

Parte 3

Artículos adicionales

JS

Ilya Kantor

Hecho el 13 de junio de 2021

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, binary arrays](#)
 - [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](#)
 - [Custom elements](#)
 - [Shadow DOM](#)
 - [Elemento template](#)
 - [Shadow DOM slots, composition](#)
 - [Estilo Shadow DOM](#)
 - [Shadow DOM and events](#)
- [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 and lookbehind
- Catastrophic backtracking
- 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.

¿Qué pasa si la ventana emergente se abre desde `onclick`, pero después de `setTimeout`? Eso es un poco complicado.

Intenta este código:

```
// open after 3 seconds
setTimeout(() => window.open("http://google.com"), 3000);
```

Las ventanas emergentes se abren en Chrome, pero se bloquean en Firefox.

...Si disminuimos el retraso, la ventana emergente también funciona en Firefox:

```
// open after 1 seconds
setTimeout(() => window.open("http://google.com"), 1000);
```

La diferencia es que Firefox trata al timeout de 2000ms o menos como aceptable, pero después elimina la “confianza” asumiendo que ahora está “fuera de la acción del usuario”. Entonces el primero está bloqueado, y el segundo no lo está.

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: Minimalizar una ventana

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.moveBy(x,y)`

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

`win.moveTo(x,y)`

Mover la ventana por coordenadas `(x,y)` en la pantalla.

`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.scrollToView(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 el pasado las páginas malignas abusaron 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 eso. Por si acaso, para algunas combinaciones de sistema operativo / navegador, asegura que el usuario esté en la nueva ventana ahora.
- Si queremos rastrear cuándo un visitante realmente usa nuestra aplicación web, Nosotros podemos rastrear `window.onfocus/onblur`. Eso nos permite suspender / reanudar las actividades en la página, animaciones etc. Pero tenga en cuenta que el evento `blur` significa que el visitante salió de la ventana, pero aún pueden observarlo. La ventana está al fondo, 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 icono 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 “desenfocar”.

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>
```

`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 incrustado). 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.

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>

i 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 clonan 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, 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 de qué sitio está abierto en la ventana deseada en este momento: el usuario podría navegar fuera y la ventana del remitente no tiene idea de ello.

Especificar `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( "Recibí: " + event.data );

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

El ejemplo completo:

<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

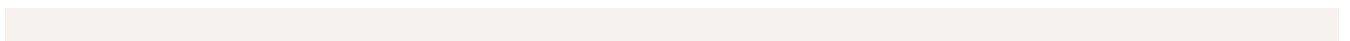
La idea es muy simple.

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>

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/UtEqQWPu7HKCx7Fj?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.

i 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 esta 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 “engañar” a los usuarios para que hagan clic en el sitio de una víctima sin siquiera 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, binary arrays

In web-development we meet binary data mostly while dealing with files (create, upload, download). Another typical use case is image processing.

That's all possible in JavaScript, and binary operations are high-performant.

Although, there's a bit of confusion, because there are many classes. To name a few:

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

Binary data in JavaScript is implemented in a non-standard way, compared to other languages. But when we sort things out, everything becomes fairly simple.

The basic binary object is `ArrayBuffer` – a reference to a fixed-length contiguous memory area.

We create it like this:

