a); // 1.2
alert(op); // *
alert(b); // 3.4
```

A solución

## Referencias inversas en patrones: `\N` y `\k<nombre>`

Podemos utilizar el contenido de los grupos de captura `( . . . )` no solo en el resultado o en la cadena de reemplazo, sino también en el patrón en sí.

### Referencia inversa por número: `\N`

Se puede hacer referencia a un grupo en el patrón usando `\N`, donde `N` es el número de grupo.

Para aclarar por qué es útil, consideremos una tarea.

Necesitamos encontrar una cadena entre comillas: con cualquiera de los dos tipos, comillas simples `' . . . '` o comillas dobles `" . . . "` – ambas variantes deben coincidir.

¿Cómo encontrarlas?

Ambos tipos de comillas se pueden poner entre corchetes: `[ ' " ](. *?) [ ' "]`, pero encontrará cadenas con comillas mixtas, como `" . . . '` y `' . . . "`. Eso conduciría a coincidencias incorrectas cuando una cita aparece dentro de otra., como en la

cadena "She's the one!" (en este ejemplo los strings no se traducen por el uso de la comilla simple):

```
let str = `He said: "She's the one!".`;
let regexp = /["](.*?)["]/g;

// El resultado no es el que nos gustaría tener
alert(str.match(regexp)); // "She"
```

Como podemos ver, el patrón encontró una cita abierta " , luego se consume el texto hasta encontrar la siguiente comilla " , esta cierra la coincidencia.

Para asegurar que el patrón busque la comilla de cierre exactamente igual que la de apertura, se pone dentro de un grupo de captura y se hace referencia inversa al 1ero: (["])(.\*?)\1 .

Aquí está el código correcto:

```
let str = `He said: "She's the one!".`;
let regexp = /(["])(.*?)\1/g;

alert(str.match(regexp)); // "She's the one!"
```

¡Ahora funciona! El motor de expresiones regulares encuentra la primera comilla ([ ]) y memoriza su contenido. Este es el primer grupo de captura.

Continuando en el patrón, \1 significa “encuentra el mismo texto que en el primer grupo”, en nuestro caso exactamente la misma comilla.

Similar a esto, \2 debería significar: el contenido del segundo grupo, \3 – del tercer grupo, y así sucesivamente.

#### **i Por favor tome nota:**

Si usamos ?: en el grupo, entonces no lo podremos referenciar. Los grupos que se excluyen de las capturas (?:...) no son memorizados por el motor.

#### **⚠️ No confundas: el patrón \1 , con el reemplazo: \$1**

En el reemplazo de cadenas usamos el signo dólar: \$1 , mientras que en el patrón – una barra invertida \1 .

## Referencia inversa por nombre: \k<nombre>

Si una regexp tiene muchos paréntesis, es conveniente asignarle nombres.

Para referenciar un grupo con nombre usamos \k<nombre> .

En el siguiente ejemplo, el grupo con comillas se llama ?<quote> , entonces la referencia inversa es \k<quote> :

```
let str = `He said: "She's the one!".`;
let regexp = /(<quote>["])(.*?)\k<quote>/g;

alert(str.match(regexp)); // "She's the one!"
```

## Alternancia (O) |

Alternancia es un término en expresión regular que simplemente significa “O”.

En una expresión regular se denota con un carácter de línea vertical | .

Por ejemplo, necesitamos encontrar lenguajes de programación: HTML, PHP, Java o JavaScript.

La expresión regular correspondiente es: html|php|java(script)? .

Un ejemplo de uso:

```

let regexp = /html|php|css|java(script)?/gi;
let str = "Primera aparición de HTML, luego CSS, luego JavaScript";
alert(str.match(regexp)); // 'HTML', 'CSS', 'JavaScript'

```

Ya vimos algo similar: corchetes. Permiten elegir entre varios caracteres, por ejemplo `gr[ae]y` coincide con `gray` o `grey`.

Los corchetes solo permiten caracteres o conjuntos de caracteres. La alternancia permite cualquier expresión. Una expresión regular `A|B|C` significa una de las expresiones `A`, `B` o `C`.

Por ejemplo:

- `gr(a|e)y` significa exactamente lo mismo que `gr[ae]y`.
- `gra|ey` significa `gra` o `ey`.

Para aplicar la alternancia a una parte elegida del patrón, podemos encerrarla entre paréntesis:

- `I love HTML|CSS` coincide con `I love HTML` o `CSS`.
- `I love (HTML|CSS)` coincide con `I love HTML` o `I love CSS`.

## Ejemplo: Expresión regular para el tiempo

En artículos anteriores había una tarea para construir una expresión regular para buscar un horario en la forma `hh:mm`, por ejemplo `12:00`. Pero esta simple expresión `\d\d:\d\d` es muy vaga. Acepta `25:99` como tiempo (ya que 99 segundos coinciden con el patrón, pero ese tiempo no es válido).

¿Cómo podemos hacer un mejor patrón?

Podemos utilizar una combinación más cuidadosa. Primero, las horas:

- Si el primer dígito es `0` o `1`, entonces el siguiente dígito puede ser cualquiera: `[01]\d`.
- De otra manera, si el primer dígito es `2`, entonces el siguiente debe ser `[0-3]`.
- (no se permite ningún otro dígito)

Podemos escribir ambas variantes en una expresión regular usando alternancia: `[01]\d|2[0-3]`.

A continuación, los minutos deben estar comprendidos entre `00` y `59`. En el lenguaje de expresiones regulares se puede escribir como `[0-5]\d`: el primer dígito `0-5`, y luego cualquier otro.

Si pegamos minutos y segundos juntos, obtenemos el patrón: `[01]\d|2[0-3]:[0-5]\d`.

Ya casi terminamos, pero hay un problema. La alternancia `|` ahora pasa a estar entre `[01]\d` y `2[0-3]:[0-5]\d`.

Es decir: se agregan minutos a la segunda variante de alternancia, aquí hay una imagen clara:

```
[01]\d | 2[0-3]:[0-5]\d
```

Este patrón busca `[01]\d` o `2[0-3]:[0-5]\d`.

Pero eso es incorrecto, la alternancia solo debe usarse en la parte “horas” de la expresión regular, para permitir `[01]\d` o `2[0-3]`. Corregiremos eso encerrando las “horas” entre paréntesis: `([01]\d|2[0-3]):[0-5]\d`.

La solución final sería:

```

let regexp = /([01]\d|2[0-3]):[0-5]\d/g;
alert("00:00 10:10 23:59 25:99 1:2".match(regexp)); // 00:00,10:10,23:59

```

## ✓ Tareas

### Encuentra lenguajes de programación

Hay muchos lenguajes de programación, por ejemplo, Java, JavaScript, PHP, C, C++.

Crea una expresión regular que los encuentre en la cadena `Java JavaScript PHP C++ C`:

```
let regexp = /your regexp/g;

alert("Java JavaScript PHP C++ C".match(regexp)); // Java JavaScript PHP C++ C
```

## A solución

### Encuentra la pareja bbtag

Un "bb-tag" se ve como `[tag]...[/tag]`, donde `tag` es uno de: `b`, `url` o `quote`.

Por ejemplo:

```
[b]text[/b]
[url]http://google.com[/url]
```

BB-tags se puede anidar. Pero una etiqueta no se puede anidar en sí misma, por ejemplo:

```
Normal:
[url] [b]http://google.com[/b] [/url]
[quote] [b]text[/b] [/quote]
```

```
No puede suceder:
[b][b]text[/b][/b]
```

Las etiquetas pueden contener saltos de línea, eso es normal:

```
[quote]
 [b]text[/b]
[/quote]
```

Cree una expresión regular para encontrar todas las BB-tags con su contenido.

Por ejemplo:

```
let regexp = /your regexp/flags;

let str = "...[url]http://google.com[/url]...";
alert(str.match(regexp)); // [url]http://google.com[/url]
```

Si las etiquetas están anidadas, entonces necesitamos la etiqueta externa (si queremos podemos continuar la búsqueda en su contenido):

```
let regexp = /your regexp/flags;

let str = "...[url][b]http://google.com[/b][/url]...";
alert(str.match(regexp)); // [url][b]http://google.com[/b][/url]
```

## A solución

### Encuentra cadenas entre comillas

Crea una expresión regular para encontrar cadenas entre comillas dobles `"..."`.

Las cadenas deben admitir el escape, de la misma manera que lo hacen las cadenas de JavaScript. Por ejemplo, las comillas se pueden insertar como `\\"`, una nueva línea como `\n`, y la barra invertida misma como `\\\`.