```
let buffer = new ArrayBuffer(16); // create a buffer of length 16
alert(buffer.byteLength); // 16
```

This allocates a contiguous memory area of 16 bytes and pre-fills it with zeroes.

⚠ `ArrayBuffer` is not an array of something

Let's eliminate a possible source of confusion. `ArrayBuffer` has nothing in common with `Array`:

- It has a fixed length, we can't increase or decrease it.
- It takes exactly that much space in the memory.
- To access individual bytes, another "view" object is needed, not `buffer[index]`.

`ArrayBuffer` is a memory area. What's stored in it? It has no clue. Just a raw sequence of bytes.

To manipulate an `ArrayBuffer`, we need to use a "view" object.

A view object does not store anything on it's own. It's the "eyeglasses" that give an interpretation of the bytes stored in the `ArrayBuffer`.

For instance:

- `Uint8Array` – treats each byte in `ArrayBuffer` as a separate number, with possible values from 0 to 255 (a byte is 8-bit, so it can hold only that much). Such value is called a "8-bit unsigned integer".
- `Uint16Array` – treats every 2 bytes as an integer, with possible values from 0 to 65535. That's called a "16-bit unsigned integer".
- `Uint32Array` – treats every 4 bytes as an integer, with possible values from 0 to 4294967295. That's called a "32-bit unsigned integer".
- `Float64Array` – treats every 8 bytes as a floating point number with possible values from 5.0×10^{-324} to 1.8×10^{308} .

So, the binary data in an `ArrayBuffer` of 16 bytes can be interpreted as 16 "tiny numbers", or 8 bigger numbers (2 bytes each), or 4 even bigger (4 bytes each), or 2 floating-point values with high precision (8 bytes each).

new ArrayBuffer(16)																
Uint8Array	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Uint16Array	0		1		2		3		4		5		6		7	
Uint32Array	0				1				2				3			
Float64Array	0								1							

`ArrayBuffer` is the core object, the root of everything, the raw binary data.

But if we're going to write into it, or iterate over it, basically for almost any operation – we must use a view, e.g:

```
let buffer = new ArrayBuffer(16); // create a buffer of length 16
```

```
let view = new Uint32Array(buffer); // treat buffer as a sequence of 32-bit integers
alert(Uint32Array.BYTES_PER_ELEMENT); // 4 bytes per integer

alert(view.length); // 4, it stores that many integers
alert(view.byteLength); // 16, the size in bytes

// let's write a value
view[0] = 123456;

// iterate over values
for(let num of view) {
  alert(num); // 123456, then 0, 0, 0 (4 values total)
}
```

TypedArray

The common term for all these views (`Uint8Array`, `Uint32Array`, etc) is [TypedArray](#). They share the same set of methods and properties.

Please note, there's no constructor called `TypedArray`, it's just a common “umbrella” term to represent one of views over `ArrayBuffer`: `Int8Array`, `Uint8Array` and so on, the full list will soon follow.

When you see something like `new TypedArray`, it means any of `new Int8Array`, `new Uint8Array`, etc.

Typed arrays behave like regular arrays: have indexes and are iterable.

A typed array constructor (be it `Int8Array` or `Float64Array`, doesn't matter) behaves differently depending on argument types.

There are 5 variants of arguments:

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

1. If an `ArrayBuffer` argument is supplied, the view is created over it. We used that syntax already.

Optionally we can provide `byteOffset` to start from (0 by default) and the `length` (till the end of the buffer by default), then the view will cover only a part of the `buffer`.

2. If an `Array`, or any array-like object is given, it creates a typed array of the same length and copies the content.

We can use it to pre-fill the array with the data:

```
let arr = new Uint8Array([0, 1, 2, 3]);
alert( arr.length ); // 4, created binary array of the same length
alert( arr[1] ); // 1, filled with 4 bytes (unsigned 8-bit integers) with given values
```

3. If another `TypedArray` is supplied, it does the same: creates a typed array of the same length and copies values. Values are converted to the new type in the process, if needed.

```
let arr16 = new Uint16Array([1, 1000]);
let arr8 = new Uint8Array(arr16);
alert( arr8[0] ); // 1
alert( arr8[1] ); // 232, tried to copy 1000, but can't fit 1000 into 8 bits (explanations below)
```

4. For a numeric argument `length` – creates the typed array to contain that many elements. Its byte length will be `length` multiplied by the number of bytes in a single item `TypedArray.BYTES_PER_ELEMENT`:

```
let arr = new Uint16Array(4); // create typed array for 4 integers
alert( Uint16Array.BYTES_PER_ELEMENT ); // 2 bytes per integer
alert( arr.byteLength ); // 8 (size in bytes)
```

5. Without arguments, creates an zero-length typed array.

We can create a `TypedArray` directly, without mentioning `ArrayBuffer`. But a view cannot exist without an underlying `ArrayBuffer`, so gets created automatically in all these cases except the first one (when provided).

To access the `ArrayBuffer`, there are properties:

- `arr.buffer` – references the `ArrayBuffer`.
- `arr.byteLength` – the length of the `ArrayBuffer`.

So, we can always move from one view to another:

```
let arr8 = new Uint8Array([0, 1, 2, 3]);

// another view on the same data
let arr16 = new Uint16Array(arr8.buffer);
```

Here's the list of typed arrays:

- `Uint8Array`, `Uint16Array`, `Uint32Array` – for integer numbers of 8, 16 and 32 bits.
 - `Uint8ClampedArray` – for 8-bit integers, “clamps” them on assignment (see below).
- `Int8Array`, `Int16Array`, `Int32Array` – for signed integer numbers (can be negative).
- `Float32Array`, `Float64Array` – for signed floating-point numbers of 32 and 64 bits.

⚠ No `int8` or similar single-valued types

Please note, despite of the names like `Int8Array`, there's no single-value type like `int`, or `int8` in JavaScript. That's logical, as `Int8Array` is not an array of these individual values, but rather a view on `ArrayBuffer`.

Out-of-bounds behavior

What if we attempt to write an out-of-bounds value into a typed array? There will be no error. But extra bits are cut-off.

For instance, let's try to put 256 into `Uint8Array`. In binary form, 256 is `100000000` (9 bits), but `Uint8Array` only provides 8 bits per value, that makes the available range from 0 to 255.

For bigger numbers, only the rightmost (less significant) 8 bits are stored, and the rest is cut off:

8-bit integer
1 00000000 256

So we'll get zero.

For 257, the binary form is `100000001` (9 bits), the rightmost 8 get stored, so we'll have `1` in the array:

8-bit integer
1 00000001 257

In other words, the number modulo 2^8 is saved.

Here's the demo:

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

let num = 256;
alert(num.toString(2)); // 100000000 (binary representation)

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

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

`Uint8ClampedArray` is special in this aspect, its behavior is different. It saves 255 for any number that is greater than 255, and 0 for any negative number. That behavior is useful for image processing.

TypedArray methods

`TypedArray` has regular `Array` methods, with notable exceptions.

We can iterate, `map`, `slice`, `find`, `reduce` etc.

There are few things we can't do though:

- No `splice` – we can't "delete" a value, because typed arrays are views on a buffer, and these are fixed, contiguous areas of memory. All we can do is to assign a zero.
- No `concat` method.

There are two additional methods:

- `arr.set(fromArr, [offset])` copies all elements from `fromArr` to the `arr`, starting at position `offset` (0 by default).
- `arr.subarray([begin, end])` creates a new view of the same type from `begin` to `end` (exclusive). That's similar to `slice` method (that's also supported), but doesn't copy anything – just creates a new view, to operate on the given piece of data.

These methods allow us to copy typed arrays, mix them, create new arrays from existing ones, and so on.

DataView

`DataView` [↗](#) is a special super-flexible "untyped" view over `ArrayBuffer`. It allows to access the data on any offset in any format.

- For typed arrays, the constructor dictates what the format is. The whole array is supposed to be uniform. The *i*-th number is `arr[i]`.
- With `DataView` we access the data with methods like `.getUint8(i)` or `.getUint16(i)`. We choose the format at method call time instead of the construction time.

The syntax:

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

- **buffer** – the underlying `ArrayBuffer`. Unlike typed arrays, `DataView` doesn't create a buffer on its own. We need to have it ready.
- **byteOffset** – the starting byte position of the view (by default 0).
- **byteLength** – the byte length of the view (by default till the end of `buffer`).

For instance, here we extract numbers in different formats from the same buffer:

```
// binary array of 4 bytes, all have the maximal value 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

// now get 16-bit number at offset 0, it consists of 2 bytes, together interpreted as 65535
alert( dataView.getUint16(0) ); // 65535 (biggest 16-bit unsigned int)

// get 32-bit number at offset 0
alert( dataView.getUint32(0) ); // 4294967295 (biggest 32-bit unsigned int)

dataView.setUint32(0, 0); // set 4-byte number to zero, thus setting all bytes to 0
```

`DataView` is great when we store mixed-format data in the same buffer. For example, when we store a sequence of pairs (16-bit integer, 32-bit float), `DataView` allows to access them easily.

Summary

`ArrayBuffer` is the core object, a reference to the fixed-length contiguous memory area.

To do almost any operation on `ArrayBuffer`, we need a view.

- It can be a `TypedArray`:
 - `Uint8Array`, `Uint16Array`, `Uint32Array` – for unsigned integers of 8, 16, and 32 bits.
 - `Uint8ClampedArray` – for 8-bit integers, “clamps” them on assignment.
 - `Int8Array`, `Int16Array`, `Int32Array` – for signed integer numbers (can be negative).
 - `Float32Array`, `Float64Array` – for signed floating-point numbers of 32 and 64 bits.
- Or a `DataView` – the view that uses methods to specify a format, e.g. `getUint8(offset)`.

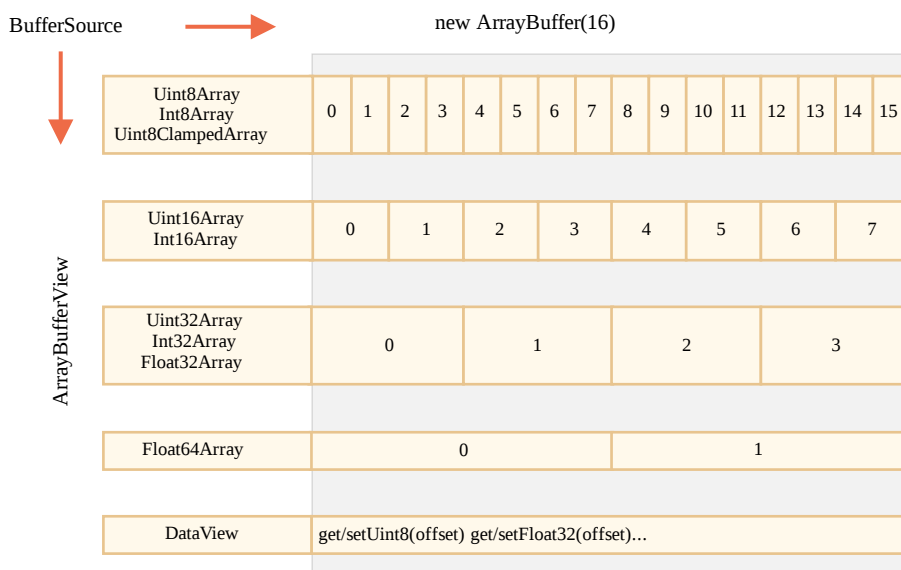
In most cases we create and operate directly on typed arrays, leaving `ArrayBuffer` under cover, as a “common denominator”. We can access it as `.buffer` and make another view if needed.

There are also two additional terms, that are used in descriptions of methods that operate on binary data:

- `ArrayBufferView` is an umbrella term for all these kinds of views.
- `BufferSource` is an umbrella term for `ArrayBuffer` or `ArrayBufferView`.

We’ll see these terms in the next chapters. `BufferSource` is one of the most common terms, as it means “any kind of binary data” – an `ArrayBuffer` or a view over it.

Here’s a cheatsheet:



✔ Tareas

Concatenate typed arrays

Given an array of `Uint8Array`, write a function `concat(arrays)` that returns a concatenation of them into a single 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 texto de un conjunto de datos binarios y convertirlo en un dato de tipo 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();

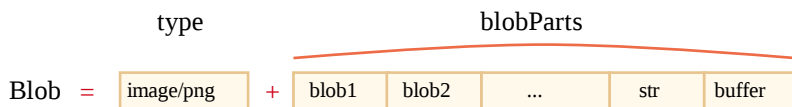
let uint8Array = encoder.encode("Hola");
alert(uint8Array); // 72,111,108,97
```

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.