```
let str = "Just like \"here\".";
```

Tenga en cuenta, en particular, que una comilla escapada `\\"` no termina una cadena.

Por lo tanto, deberíamos buscar de una comilla a otra (la de cierre), ignorando las comillas escapadas en el camino.

Esa es la parte esencial de la tarea, de lo contrario sería trivial.

Ejemplos de cadenas para hacer coincidir:

```
.. "test me" ..
.. "Say \"Hello\"!" .. (comillas escapadas dentro)
.. "\\\" .. (doble barra invertida dentro)
.. "\\ \\\"" .. (doble barra y comilla escapada dentro.)
```

En JavaScript, necesitamos duplicar las barras para pasárselas directamente a la cadena, así:

```
let str = ' .. "test me" .. "Say \\\"Hello\\\"!" .. \"\\\\\\ \\\"\" .. ';

// the in-memory string
alert(str); // .. "test me" .. "Say \"Hello\"!" .. "\\ \\\"\" ..
```

A solución

## Encuentra la etiqueta completa

Escriba una expresión regular para encontrar la etiqueta `<style...>`. Debe coincidir con la etiqueta completa: puede no tener atributos `<style>` o tener varios de ellos `<style type="..." id="...">`.

...¡Pero la expresión regular no debería coincidir con `<styler>`!

Por ejemplo:

```
let regexp = /your regexp/g;

alert('<style> <styler> <style test="...">>'.match(regexp)); // <style>, <style test="...">>
```

A solución

## Lookahead y lookbehind (revisar delante/detrás)

A veces necesitamos buscar únicamente aquellas coincidencias donde un patrón es precedido o seguido por otro patrón.

Existe una sintaxis especial para eso llamadas “lookahead” y “lookbehind” (“ver delante” y “ver detrás”), juntas son conocidas como “lookaround” (“ver alrededor”).

Para empezar, busquemos el precio de la cadena siguiente 1 pavo cuesta 30€. Eso es: un número, seguido por el signo €.

### Lookahead

La sintaxis es: `X(?=Y)`. Esto significa “buscar `X`, pero considerarlo una coincidencia solo si es seguido por `Y`”. Puede haber cualquier patrón en `X` y `Y`.

Para un número entero seguido de €, la expresión regular será `\d+(?=€)`:

```
let str = "1 pavo cuesta 30€";

alert(str.match(/^\d+(?=€)/)); // 30, el número 1 es ignorado porque no está seguido de €
```

Tenga en cuenta que “lookahead” es solamente una prueba, lo contenido en los paréntesis `(?=...)` no es incluido en el resultado `30`.

Cuando buscamos `X(?=Y)`, el motor de expresión regular encuentra `X` y luego verifica si existe `Y` inmediatamente después de él. Si no existe, entonces la coincidencia potencial es omitida y la búsqueda continúa.

Es posible realizar pruebas más complejas, por ejemplo `X(?=Y)(?=Z)` significa:

1. Encuentra `X`.

2. Verifica si Y está inmediatamente después de X (omite si no es así).
3. Verifica si Z está también inmediatamente después de X (omite si no es así).
4. Si ambas verificaciones se cumplen, el X es una coincidencia. De lo contrario continúa buscando.

En otras palabras, dicho patrón significa que estamos buscando por X seguido de Y y Z al mismo tiempo.

Eso es posible solamente si los patrones Y y Z no se excluyen mutuamente.

Por ejemplo, `\d+(?=\\s)(?=.*30)` busca un \d+ que sea seguido por un espacio `(?=\\s)` y que también tenga un `30` en algún lugar después de él `(?=.*30)`:

```
let str = "1 pavo cuesta 30€";
alert(str.match(/\d+(?=\\s)(?=.*30)/)); // 1
```

En nuestra cadena eso coincide exactamente con el número `1`.

## Lookahead negativo

Digamos que queremos una cantidad, no un precio de la misma cadena. Eso es el número \d+ NO seguido por €.

Para eso se puede aplicar un “lookahead negativo”.

La sintaxis es: `X(?![Y])`, que significa “busca X, pero solo si no es seguido por Y”.

```
let str = "2 pavos cuestan 60€";
alert(str.match(/\d+\\b(?![€])/g)); // 2 (el precio es omitido)
```

## Lookbehind

### Compatibilidad de navegadores en lookbehind

Ten en cuenta: Lookbehind no está soportado en navegadores que no utilizan V8, como Safari, Internet Explorer.

“lookahead” permite agregar una condición para “lo que sigue”.

“Lookbehind” es similar. Permite coincidir un patrón solo si hay algo anterior a él.

La sintaxis es:

- Lookbehind positivo: `(?<=Y)X`, coincide X, pero solo si hay Y antes de él.
- Lookbehind negativo: `(?<!Y)X`, coincide X, pero solo si no hay Y antes de él.

Por ejemplo, cambiamos el precio a dólares estadounidenses. El signo de dólar usualmente va antes del número, entonces para buscar `$30` usaremos `(?<=\$)\\d+`: una cantidad precedida por \$:

```
let str = "1 pavo cuesta $30";
// el signo de dólar se ha escapado \\
alert(str.match(/(?<=\$)\\d+/)); // 30 (omite los números aislados)
```

Y si necesitamos la cantidad (un número no precedida por \$), podemos usar “lookbehind negativo” `(?<!\\$)\\d+`:

```
let str = "2 pavos cuestan $60";
alert(str.match(/(?<!\\$)\\d+/g)); // 2 (el precio es omitido)
```

## Atrapando grupos

Generalmente, los contenidos dentro de los paréntesis de “lookaround” (ver alrededor) no se convierten en parte del resultado.

Ejemplo en el patrón `\d+(?=€)`, el signo `€` no es capturado como parte de la coincidencia. Eso es esperado: buscamos un número `\d+`, mientras `(?=€)` es solo una prueba que indica que debe ser seguida por `€`.

Pero en algunas situaciones nosotros podríamos querer capturar también la expresión en "lookaround", o parte de ella. Eso es posible: solo hay que rodear esa parte con paréntesis adicionales.

En los ejemplos de abajo el signo de divisa `(€|kr)` es capturado junto con la cantidad:

```
let str = "1 pavo cuesta 30€";
let regexp = /\d+(?=($|kr))/; // paréntesis extra alrededor de €|kr

alert(str.match(regexp)); // 30, €
```

Lo mismo para "lookbehind":

```
let str = "1 pavo cuesta $30";
let regexp = /(?(?<=($|£))\d+);

alert(str.match(regexp)); // 30, $
```

## Resumen

Lookahead y lookbehind (en conjunto conocidos como "lookaround") son útiles cuando queremos hacer coincidir algo dependiendo del contexto antes/después.

Para expresiones regulares simples podemos hacer lo mismo manualmente. Esto es: coincidir todo, en cualquier contexto, y luego filtrar por contexto en el bucle.

Recuerda, `str.match` (sin el indicador `g`) y `str.matchAll` (siempre) devuelven las coincidencias como un array con la propiedad `index`, así que sabemos exactamente dónde están dentro del texto y podemos comprobar su contexto.

Pero generalmente "lookaround" es más conveniente.

Tipos de "lookaround":

Patrón	Tipo	Coincidencias
X(?=Y)	lookahead positivo	X si está seguido por Y
X(?!Y)	lookahead negativo	X si no está seguido por Y
(?<=Y)X	lookbehind positivo	X si está después de Y
(?<!Y)X	lookbehind negativo	X si no está después de Y

## ✓ Tareas

### Encontrar enteros no negativos

Tenemos un string de números enteros.

Crea una expresión regular que encuentre solamente los no negativos (el cero está permitido).

Un ejemplo de uso:

```
let regexp = /tu regexp/g;

let str = "0 12 -5 123 -18";

alert(str.match(regexp)); // 0, 12, 123
```

## A solución

### Insertar después de la cabecera

Tenemos un string con un documento HTML.

Escribe una expresión regular que inserte `<h1>Hello</h1>` inmediatamente después de la etiqueta `<body>`. La etiqueta puede tener atributos.

Por ejemplo:

```
let regexp = /tu expresión regular/;

let str = `
<html>
 <body style="height: 200px">
 ...
 </body>
</html>
`;

str = str.replace(regexp, `<h1>Hello</h1>`);
```

Después de esto el valor de `str` debe ser:

```
<html>
 <body style="height: 200px"><h1>Hello</h1>
 ...
 </body>
</html>
```

## A solución

## Backtracking catastrófico

Algunas expresiones regulares parecen simples, pero pueden ejecutarse durante demasiado tiempo e incluso “colgar” el motor de JavaScript.

Tarde o temprano la mayoría de los desarrolladores se enfrentan ocasionalmente a este comportamiento. El síntoma típico: una expresión regular funciona bien a veces, pero para ciertas cadenas se “cuelga” consumiendo el 100% de la CPU.

En este caso el navegador sugiere matar el script y recargar la página. No es algo bueno, sin duda.

Para el lado del servidor de JavaScript tal regexp puede colgar el proceso del servidor, que es aún peor. Así que definitivamente deberíamos echarle un vistazo.

## Ejemplo

Supongamos que tenemos una cadena y queremos comprobar si está formada por palabras `\w+` con un espacio opcional `\s?` después de cada una.

Una forma obvia de construir una regexp sería tomar una palabra seguida de un espacio opcional `\w+\s?` y luego repetirla con `*`.

Esto nos lleva a la regexp `^( \w+\s?)*$` que especifica cero o más palabras de este tipo, que comienzan al principio `^` y terminan al final `$` de la línea.

En la práctica:

```
let regexp = /^(\w+\s?)*$/;

alert(regexp.test("A good string")); // true
alert(regexp.test("Bad characters: $@#")); // false
```

La regexp parece funcionar. El resultado es correcto. Aunque en ciertas cadenas tarda mucho tiempo. Tanto tiempo que el motor de JavaScript se “cuelga” con un consumo del 100% de la CPU.

Si ejecuta el ejemplo de abajo probablemente no se verá nada ya que JavaScript simplemente se “colgará”. El navegador dejará de reaccionar a los eventos, la interfaz de usuario dejará de funcionar (la mayoría de los navegadores sólo permiten el desplazamiento). Después de algún tiempo se sugerirá recargar la página. Así que ten cuidado con esto:

```

let regexp = /^(\\w+\\s?)*$/;
let str = "An input string that takes a long time or even makes this regexp hang!";

// tardará mucho tiempo
alert(regexp.test(str));

```

Para ser justos observemos que algunos motores de expresión regular pueden manejar este tipo de búsqueda con eficacia, por ejemplo, la versión del motor V8 a partir de la 8.8 puede hacerlo (por lo que Google Chrome 88 no se cuelga aquí) mientras que el navegador Firefox sí se cuelga.

## Ejemplo simplificado

¿Qué ocurre? ¿Por qué se cuelga la expresión regular?

Para entenderlo simplifiquemos el ejemplo: elimine los espacios `\s?`. Entonces se convierte en `^(\\w+)*$`.

Y, para hacer las cosas más obvias sustituymos `\w` por `\d`. La expresión regular resultante sigue colgando, por ejemplo:

```

let regexp = /^(\\d+)*$/;

let str = "012345678901234567890123456789z";

// tardará mucho tiempo (¡cuidado!)
alert(regexp.test(str));

```

¿Qué ocurre con la regexp?

En primer lugar uno puede notar que la regexp `(\\d+)*` es un poco extraña. El cuantificador `*` parece extraño. Si queremos un número podemos utilizar `\\d+`.

Efectivamente la regexp es artificial; la hemos obtenido simplificando el ejemplo anterior. Pero la razón por la que es lenta es la misma. Así que vamos a entenderlo y entonces el ejemplo anterior se hará evidente.

¿Qué sucede durante la búsqueda de `^(\\d+)*$` en la línea `123456789z` (acortada un poco para mayor claridad, por favor tenga en cuenta un carácter no numérico `z` al final, es importante) que tarda tanto?

Esto es lo que hace el motor regexp:

1. En primer lugar el motor regexp intenta encontrar el contenido de los paréntesis: el número `d+`. El `+` es codicioso por defecto, por lo que consume todos los dígitos:

```

\d+.....
(123456789)z

```

Una vez consumidos todos los dígitos se considera que se ha encontrado el `d+` (como `123456789`).

Entonces se aplica el cuantificador de asterisco `(\\d+)*`. Pero no hay más dígitos en el texto, así que el asterisco no da nada.

El siguiente carácter del patrón es el final de la cadena `$`. Pero en el texto tenemos `z` en su lugar, por lo que no hay coincidencia:

```

X
\d+.....$
(123456789)z

```

2. Como no hay ninguna coincidencia, el cuantificador codicioso `+` disminuye el recuento de repeticiones, retrocede un carácter.

Ahora `\d+` toma todos los dígitos excepto el último (`12345678`):

```

\d+.....
(12345678)9z

```

3. Entonces el motor intenta continuar la búsqueda desde la siguiente posición (justo después de `12345678`).

Se puede aplicar el asterisco patrón: `(\d+)*` : da una coincidencia más de patrón: `\d+`, el número 9 :

```
\d+.....\d+
(12345678)(9)z
```

El motor intenta coincidir con `$` de nuevo, pero falla, porque encuentra `z` en su lugar:

```
X
\d+.....\d+
(12345678)(9)z
```

4. No hay coincidencia así que el motor continuará con el retroceso disminuyendo el número de repeticiones. El retroceso generalmente funciona así: el último cuantificador codicioso disminuye el número de repeticiones hasta llegar al mínimo. Entonces el cuantificador codicioso anterior disminuye, y así sucesivamente.

Se intentan todas las combinaciones posibles. Estos son sus ejemplos.

El primer número `\d+` tiene 7 dígitos y luego un número de 2 dígitos:

```
X
\d+.....\d+
(1234567)(89)z
```

El primer número tiene 7 dígitos y luego dos números de 1 dígito cada uno:

```
X
\d+.....\d+\d+
(1234567)(8)(9)z
```

El primer número tiene 6 dígitos y luego un número de 3 dígitos:

```
X
\d+.....\d+
(123456)(789)z
```

El primer número tiene 6 dígitos, y luego 2 números:

```
X
\d+.....\d+ \d+
(123456)(78)(9)z
```

...Y así sucesivamente.

Hay muchas formas de dividir una secuencia de dígitos 123456789 en números. Para ser precisos, hay  $2^n - 1$ , donde `n` es la longitud de la secuencia.

- Para 123456789 tenemos `n=9`, lo que da 511 combinaciones.
- Para una secuencia más larga con “`n=20`” hay alrededor de un millón (1048575) de combinaciones.
- Para `n=30` – mil veces más (1073741823 combinaciones).

Probar cada una de ellas es precisamente la razón por la que la búsqueda lleva tanto tiempo.

## Volver a las palabras y cadenas

Lo mismo ocurre en nuestro primer ejemplo, cuando buscamos palabras por el patrón `^( \w+ \s? ) * $` en la cadena [An input that hangs!](#).

La razón es que una palabra puede representarse como un `\w+` o muchos:

```
(input)
```

```
(inpu)(t)
(inp)(u)(t)
(in)(p)(ut)
...
```

Para un humano es obvio que puede no haber coincidencia porque la cadena termina con un signo de exclamación ! pero la expresión regular espera un carácter denominativo \w o un espacio \s al final. Pero el motor no lo sabe.

El motor prueba todas las combinaciones de cómo la regexp `(\w+\s?)*` puede “consumir” la cadena, incluyendo las variantes con espacios `(\w+\s)*` y sin ellos `(\w+)*` (porque los espacios \s? son opcionales). Como hay muchas combinaciones de este tipo (lo hemos visto con dígitos), la búsqueda lleva muchísimo tiempo.

¿Qué hacer?

¿Debemos activar el lazy mode?

Desgraciadamente eso no ayudará: si sustituimos \w+ por \w+? la regexp seguirá colgada. El orden de las combinaciones cambiará, pero no su número total.

Algunos motores de expresiones regulares hacen análisis complicados y automatizaciones finitas que permiten evitar pasar por todas las combinaciones o hacerlo mucho más rápido, pero la mayoría de los motores no lo hacen. Además, eso no siempre ayuda.

## ¿Cómo solucionarlo?

Hay dos enfoques principales para solucionar el problema.

El primero es reducir el número de combinaciones posibles.

Hagamos que el espacio no sea opcional reescribiendo la expresión regular como `^( \w+\s )*\w*$` buscaremos cualquier número de palabras seguidas de un espacio `(\w+\s)*`, y luego (opcionalmente) una palabra final `\w*`.

Esta regexp es equivalente a la anterior (coincide con lo mismo) y funciona bien:

```
let regexp = /^(\w+\s)*\w*$/
let str = "An input string that takes a long time or even makes this regex hang!";

alert(regexp.test(str)); // false
```

¿Por qué ha desaparecido el problema?

Porque ahora el espacio es obligatorio.

La regexp anterior, si omitimos el espacio, se convierte en `(\w+)*`, dando lugar a muchas combinaciones de \w+ dentro de una misma palabra

Así, `input` podría coincidir con dos repeticiones de \w+ así:

```
\w+ \w+
(inp)(ut)
```

El nuevo patrón es diferente: `(\w+\s)*` especifica repeticiones de palabras seguidas de un espacio. La cadena `input` no puede coincidir con dos repeticiones de `\w+\s`, porque el espacio es obligatorio.

Ahora se ahorra el tiempo necesario para probar un montón de combinaciones (en realidad la mayoría).

## Previendo el backtracking

Sin embargo no siempre es conveniente reescribir una regexp. En el ejemplo anterior era fácil, pero no siempre es obvio cómo hacerlo.

Además una regexp reescrita suele ser más compleja y eso no es bueno. Las regexps son suficientemente complejas sin necesidad de esfuerzos adicionales.

Por suerte hay un enfoque alternativo. Podemos prohibir el retroceso para el cuantificador.

La raíz del problema es que el motor de regexp intenta muchas combinaciones que son obviamente erróneas para un humano.

Por ejemplo, en la regexp `(\d+)*$` es obvio para un humano que `patrón:+` no debería retroceder. Si sustituimos un `patrón:\d+` por dos `\d+\d+` separados nada cambia:

```
\d+.....
(123456789)!
```

```
\d+...\d+...
(1234)(56789)!
```

Y en el ejemplo original `^(w+s?)*$` podemos querer prohibir el backtracking en `\w+`. Es decir: `\w+` debe coincidir con una palabra entera, con la máxima longitud posible. No es necesario reducir el número de repeticiones en `\w+` o dividirlo en dos palabras `\w+\w+` y así sucesivamente.

Los motores de expresiones regulares modernos admiten cuantificadores posesivos para ello. Los cuantificadores regulares se convierten en posesivos si añadimos `+` después de ellos. Es decir, usamos `\d++` en lugar de `\d+` para evitar que `+` retroceda.

Los cuantificadores posesivos son de hecho más simples que los “regulares”. Simplemente coinciden con todos los que pueden sin ningún tipo de retroceso. El proceso de búsqueda sin retroceso es más sencillo.

También existen los llamados “grupos de captura atómicos”, una forma de desactivar el retroceso dentro de los paréntesis.

...Pero la mala noticia es que, por desgracia, en JavaScript no están soportados.

Sin embargo, podemos emularlos utilizando “lookahead transform”.

### Lookahead al rescate!

Así que hemos llegado a temas realmente avanzados. Nos gustaría que un cuantificador como `+` no retrocediera porque a veces retroceder no tiene sentido.

El patrón para tomar tantas repeticiones de `\w` como sea posible sin retroceder es: `(?=(\w+))\1`. Por supuesto, podríamos tomar otro patrón en lugar de `\w`.

Puede parecer extraño, pero en realidad es una transformación muy sencilla.

Vamos a descifrarla:

- Lookahead `?=` busca la palabra más larga `\w+` a partir de la posición actual.
- El contenido de los paréntesis con `?=...` no es memorizado por el motor así que envuelva `\w+` en paréntesis. Entonces el motor memorizará su contenido
- ...y nos permitirá hacer referencia a él en el patrón como `\1`.

Es decir: miramos hacia adelante y si hay una palabra `\w+`, entonces la emparejamos como `\1`.

¿Por qué? Porque el lookahead encuentra una palabra `\w+` como un todo y la capturamos en el patrón con `\1`. Así que esencialmente implementamos un cuantificador posesivo más `+`. Captura sólo la palabra entera `patrón:\w+`, no una parte de ella.

Por ejemplo, en la palabra `JavaScript` no sólo puede coincidir con `Java` sino que deja fuera `Script` para que coincida con el resto del patrón.

He aquí la comparación de dos patrones:

```
alert("JavaScript".match(/\w+Script/)); // JavaScript
alert("JavaScript".match(/(?=(\w+))\1Script/)); // null
```

1. En la primera variante, `\w+` captura primero la palabra completa `JavaScript`, pero luego `+` retrocede carácter por carácter, para intentar coincidir con el resto del patrón, hasta que finalmente tiene éxito (cuando `\w+` coincide con `Java`).
2. En la segunda variante `(?=(\w+))` mira hacia adelante y encuentra la palabra `JavaScript`, que está incluida en el patrón como un todo por `\1`, por lo que no hay manera de encontrar `Script` después de ella.

Podemos poner una expresión regular más compleja en `(?=(\w+))\1` en lugar de `\w`, cuando necesitemos prohibir el retroceso para `+` después de ella.

### Por favor tome nota:

Hay más (en inglés) acerca de la relación entre los cuantificadores posesivos y lookahead en los artículos [Regex: Emulate Atomic Grouping \(and Possessive Quantifiers\) with LookAhead ↗](#) y [Mimicking Atomic Groups ↗](#).

Reescribamos el primer ejemplo utilizando lookahead para evitar el backtracking:

```
let regexp = /^(?=(\w+)\2\s?)*$/;

alert(regexp.test("A good string")); // true

let str = "An input string that takes a long time or even makes this regex hang!";

alert(regexp.test(str)); // false, funciona, ¡y rápido!
```

Aquí se utiliza `\2` en lugar de `\1` porque hay paréntesis exteriores adicionales. Para evitar enredarnos con los números, podríamos dar a los paréntesis un nombre, por ejemplo `(?<word>\w+)`.

```
// nombramos a los parentesis ?<word>, y los referenciamos como \k<word>
let regexp = /^(?=(?<word>\w+)\k<word>\s?)*$/;

let str = "An input string that takes a long time or even makes this regex hang!";

alert(regexp.test(str)); // false

alert(regexp.test("A correct string")); // true
```

El problema descrito en este artículo se llama “backtracking catastrófico”.

Cubrimos dos formas de resolverlo:

- Reescribir la regexp para reducir el número de combinaciones posibles.
- Evitar el retroceso.

## Indicador adhesivo “y”, buscando en una posición.

EL indicador `y` permite realizar la búsqueda en una posición dada en el string de origen.

Para entender el caso de uso del indicador `y` exploremos un ejemplo práctico.

Una tarea común para regexps es el “Análisis léxico”: tomar un texto (como el de un lenguaje de programación), y analizar sus elementos estructurales. Por ejemplo, HTML tiene etiquetas y atributos, el código JavaScript tiene funciones, variables, etc.

Escribir analizadores léxicos es un área especial, con sus propias herramientas y algoritmos, así que no profundizaremos en ello; pero existe una tarea común: leer algo en una posición dada.

Por ej. tenemos una cadena de código `let varName = "value"`, y necesitamos leer el nombre de su variable, que comienza en la posición 4 .

Buscaremos el nombre de la variable usando regexp `\w+`. En realidad, el nombre de la variable de JavaScript necesita un regexp un poco más complejo para un emparejamiento más preciso, pero aquí eso no importa.

Una llamada a `str.match(/\w+/)` solo encontrará la primera palabra de la línea (`let`). No es la que queremos. Podríamos añadir el indicador `g`, pero al llamar a `str.match(/\w+/g)` buscará todas las palabras del texto y solo necesitamos una y en la posición 4 . De nuevo, no es lo que necesitamos.

### Entonces, ¿cómo buscamos exactamente en un posición determinada?

Usemos el método `regexp.exec(str)`.

Para un `regexp` sin los indicadores `g` y `y`, este método busca la primera coincidencia y funciona exactamente igual a `str.match(regexp)`.

...Pero si existe el indicador `g`, realiza la búsqueda en `str` empezando desde la posición almacenada en su propiedad `regexp.lastIndex` . Y si encuentra una coincidencia, establece `regexp.lastIndex` en el index inmediatamente posterior a la coincidencia.

En otras palabras, `regexp.lastIndex` funciona como punto de partida para la búsqueda, cada llamada lo reestablece a un nuevo valor: el posterior a la última coincidencia.

Entonces, llamadas sucesivas a `regexp.exec(str)` devuelven coincidencias una después de la otra.

Un ejemplo (con el indicador `g`):

```
let str = 'let varName'; // encontramos todas las palabras del string
let regexp = /\w+/g;

alert(regexp.lastIndex); // 0 (inicialmente lastIndex=0)

let word1 = regexp.exec(str);
alert(word1[0]); // let (primera palabra)
alert(regexp.lastIndex); // 3 (Posición posterior a la coincidencia)

let word2 = regexp.exec(str);
alert(word2[0]); // varName (2da palabra)
alert(regexp.lastIndex); // 11 (Posición posterior a la coincidencia)

let word3 = regexp.exec(str);
alert(word3); // null (no más coincidencias)
alert(regexp.lastIndex); // 0 (se reinicia al final de la búsqueda)
```

Podemos conseguir todas las coincidencias en el loop:

```
let str = 'let varName';
let regexp = /\w+/g;

let result;

while (result = regexp.exec(str)) {
 alert(`Found ${result[0]} at position ${result.index}`);
 // Found let at position 0, then
 // Found varName at position 4
}
```

Tal uso de `regexp.exec` es una alternativa al método `str.match bAll`, con más control sobre el proceso.

Volvamos a nuestra tarea.

Podemos establecer manualmente `lastIndex` a `4`, para comenzar la búsqueda desde la posición dada.

Como aquí:

```
let str = 'let varName = "value"';

let regexp = /\w+/g; // Sin el indicador "g", la propiedad lastindex es ignorada.

regexp.lastIndex = 4;

let word = regexp.exec(str);
alert(word); // varName
```

¡Problema resuelto!

Realizamos una búsqueda de `\w+`, comenzando desde la posición `regexp.lastIndex = 4`.

El resultado es correcto.

...Pero espera, no tan rápido.

Nota que la búsqueda comienza en la posición `lastIndex` y luego sigue adelante. Si no hay ninguna palabra en la posición `lastIndex` pero la hay en algún lugar posterior, entonces será encontrada:

```
let str = 'let varName = "value"';

let regexp = /\w+/g;
```

```
// comenzando desde la posición 3
regexp.lastIndex = 3;

let word = regexp.exec(str);
// encuentra coincidencia en la posición 4
alert(word[0]); // varName
alert(word.index); // 4
```

Para algunas tareas, incluido el análisis léxico, esto está mal. Necesitamos la coincidencia en la posición exacta, y para ello es el flag `y`.

**El indicador `y` hace que `regexp.exec` busque “exactamente en” la posición `lastIndex`, no “comenzando en” ella.**

Aquí está la misma búsqueda con el indicador `y`:

```
let str = 'let varName = "value"';

let regexp = /\w+/y;

regexp.lastIndex = 3;
alert(regexp.exec(str)); // null (Hay un espacio en la posición 3, no una palabra)

regexp.lastIndex = 4;
alert(regexp.exec(str)); // varName (Una palabra en la posición 4)
```

Como podemos ver, el `/\w+/y` de `regexp` no coincide en la posición `3` (a diferencia del indicador `g`), pero coincide en la posición `4`.

No solamente es lo que necesitamos, el uso del indicador `y` mejora el rendimiento.

Imagina que tenemos un texto largo, y no hay coincidencias en él. Entonces la búsqueda con el indicador `g` irá hasta el final del texto, y esto tomará significativamente más tiempo que la búsqueda con el indicador `y`.

En tareas tales como el análisis léxico, normalmente hay muchas búsquedas en una posición exacta. Usar el indicador `y` es la clave para un buen desempeño.

## Métodos de RegExp y String

En este artículo vamos a abordar varios métodos que funcionan con expresiones regulares a fondo.

### `str.match(regexp)`

El método `str.match(regexp)` encuentra coincidencias para las expresiones regulares (`regexp`) en la cadena (`str`).

Tiene 3 modos:

- Si la expresión regular (`regexp`) no tiene la bandera `g`, retorna un array con los grupos capturados y las propiedades `index` (posición de la coincidencia), `input` (cadena de entrada, igual a `str`):

```
let str = "I love JavaScript";

let result = str.match(/Java(Script)/);

alert(result[0]); // JavaScript (toda la coincidencia)
alert(result[1]); // Script (primer grupo capturado)
alert(result.length); // 2

// Additional information:
alert(result.index); // 7 (match position)
alert(result.input); // I love JavaScript (cadena de entrada)
```

- Si la expresión regular (`regexp`) tiene la bandera `g`, retorna un array de todas las coincidencias como cadenas, sin capturar grupos y otros detalles.

```
let str = "I love JavaScript";

let result = str.match(/Java(Script)/g);
```

```
alert(result[0]); // JavaScript
alert(result.length); // 1
```

3. Si no hay coincidencias, no importa si tiene la bandera `g` o no, retorna `null`.

Esto es algo muy importante. Si no hay coincidencias, no vamos a obtener un array vacío, pero sí un `null`. Es fácil cometer un error olvidándolo, ej.:

```
let str = "I love JavaScript";
let result = str.match(/HTML/);
alert(result); // null
alert(result.length); // Error: Cannot read property 'length' of null
```

Si queremos que el resultado sea un array, podemos escribirlo así:

```
let result = str.match(regexp) || [];
```

## str.matchAll(regexp)



### Una adición reciente

Esta es una adición reciente al lenguaje. Los navegadores antiguos pueden necesitar polyfills.

El método `str.matchAll(regexp)` es una variante ("nueva y mejorada") de `str.match`.

Es usado principalmente para buscar por todas las coincidencias con todos los grupos.

Hay 3 diferencias con `match`:

1. Retorna un objeto iterable con las coincidencias en lugar de un array. Podemos convertirlo en un array usando el método `Array.from`.
2. Cada coincidencia es retornada como un array con los grupos capturados (el mismo formato de `str.match` sin la bandera `g`).
3. Si no hay resultados devuelve un objeto iterable vacío en lugar de `null`.

Ejemplo de uso:

```
let str = '<h1>Hello, world!</h1>';
let regexp = /<(.+?)>/g;

let matchAll = str.matchAll(regexp);

alert(matchAll); // [object RegExp String Iterator], no es un array, pero sí un objeto iterable

matchAll = Array.from(matchAll); // ahora es un array

let firstMatch = matchAll[0];
alert(firstMatch[0]); // <h1>
alert(firstMatch[1]); // h1
alert(firstMatch.index); // 0
alert(firstMatch.input); // <h1>Hello, world!</h1>
```

Si usamos `for..of` para iterar todas las coincidencias de `matchAll`, no necesitamos `Array.from`.

## str.split(regexp|substr, limit)

Divide la cadena usando la expresión regular (o una sub-cadena) como delimitador.

Podemos usar `split` con cadenas, así:

```
alert('12-34-56'.split('-')) // array de ['12', '34', '56']
```

O también dividir una cadena usando una expresión regular de la misma forma:

```
alert('12, 34, 56'.split(/\s*/)) // array de ['12', '34', '56']
```

## str.search(regexp)

El método `str.search(regexp)` retorna la posición de la primera coincidencia o `-1` si no encuentra nada:

```
let str = "A drop of ink may make a million think";
alert(str.search(/ink/i)); // 10 (posición de la primera coincidencia)
```

**Limitación importante:** `search` solamente encuentra la primera coincidencia.

Si necesitamos las posiciones de las demás coincidencias, deberíamos usar otros medios, como encontrar todos con `str.matchAll(regexp)`.

## str.replace(str|regexp, str|func)

Este es un método genérico para buscar y reemplazar, uno de los más útiles. La navaja suiza para buscar y reemplazar.

Podemos usarlo sin expresiones regulares, para buscar y reemplazar una sub-cadena:

```
// reemplazar guion por dos puntos
alert('12-34-56'.replace("-", ":")) // 12:34:56
```

Sin embargo hay una trampa:

**Cuando el primer argumento de `replace` es una cadena, solo reemplaza la primera coincidencia.**

Puedes ver eso en el ejemplo anterior: solo el primer `" - "` es reemplazado por `" : "`.

Para encontrar todos los guiones, no necesitamos usar un cadena `" - "` sino una expresión regular `/ - /g` con la bandera `g` obligatoria:

```
// reemplazar todos los guiones por dos puntos
alert('12-34-56'.replace(/-/g, ":")) // 12:34:56
```

El segundo argumento es la cadena de reemplazo. Podemos usar caracteres especiales:

Símbolos	Acción en la cadena de reemplazo
<code>\$&amp;</code>	inserta toda la coincidencia
<code>\$`</code>	inserta una parte de la cadena antes de la coincidencia
<code>\$'</code>	inserta una parte de la cadena después de la coincidencia
<code>\$n</code>	si <code>n</code> es un número, inserta el contenido del enésimo grupo capturado, para más detalles ver <a href="#">Grupos de captura</a>
<code>\$&lt;nombre&gt;</code>	inserta el contenido de los paréntesis con el <code>nombre</code> dado, para más detalles ver <a href="#">Grupos de captura</a>
<code>\$\$</code>	inserta el carácter <code>\$</code>

Por ejemplo:

```
let str = "John Smith";
// intercambiar el nombre con el apellido
alert(str.replace(/(john) (smith)/i, '$2, $1')) // Smith, John
```

**Para situaciones que requieran reemplazos “inteligentes”, el segundo argumento puede ser una función.**

Puede ser llamado por cada coincidencia y el valor retornado puede ser insertado como un reemplazo.

La función es llamada con los siguientes argumentos `func(match, p1, p2, ..., pn, offset, input, groups)`:

1. `match` – la coincidencia,
2. `p1, p2, ..., pn` – contenido de los grupos capturados (si hay alguno),
3. `offset` – posición de la coincidencia,
4. `input` – la cadena de entrada,
5. `groups` – un objeto con los grupos nombrados.

Si hay paréntesis en la expresión regular, entonces solo son 3 argumentos: `func(str, offset, input)`.

Por ejemplo, hacer mayúsculas todas las coincidencias:

```
let str = "html and css";

let result = str.replace(/html|css/gi, str => str.toUpperCase());

alert(result); // HTML and CSS
```

Reemplazar cada coincidencia por su posición en la cadena:

```
alert("Ho-Ho-ho".replace(/ho/gi, (match, offset) => offset)); // 0-3-6
```

En el ejemplo anterior hay dos paréntesis, entonces la función de reemplazo es llamada con 5 argumentos: el primero es toda la coincidencia, luego dos paréntesis, y después (no usado en el ejemplo) la posición de la coincidencia y la cadena de entrada:

```
let str = "John Smith";

let result = str.replace(/(\w+) (\w+)/, (match, name, surname) => `${surname}, ${name}`);

alert(result); // Smith, John
```

Si hay muchos grupos, es conveniente usar parámetros rest para acceder a ellos:

```
let str = "John Smith";

let result = str.replace(/(\w+) (\w+)/, (...match) => `${match[2]}, ${match[1]}`);

alert(result); // Smith, John
```

O, si estamos usando grupos nombrados, entonces el objeto `groups` con ellos es siempre el último, por lo que podemos obtenerlos así:

```
let str = "John Smith";

let result = str.replace(/(<name>\w+) (<surname>\w+)/, (...match) => {
 let groups = match.pop();

 return `${groups.surname}, ${groups.name}`;
});

alert(result); // Smith, John
```

Usando una función nos da todo el poder del reemplazo, porque obtiene toda la información de la coincidencia, ya que tiene acceso a las variables externas y se puede hacer de todo.

## **str.replaceAll(str|regexp, str|func)**

Este método es esencialmente el mismo que `str.replace`, con dos diferencias principales:

- Si el primer argumento es un string, reemplaza *todas las ocurrencias* del string, mientras que `replace` solamente reemplaza la *primera ocurrencia*.
- Si el primer argumento es una expresión regular sin la bandera `g`, habrá un error. Con la bandera `g`, funciona igual que `replace`.

El caso de uso principal para `replaceAll` es el reemplazo de todas las ocurrencias de un string.

Como esto:

```
// reemplaza todos los guiones por dos puntos
alert('12-34-56'.replaceAll("-", ":")) // 12:34:56
```

### `regexp.exec(str)`

El método `regexp.exec(str)` retorna una coincidencia por expresión regular `regexp` en la cadena `str`. A diferencia de los métodos anteriores, se llama en una expresión regular en lugar de en una cadena.

Se comporta de manera diferente dependiendo de si la expresión regular tiene la bandera `g` o no.

Si no está la bandera `g`, entonces `regexp.exec(str)` retorna la primera coincidencia igual que `str.match(regexp)`. Este comportamiento no trae nada nuevo.

Pero si está la bandera `g`, entonces:

- Una llamada a `regexp.exec(str)` retorna la primera coincidencia y guarda la posición inmediatamente después en `regexp.lastIndex`.
- La siguiente llamada de la búsqueda comienza desde la posición de `regexp.lastIndex`, retorna la siguiente coincidencia y guarda la posición inmediatamente después en `regexp.lastIndex`.
- ...y así sucesivamente.
- Si no hay coincidencias, `regexp.exec` retorna `null` y resetea `regexp.lastIndex` a `0`.

Entonces, repetidas llamadas retornan todas las coincidencias una tras otra, usando la propiedad `regexp.lastIndex` para realizar el rastreo de la posición actual de la búsqueda.

En el pasado, antes de que el método `str.matchAll` fuera agregado a JavaScript, se utilizaban llamadas de `regexp.exec` en el ciclo para obtener todas las coincidencias con sus grupos:

```
let str = 'More about JavaScript at https://javascript.info';
let regexp = /javascript/ig;

let result;

while (result = regexp.exec(str)) {
 alert(`Se encontró ${result[0]} en la posición ${result.index}`);
 // Se encontró JavaScript en la posición 11, luego
 // Se encontró javascript en la posición 33
}
```

Esto también funciona, aunque para navegadores modernos `str.matchAll` usualmente es lo más conveniente.

**Podemos usar `regexp.exec` para buscar desde una posición dada configurando manualmente el `lastIndex`.**

Por ejemplo:

```
let str = 'Hello, world!';

let regexp = /\w+/g; // sin la bandera "g", la propiedad `lastIndex` es ignorada
regexp.lastIndex = 5; // buscar desde la 5ta posición (desde la coma)

alert(regexp.exec(str)); // world
```

Si la expresión regular tiene la bandera `y`, entonces la búsqueda se realizará exactamente en la posición del `regexp.lastIndex`, no más adelante.

Vamos a reemplazar la bandera `g` con `y` en el ejemplo anterior. No habrá coincidencias, ya que no hay palabra en la posición 5 :

```

let str = 'Hello, world!';
let regexp = /\w+/y;
regexp.lastIndex = 5; // buscar exactamente en la posición 5
alert(regexp.exec(str)); // null

```

Esto es conveniente cuando con una expresión regular necesitamos “leer” algo de la cadena en una posición exacta, no en otro lugar.

### regexp.test(str)

El método `regexp.test(str)` busca por una coincidencia y retorna `true/false` si existe.

Por ejemplo:

```

let str = "I love JavaScript";
// estas dos pruebas hacen lo mismo
alert(/love/i.test(str)); // true
alert(str.search(/love/i) != -1); // true

```

Un ejemplo con respuesta negativa:

```

let str = "Bla-bla-bla";
alert(/love/i.test(str)); // false
alert(str.search(/love/i) != -1); // false

```

Si la expresión regular tiene la bandera `g`, el método `regexp.test` busca la propiedad `regexp.lastIndex` y la actualiza, igual que `regexp.exec`.

Entonces podemos usarlo para buscar desde un posición dada:

```

let regexp = /love/gi;
let str = "I love JavaScript";
// comienza la búsqueda desde la posición 10:
regexp.lastIndex = 10;
alert(regexp.test(str)); // false (sin coincidencia)

```

#### ⚠️ La misma expresión regular probada (de manera global) repetidamente en diferentes lugares puede fallar

Si nosotros aplicamos la misma expresión regular (de manera global) a diferentes entradas, puede causar resultados incorrectos, porque `regexp.test` anticipa las llamadas usando la propiedad `regexp.lastIndex`, por lo que la búsqueda en otra cadena puede comenzar desde una posición distinta a cero.

Por ejemplo, aquí llamamos `regexp.test` dos veces en el mismo texto y en la segunda vez falla:

```

let regexp = /javascript/g; // (expresión regular creada: regexp.lastIndex=0)
alert(regexp.test("javascript")); // true (ahora regexp.lastIndex es 10)
alert(regexp.test("javascript")); // false

```

Eso es porque `regexp.lastIndex` no es cero en la segunda prueba.

Para solucionarlo, podemos establecer `regexp.lastIndex = 0` antes de cada búsqueda. O en lugar de llamar a los métodos en la expresión regular usar los métodos de cadena `str.match/search/...`, ellos no usan el `lastIndex`.

## Soluciones

## ArrayBuffer, arrays binarios

---

### Concatenar arrays tipados

```
function concat(arrays) {
 // suma de las longitudes de array individuales
 let totalLength = arrays.reduce((acc, value) => acc + value.length, 0);

 let result = new Uint8Array(totalLength);

 if (!arrays.length) return result;

 // para cada array: copiarlo sobre "result"
 // el siguiente array es copiado inmediatamente después del anterior
 let length = 0;
 for(let array of arrays) {
 result.set(array, length);
 length += array.length;
 }

 return result;
}
```

Abrir la solución con pruebas en un entorno controlado. ↗

A formulación

## Fetch

---

### Fetch de usuarios de GitHub

Para obtener un usuario tenemos que ejecutar el siguiente código:

```
fetch('https://api.github.com/users/USERNAME')
```

Si la respuesta contiene el status `200`, utilizamos el método `.json()` para leer el objeto JS.

Por el contrario, si el `fetch` falla o la respuesta no contiene un status 200, devolvemos `null` en el resultado del arreglo.

Código:

```
async function getUsers(names) {
 let jobs = [];

 for(let name of names) {
 let job = fetch(`https://api.github.com/users/${name}`).then(
 successResponse => {
 if (successResponse.status != 200) {
 return null;
 } else {
 return successResponse.json();
 }
 },
 failResponse => {
 return null;
 }
);
 jobs.push(job);
 }

 let results = await Promise.all(jobs);

 return results;
}
```

Nota: la función `.then` está directamente vinculada al `fetch`. Por lo tanto, cuando se obtiene la respuesta se procede a ejecutar la función `.json()` inmediatamente en lugar de esperar a las otras peticiones.

Si en su lugar utilizáramos `await Promise.all(names.map(name => fetch(...)))` y llamamos a la función `.json()` sobre los resultados, entonces esperaríamos a que todas las peticiones `fetch` completen antes de obtener una respuesta. Al agregar `.json()` directamente en cada `fetch`, nos aseguramos de que las peticiones se procesen de manera independiente obteniendo una mejor respuesta en nuestra aplicación.

Esto es un ejemplo de cómo la API de Promesas puede ser útil aunque mayormente se utilice `async/await`.

[Abrir la solución con pruebas en un entorno controlado.](#) ↗

[A formulación](#)

## Fetch: Cross-Origin Requests

### ¿Por qué necesitamos el origen (Origin)?

Necesitamos la cabecera `Origin`, ya que en algunos casos `Referer` no está presente. Por ejemplo, cuando realizamos un `fetch` a una página HTTP desde una HTTPS (acceder a un sitio menos seguro desde uno más seguro), en ese caso no tendremos el campo `Referer`.

La [Política de seguridad de contenido](#) ↗ puede prohibir el envío de `Referer`.

Como veremos, `fetch` tiene opciones con las que es posible evitar el envío de `Referer` e incluso permite su modificación (dentro del mismo sitio).

Por especificación, `Referer` es una cabecera HTTP opcional.

Por el hecho de que `Referer` no es confiable, la cabecera `Origin` ha sido creada. El navegador garantiza el envío correcto de `Origin` para las solicitudes de origen cruzado.

[A formulación](#)

## LocalStorage, sessionStorage

### Guardar automáticamente un campo de formulario

[Abrir la solución en un entorno controlado.](#) ↗

[A formulación](#)

## Animaciones CSS

### Animar un avión (CSS)

CSS para animar tanto `width` como `height`:

```
/* clase original */

#flyjet {
 transition: all 3s;
}

/* JS añade .growing */
#flyjet.growing {
 width: 400px;
```

```
height: 240px;
}
```

Ten en cuenta que `transitionend` se dispara dos veces, una para cada propiedad. Entonces, si no realizamos una verificación adicional, el mensaje aparecería 2 veces.

[Abrir la solución en un entorno controlado.](#)

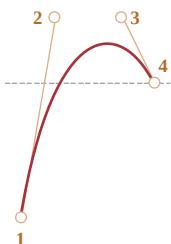
A formulación

## Animar el avión volando (CSS)

Necesitamos elegir la curva de Bézier correcta para esa animación. Debe tener `y>1` en algún punto para que el avión “salte”.

Por ejemplo, podemos tomar ambos puntos de control con `y>1`, como: `cubic-bezier(0.25, 1.5, 0.75, 1.5)`.

La gráfica:



[Abrir la solución en un entorno controlado.](#)

A formulación

## Círculo animado

[Abrir la solución en un entorno controlado.](#)

A formulación

## Círculo animado con función de callback

[Abrir la solución en un entorno controlado.](#)

A formulación

## Animaciones JavaScript

### Animar la pelota que rebota

Para rebotar podemos usar la propiedad CSS `top` y `position: absolute` para la pelota dentro del campo con `position: relative`.

La coordenada inferior del campo es `field.clientHeight`. La propiedad CSS `top` se refiere al borde superior de la bola. Por lo tanto, debe ir desde `0` hasta `field.clientHeight - ball.clientHeight`, que es la posición final más baja del borde superior de la pelota.

Para obtener el efecto de “rebote”, podemos usar la función de sincronización `bounce` en el modo `easeOut`.

Aquí está el código final de la animación:

```

let to = field.clientHeight - ball.clientHeight;

animate({
 duration: 2000,
 timing: makeEaseOut(bounce),
 draw(progress) {
 ball.style.top = to * progress + 'px'
 }
});

```

[Abrir la solución en un entorno controlado.](#)

A formulación

## Animar la pelota rebotando hacia la derecha

En la tarea [Animar la pelota que rebota](#) solo teníamos una propiedad para animar. Ahora necesitamos una más: `elem.style.left`.

La coordenada horizontal cambia por otra ley: no “rebota”, sino que aumenta gradualmente desplazando la pelota hacia la derecha.

Podemos escribir una `animate` más para ello.

Como función de tiempo podríamos usar `linear`, pero algo como `makeEaseOut(quad)` se ve mucho mejor.

El código:

```

let height = field.clientHeight - ball.clientHeight;
let width = 100;

// animate top (rebotando)
animate({
 duration: 2000,
 timing: makeEaseOut(bounce),
 draw: function(progress) {
 ball.style.top = height * progress + 'px'
 }
});

// animate left (moviéndose a la derecha)
animate({
 duration: 2000,
 timing: makeEaseOut(quad),
 draw: function(progress) {
 ball.style.left = width * progress + "px"
 }
});

```

[Abrir la solución en un entorno controlado.](#)

A formulación

## Elementos personalizados

### Elemento reloj dinámico

Por favor ten en cuenta:

1. Borramos el temporizador `setInterval` cuando el elemento es quitado del documento. Esto es importante, de otro modo continuará ejecutando aunque no se lo necesite más, y el navegador no puede liberar la memoria asignada a este elemento.
2. Podemos acceder a la fecha actual con la propiedad `elem.date`. Todos los métodos y propiedades de clase son naturalmente métodos y propiedades del elemento.

[Abrir la solución en un entorno controlado.](#)

A formulación

## Anclas: inicio ^ y final \$ de cadena

### Regexp ^\$

Una cadena vacía es la única coincidencia: comienza y termina inmediatamente.

Esta tarea demuestra una vez más que los anclajes no son caracteres, sino pruebas.

La cadena está vacía `" "`. El motor primero coincide con `^` (inicio de entrada), sí, está allí, y luego inmediatamente el final `$`, también está. Entonces hay una coincidencia.

A formulación

## Límite de palabra: \b

### Encuentra la hora

La respuesta: `\b\d\d:\d\d\b`.

```
alert("Desayuno a las 09:00 en la habitación 123:456.".match(/\b\d\d:\d\d\b/)); // 09:00
```

A formulación

## Conjuntos y rangos [...]

### Java[^script]

Respuestas: **no, si**.

- En el script `Java` no coincide con nada, porque `[^script]` significa “cualquier carácter excepto los dados”. Entonces, la expresión regular busca `"Java"` seguido de uno de esos símbolos, pero hay un final de cadena, sin símbolos posteriores.

```
alert("Java".match(/Java[^script]/)); // null
```

- Sí, porque la sección `[^script]` en parte coincide con el carácter `"S"`. No está en `script`. Como el regexp distingue entre mayúsculas y minúsculas (sin flag `i`), procesa a `"S"` como un carácter diferente de `"s"`.

```
alert("JavaScript".match(/Java[^script]/)); // "Javas"
```

A formulación

## Encuentra la hora como hh:mm o hh-mm

Respuesta: `\d\d[-:]\d\d`.

```
let regexp = /\d\d[-:]\d\d/g;
alert("El desayuno es a las 09:00. La cena es a las 21-30".match(regexp)); // 09:00, 21-30
```

Tenga en cuenta que el guión `' - '` tiene un significado especial entre corchetes, pero solo entre otros caracteres, no al principio o al final, por lo que no necesitamos escaparlo.

A formulación

## Cuantificadores +, \*, ? y {n}

### ¿Cómo encontrar puntos suspensivos "...?"?

Solución:

```
let regexp = /\.{3,}/g;
alert("Hola!... ¿Cómo vas?.....".match(regexp)); // ...,
```

Tenga en cuenta que el punto es un carácter especial, por lo que debemos escaparlo e insertarlo como `\.`.

A formulación

## Regexp para colores HTML

Necesitamos buscar `#` seguido de 6 caracteres hexadecimales.

Un carácter hexadecimal se puede describir como `[0-9a-fA-F]`. O si usamos la bandera `i`, entonces simplemente `[0-9a-f]`.

Entonces podemos buscar 6 de ellos usando el cuantificador `{6}`.

Como resultado, tenemos la regexp: `/#[a-f0-9]{6}/gi`.

```
let regexp = /#[a-f0-9]{6}/gi;

let str = "color:#121212; background-color:#AA00ef bad-colors:f#fddee #fd2"

alert(str.match(regexp)); // #121212,#AA00ef
```

El problema es que también encuentra el color en secuencias más largas:

```
alert("#12345678".match(/#[a-f0-9]{6}/gi)) // #123456
```

Para corregir eso, agregamos `\b` al final:

```
// color
alert("#123456".match(/#[a-f0-9]{6}\b/gi)); // #123456

// sin color
alert("#12345678".match(/#[a-f0-9]{6}\b/gi)); // null
```

A formulación

## Cuantificadores codiciosos y perezosos

### Una coincidencia para `/d+? d+?/`

El resultado es: `123 4`.

Primero el perezoso `\d+?` trata de tomar la menor cantidad de dígitos posible, pero tiene que llegar al espacio, por lo que toma `123`.

Después el segundo `\d+?` toma solo un dígito, porque es suficiente.

A formulación

## Encuentra el comentario HTML

Necesitamos encontrar el inicio del comentario `<!--`, después todo hasta el fin de `-->`.

Una variante aceptable es `<!--.*?-->` – el cuantificador perezoso detiene el punto justo antes de `-->`. También necesitamos agregar la bandera `s` al punto para incluir líneas nuevas.

De lo contrario, no se encontrarán comentarios multilínea:

```
let regexp = //gs;

let str = `... <!-- Mi -- comentario
prueba --> .. <!--> ..
`;

alert(str.match(regexp)); // '<!-- Mi -- comentario \n prueba -->', '<!-->'
```

A formulación

## Encontrar las etiquetas HTML

La solución es `<[^<>]+>`.

```
let regexp = /<[^<>]+>/g;

let str = '<> <input type="radio" checked> ';

alert(str.match(regexp)); // '', '<input type="radio" checked>', ''
```

A formulación

## Grupos de captura

### Verificar dirección MAC

Un número hexadecimal de dos dígitos es `[0-9a-f]{2}` (suponiendo que se ha establecido el indicador `i`).

Necesitamos ese número `NN`, y luego `:NN` repetido 5 veces (más números);

La expresión regular es: `[0-9a-f]{2}(:[0-9a-f]{2}){5}`

Ahora demostremos que la coincidencia debe capturar todo el texto: comience por el principio y termine por el final. Eso se hace envolviendo el patrón en `^...$`.

Finalmente:

```
let regexp = /^[0-9a-f]{2}(:[0-9a-f]{2}){5}$/i;

alert(regexp.test('01:32:54:67:89:AB')); // true

alert(regexp.test('0132546789AB')); // false (sin dos puntos)

alert(regexp.test('01:32:54:67:89')); // false (5 números, necesita 6)

alert(regexp.test('01:32:54:67:89:ZZ')); // false (ZZ al final)
```

A formulación

## Encuentra el color en el formato #abc o #abcdef

Una expresión regular para buscar colores de 3 dígitos `#abc`: `/#[a-f0-9]{3}/i`.

Podemos agregar exactamente 3 dígitos hexadecimales opcionales más. No necesitamos más ni menos. El color tiene 3 o 6 dígitos.

Utilicemos el cuantificador `{1,2}` para esto: llegaremos a `/#[a-f0-9]{3}{1,2}/i`.

Aquí el patrón `[a-f0-9]{3}` está rodeado en paréntesis para aplicar el cuantificador `{1,2}`.

En acción:

```
let regexp = /#[[a-f0-9]{3}}{1,2}/gi;
let str = "color: #3f3; background-color: #AA00ef; and: #abcd";
alert(str.match(regexp)); // #3f3 #AA00ef #abc
```

Hay un pequeño problema aquí: el patrón encontrado `#abc` en `#abcd`. Para prevenir esto podemos agregar `\b` al final:

```
let regexp = /#[[a-f0-9]{3}}{1,2}\b/gi;
let str = "color: #3f3; background-color: #AA00ef; and: #abcd";
alert(str.match(regexp)); // #3f3 #AA00ef
```

## A formulación

### Encuentre todos los números

Un número positivo con una parte decimal opcional es: `\d+(\.\d+)?`.

Agreguemos el opcional al comienzo `-`:

```
let regexp = /-?\d+(\.\d+)?/g;
let str = "-1.5 0 2 -123.4";
alert(str.match(regexp)); // -1.5, 0, 2, -123.4
```

## A formulación

### Analizar una expresión:

Una expresión regular para un número es: `-?\d+(\.\d+)?`. La creamos en tareas anteriores.

Un operador es `[-+*/]`. El guión `-` va primero dentro de los corchetes porque colocado en el medio significaría un rango de caracteres, cuando nosotros queremos solamente un carácter `-`.

La barra inclinada `/` debe ser escapada dentro de una expresión regular de JavaScript `/ . . . /`, eso lo haremos más tarde.

Necesitamos un número, un operador y luego otro número. Y espaciosopcionales entre ellos.

La expresión regular completa: `-?\d+(\.\d+)?\s*[-+*/]\s*-?\d+(\.\d+)?`.

Tiene 3 partes, con `\s*` en medio de ellas:

1. `-?\d+(\.\d+)?` – el primer número,
2. `[-+*/]` – el operador,

3. `-?\d+(\.\d+)?` – el segundo número.

Para hacer que cada una de estas partes sea un elemento separado del array de resultados, encerrémoslas entre paréntesis: `( -?\d+(\.\d+)? )\s*([-+*/])\s*(-?\d+(\.\d+)? )`.

En acción:

```
let regexp = /(-?\d+(\.\d+)?)\s*([-+*/])\s*(-?\d+(\.\d+)?)/;

alert("1.2 + 12".match(regexp));
```

El resultado incluye:

- `result[0] == "1.2 + 12"` (coincidencia completa)
- `result[1] == "1.2"` (primer grupo `( -?\d+(\.\d+)? )` – el primer número, incluyendo la parte decimal)
- `result[2] == ".2"` (segundo grupo `(\.\d+)?` – la primera parte decimal)
- `result[3] == "+"` (tercer grupo `( [-+*/] )` – el operador)
- `result[4] == "12"` (cuarto grupo `( -?\d+(\.\d+)? )` – el segundo número)
- `result[5] == undefined` (quinto grupo `(\.\d+)?` – la última parte decimal no está presente, por lo tanto es indefinida)

Solo queremos los números y el operador, sin la coincidencia completa o las partes decimales, así que “limpiemos” un poco el resultado.

La coincidencia completa (el primer elemento del array) se puede eliminar cambiando el array `result.shift()`.

Los grupos que contengan partes decimales (número 2 y 4) `(\.\d+)` pueden ser excluidos al agregar `?:` al comienzo: `(?:\.\d+)?`.

La solución final:

```
function parse(expr) {
 let regexp = /(-?\d+(?:\.\d+)?)\s*([-+*/])\s*(-?\d+(?:\.\d+)?)/;

 let result = expr.match(regexp);

 if (!result) return [];
 result.shift();

 return result;
}

alert(parse("-1.23 * 3.45")); // -1.23, *, 3.45
```

Como alternativa al uso de la exclusión de captura `?:`, podemos dar nombre a los grupos:

```
function parse(expr) {
 let regexp = /(<a>-?\d+(?:\.\d+)?)\s*(<operator>[-+*/])\s*(?-?\d+(?:\.\d+)?)/;

 let result = expr.match(regexp);

 return [result.groups.a, result.groups.operator, result.groups.b];
}

alert(parse("-1.23 * 3.45")); // -1.23, *, 3.45;
```

A formulación

## Alternancia (O) |

Encuentra lenguajes de programación

La primera idea puede ser listar los idiomas con `|` en el medio.

Pero eso no funciona bien:

```
let regexp = /Java|JavaScript|PHP|C|C\+\+/g;

let str = "Java, JavaScript, PHP, C, C++";

alert(str.match(regexp)); // Java, Java, PHP, C, C
```

El motor de expresiones regulares busca las alternancias una por una. Es decir: primero verifica si tenemos `Java`, de lo contrario – busca `JavaScript` y así sucesivamente.

Como resultado, nunca se puede encontrar `JavaScript`, simplemente porque encuentra primero `Java`.

Lo mismo con `C` y `C++`.

Hay dos soluciones para ese problema:

1. Cambiar el orden para comprobar primero la coincidencia más larga: `JavaScript | Java | C\+\+ | C | PHP`.
2. Fusionar variantes con el mismo inicio: `Java( Script )? | C( \+\+ )? | PHP`.

En acción:

```
let regexp = /Java(Script)? | C(\+\+)? | PHP/g;

let str = "Java, JavaScript, PHP, C, C++";

alert(str.match(regexp)); // Java, JavaScript, PHP, C, C++
```

## A formulación

### Encuentra la pareja bbtag

La etiqueta de apertura es `\[( b | url | quote )]`.

Luego, para encontrar todo hasta la etiqueta de cierre, usemos el patrón `. *?` con la bandera `S` para que coincida con cualquier carácter, incluida la nueva línea, y luego agreguemos una referencia inversa a la etiqueta de cierre.

El patrón completo: `\[( ( b | url | quote ) )\] . *? \[/\1]`.

En acción:

```
let regexp = /\[((b|url|quote)\)\] . *? \[/\1]/gs;

let str = `
[b]hello![/b]
[quote]
[url]http://google.com[/url]
[/quote]
`;

alert(str.match(regexp)); // [b]hello![/b], [quote][url]http://google.com[/url][/quote]
```

Tenga en cuenta que además de escapar `[` tuvimos que escapar de una barra para la etiqueta de cierre `\[/\1]`, porque normalmente la barra cierra el patrón.

## A formulación

### Encuentra cadenas entre comillas

La solución: `"/"(\\.|[^"\\"])*"/g`.

El paso a paso:

- Primero buscamos una comilla de apertura `"`
- Luego, si tenemos una barra invertida `\\"` (tenemos que duplicarla en el patrón porque es un carácter especial). Luego, cualquier carácter está bien después de él (un punto).
- De lo contrario, tomamos cualquier carácter excepto una comilla (que significaría el final de la cadena) y una barra invertida (para evitar barras invertidas solitarias, la barra invertida solo se usa con algún otro símbolo después): `[^"\\"]`
- ...Y así sucesivamente hasta la comilla de cierre.

En acción:

```
let regexp = /"(\\.|[^"\\"])*"/g;
let str = ' .. "test me" .. "Say \\\"Hello\\\"!" .. "\\\\\\\\ \\\" .. ';

alert(str.match(regexp)); // "test me", "Say \\\"Hello\\\"!", "\\ \\\""
```

A formulación

## Encuentra la etiqueta completa

El inicio del patrón es obvio: `<style>`.

...Pero entonces no podemos simplemente escribir `<style.*?>`, porque `<styler>` coincidiría.

Necesitamos un espacio después `<style>` y luego, opcionalmente, algo más o el final `>`.

En el lenguaje de expresión regular: `<style(>|\s.*?>)`.

En acción:

```
let regexp = /<style(>|\s.*?>)/g;

alert('<style> <styler> <style test="...">>'.match(regexp)); // <style>, <style test="...">
```

A formulación

## Lookahead y lookbehind (revisar delante/detrás)

### Encontrar enteros no negativos

La expresión regular para un número entero es `\d+`.

Podemos excluir los negativos anteponiendo un “lookbehind negativo”: `(?<! - )\d+`.

Pero al probarlo, notamos un resultado de más:

```
let regexp = /(?<! -)\d+/g;

let str = "0 12 -5 123 -18";

console.log(str.match(regexp)); // 0, 12, 123, 8
```

Como puedes ver, hay coincidencia de `8`, con `-18`. Para excluirla necesitamos asegurarnos de que `regexp` no comience la búsqueda desde el medio de otro número (no coincidente).

Podemos hacerlo especificando otra precedencia “lookbehind negativo”: `(?<! - )(?! \d)\d+`. Ahora `(?<! \d)` asegura que la coincidencia no comienza después de otro dígito, justo lo que necesitamos.

También podemos unirlos en un único “lookbehind”:

```
let regexp = /(?<! [-\d])\d+/g;
```

```
let str = "0 12 -5 123 -18";

alert(str.match(regexp)); // 0, 12, 123
```

A formulación

## Insertar después de la cabecera

Para insertar algo después de la etiqueta `<body>`, primero debemos encontrarla. Para ello podemos usar la expresión regular `<body.*?>`.

En esta tarea no necesitamos modificar la etiqueta `<body>`. Solamente agregar texto después de ella.

Veamos cómo podemos hacerlo:

```
let str = '...<body style="..."><...';
str = str.replace(<body.*?>/, '$&<h1>Hello</h1>');

alert(str); // ...<body style="..."><h1>Hello</h1>...
```

En el string de reemplazo, `$&` significa la coincidencia misma, la parte del texto original que corresponde a `<body.*?>`. Es reemplazada por sí misma más `<h1>Hello</h1>`.

Una alternativa es el uso de “lookbehind”:

```
let str = '...<body style="..."><...';
str = str.replace(/(?<<body.*?>)/, `<h1>Hello</h1>`);

alert(str); // ...<body style="..."><h1>Hello</h1>...
```

Como puedes ver, solo está presente la parte “lookbehind” en esta expresión regular.

Esto funciona así:

- En cada posición en el texto:
- Verifica si está precedida por `<body.*?>`.
- Si es así, tenemos una coincidencia.

La etiqueta `<body.*?>` no será devuelta. El resultado de esta expresión regular es un string vacío, pero coincide solo en las posiciones precedidas por `<body.*?>`.

Entonces reemplaza la “línea vacía”, precedida por `<body.*?>`, con `<h1>Hello</h1>`. Esto es, la inserción después de `<body>`.

P.S. Los indicadores de Regexp tales como `s` y `i` también nos pueden ser útiles: `/<body.*?>/si`. El indicador `s` hace que el punto `.` coincida también con el carácter de salto de línea, y el indicador `i` hace que `<body>` también acepte coincidencias `<BODY>` en mayúsculas y minúsculas.

A formulación