i 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 clickeable, 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>
```

**i 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 metodos `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 el 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 **A**synchronous **J**avaScript **A**nd **X**ML, en español: “JavaScript y XML Asíncronico”) 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 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 quisiéramos 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 (e.g. JSON),
  - Objeto `FormData`, para enviar información como `form/multipart`,
  - `Blob/BufferSource` para enviar información en formato binario,
  - `URLSearchParams`, para enviar información en código `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, cómo 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>

```



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 similar al 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` (explicado en [el siguiente capítulo](#)),
- `response.blob()` – devuelve la respuesta como `Blob` (datos binarios tipados),
- `response.arrayBuffer()` – devuelve la respuesta como un objeto `ArrayBuffer`

Opciones de fetch hasta el momento:

- `method` – metodo 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 especifica `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.](#) ↗

[A solución](#)

## 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

Enviemos 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>

```

John Smith Submit

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 sólo un campo existe 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:  No file chosen

## 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>

```



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 enviara 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 crearlo 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 el cuerpo fragmento a fragmento, tal como viene. Las transmisiones legibles 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);
 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);
```

Explicemos esto paso a paso:

1. 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.

2. 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.

3. 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).

4. 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:

1. Creamos `chunksAll = new Uint8Array(selectedLength)` – una matriz del mismo tipo con la longitud combinada.
  2. Luego usa el método `.set(chunk, position)` para copiar cada `chunk` uno tras otro en él.
5. 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 asíncronas 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;

// El 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 realizar una petición a `http://example.com`:

```

try {
 await fetch('http://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 que 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 fundacional 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>`. Las personas lo utilizaban para enviar el resultado hacia un `<iframe>`, y de este modo mantenerse en el mismo sitio, de este modo:

```
<!-- 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 cabeceras 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, comencemos 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
...
```

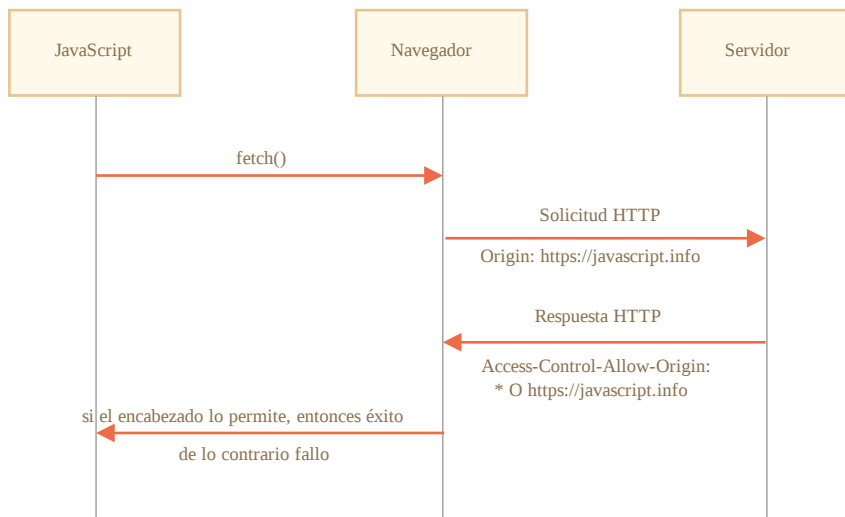
Tal como se puede ver, la cabecera `Origin` contiene exactamente el origen (protocolo/dominio/puerto), sin el path.

El servidor puede inspeccionar el origen `Origin` y, si está 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-Type`
- `Expires`
- `Last-Modified`
- `Pragma`

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

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

Como se puede ver, ¡no está la cabecera `Content-Length` en la lista!

Esta cabecera contiene el tamaño total de la respuesta. Por lo que si queremos mostrar el progreso de la descarga, en ese caso necesitaremos un permiso adicional para acceder a ese campo de la cabecera.

Para permitirle a JavaScript acceso a otros tipos de cabeceras, 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
API-Key: 2c9de507f2c54aa1
```

```
Access-Control-Allow-Origin: https://javascript.info
Access-Control-Expose-Headers: Content-Length, API-Key
```

Con este valor de `Access-Control-Expose-Headers`, el script tendrá permitido acceder a los valores de las cabeceras `Content-Length` 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.

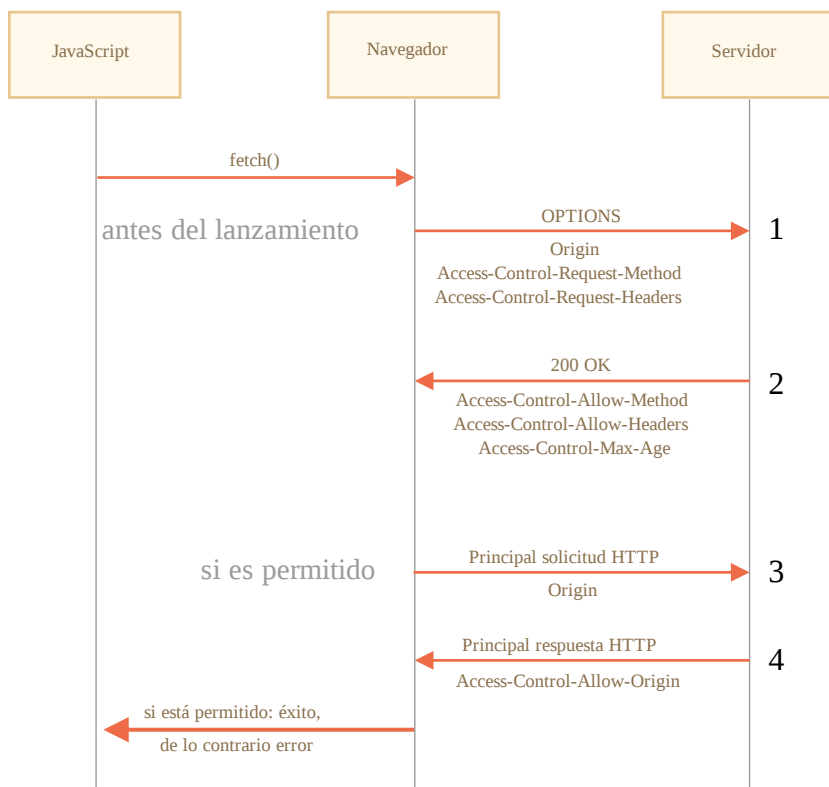
Por lo tanto, para evitar malentendidos, cualquier solicitud “insegura” (Estas que no podían ser realizadas en los viejos tiempos), no será realizada por el navegador en forma directa. Antes, enviará una solicitud preliminar llamada solicitud de “pre-vuelo”, solicitando que se le concedan los permisos.

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

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.

##### 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 debera enviar explícitamente 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 a 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 que 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.

### 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 Referer,
 // o una URL del origen actual
 referrerPolicy: "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 referer, 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](#) :

- **"no-referrer-when-downgrade"** – el valor predeterminado: 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 `Referer` 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.
- **"strict-origin-when-cross-origin"** – 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.
- **"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
"no-referrer"	-	-	-
"no-referrer-when-downgrade" o "" (predeterminado)	completo	completo	-
"origin"	origen	origen	origen
"origin-when-cross-origin"	completo	origen	origen
"same-origin"	completo	-	-
"strict-origin"	origen	origen	-
"strict-origin-when-cross-origin"	completo	origen	-
"unsafe-url"	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 `Accept-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);

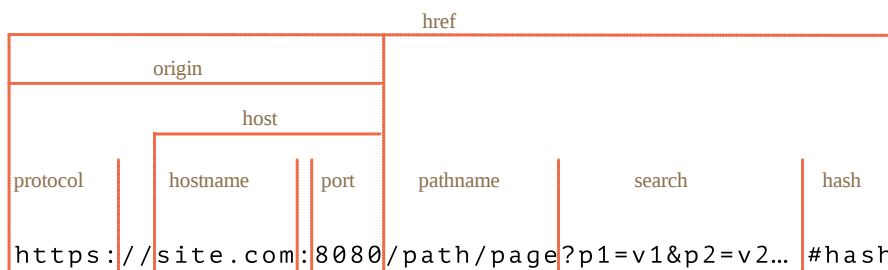
alert(newUrl); // https://javascript.info/profile/tester
```

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: `url.searchParams`, 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 parametros 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/Tect');

url.searchParams.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 `Tect` 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:

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

Una pregunta natural es: "¿Cuál es la diferencia entre `encodeURIComponent` y `encodeURIComponent`? ¿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.

- `encodeURIComponent` 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 `encodeURIComponent`:

```
// Usando caracteres cirílicos en el path URL
let url = encodeURIComponent('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 `encodeURIComponent`:

```
let music = encodeURIComponent('Rock&Roll');

let url = `https://google.com/search?q=${music}`;
alert(url); // https://google.com/search?q=Rock&Roll
```

Como podemos ver, `encodeURIComponent` 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

#### **i** 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(encodeURIComponent(url)); // http://%5B2607:f8b0:4005:802::1007%5D/
alert(new URL(url)); // http://[2607:f8b0:4005:802::1007]/
```

Como podemos ver, `encodeURIComponent` 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/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, binary arrays](#)),
- `"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!
};
```

**i 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 síncronas. 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 sobrescriben.

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 dividir las 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 sobrescribe 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 empezado.
- `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 quizá 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", true);

// 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 esta la demostración con el código tanto del cliente como del servidor, escrito en Node.js.

Esto funciona solo parcialmente en este sitio, ya que Node.js esta detrás de otro servidor llamado Nginx, que almacena cargas, pasándolas a Node.js cuando esta completamente lleno.

Pero puedes cargarlo y ejecutarlo localmente para la demostración completa:

<https://plnkr.co/edit/jAV1UjbuOz4ZJSZr?p=preview>

Como podemos ver, los métodos de red modernos estan 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

Sondeo largo es la forma más sencilla de tener una conexión persistente con el servidor, que no utiliza ningún protocolo específico como WebSocket o Eventos enviados por el servidor.

Al ser muy fácil de implementar, también es suficientemente bueno en muchos casos.

## Sondeo regular

La forma más sencilla de obtener nueva información del 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 de mensajes que recibió hasta ese momento.

Eso funciona, pero hay desventajas:

1. Los mensajes se transmiten con un retraso de hasta 10 segundos (entre solicitudes).
2. Incluso si no hay mensajes, el servidor se bombardea con solicitudes cada 10 segundos, aunque el usuario haya cambiado a otro lugar o esté dormido. Eso es bastante difícil de manejar, hablando en términos de rendimiento.

Entonces, si hablamos de un servicio muy pequeño, el enfoque puede ser viable, pero en general, necesita una mejora.

## Sondeo largo

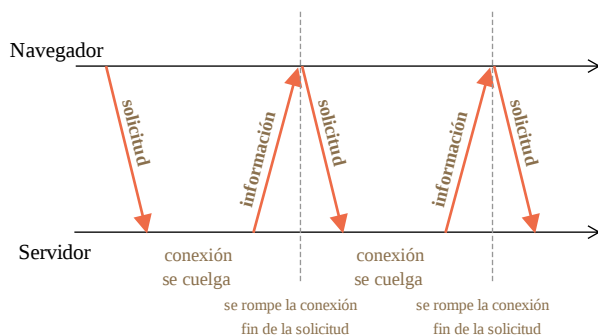
El llamado “sondeo largo” es una forma mucho mejor de sondear el servidor.

También es muy fácil de implementar y envía mensajes sin demoras.

El flujo:

1. Se envía una solicitud al servidor.
2. El servidor no cierra la conexión hasta que tiene un mensaje para enviar.
3. Cuando aparece un mensaje, el servidor responde a la solicitud con él.
4. El navegador realiza una nueva solicitud de inmediato.

La situación en la que el navegador envió una solicitud y tiene una conexión pendiente con el servidor, es estándar para este método. Solo cuando se entrega un mensaje, se restablece la conexión.



Si se pierde la conexión, por ejemplo, debido a un error de red, el navegador envía inmediatamente una nueva solicitud.

Un esquema de la función de suscripción del lado del cliente 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
 await 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));
 await subscribe();
 } else {
 // Recibe y muestra el mensaje
 let message = await response.text();
 showMessage(message);
 // Llama a subscribe () nuevamente para obtener el siguiente mensaje
 await subscribe();
 }
}

subscribe();

```

Como puedes ver, la función `subscribe` realiza una búsqueda, luego espera la respuesta, la maneja y se llama a sí misma nuevamente.

#### ⚠ El servidor debería estar bien aún con muchas conexiones pendientes

La arquitectura del servidor debe poder funcionar 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 estos problemas.

Dicho esto, no es un problema del lenguaje sino de 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

Aquí hay un chat de demostración, también puedes descargarlo y ejecutarlo 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 es raro recibir mensajes.

Si los mensajes llegan con mucha frecuencia, entonces el gráfico de mensajes solicitados vs recibidos, pintado arriba, se vuelve en forma de sierra.

Cada mensaje es una solicitud separada, provista de encabezados, sobrecarga de autenticación, etc.

Entonces, en este caso, se prefiere otro método, como [WebSocket](#) o [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 de HTTP “HTTP-requests” adicionales.

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 conocen el 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 del que envía y los desencripta para el que recibe. 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] ${error.message}`);
};
};

```

Para propósitos de demostración, hay 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`.

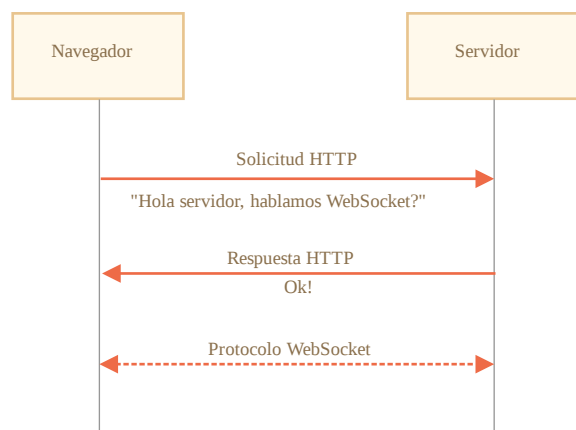
Eso es realmente todo, ya podemos conversar con WebSocket. Bastante simple, ¿no es cierto?

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 “header”) le pregunta al servidor: “¿Soportas Websockets?” y 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+1YFgZWcXczP8Q==
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 seguridad aleatoria generada por el navegador.
- **Sec-WebSocket-Version** – Versión del protocolo `WebSocket`, 13 es la actual.

### 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: hsB1buDTkk24srzE0TBu1ZA1C2g=
```

Aquí `Sec-WebSocket-Accept` es `Sec-WebSocket-Key`, recodificado usando un algoritmo especial. El navegador lo usa para asegurarse de que la respuesta se corresponde a la petición.

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+1YFgZwcXczP8Q==
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: hsB1buDTkk24srzE0TBu1ZA1C2g=
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 otros pocos 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 de HTTP, pero diferentes. En particular, los 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 algo 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:

Send

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:

<code>WebSocket</code>	<code>EventSource</code>
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 "reabirla". 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.

**❗ 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 reintento 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

Cookies are small strings of data that are stored directly in the browser. They are a part of the HTTP protocol, defined by the [RFC 6265](#) specification.

Cookies are usually set by a web-server using the response `Set-Cookie` HTTP-header. Then, the browser automatically adds them to (almost) every request to the same domain using the `Cookie` HTTP-header.

One of the most widespread use cases is authentication:

1. Upon sign in, the server uses the `Set-Cookie` HTTP-header in the response to set a cookie with a unique “session identifier”.
2. Next time when the request is sent to the same domain, the browser sends the cookie over the net using the `Cookie` HTTP-header.
3. So the server knows who made the request.

We can also access cookies from the browser, using `document.cookie` property.

There are many tricky things about cookies and their options. In this chapter we'll cover them in detail.

### Reading from document.cookie

Assuming you're on a website, it's possible to see the cookies from it, like this:

```
// At javascript.info, we use Google Analytics for statistics,
// so there should be some cookies
```

```
alert(document.cookie); // cookie1=value1; cookie2=value2;...
```

The value of `document.cookie` consists of `name=value` pairs, delimited by `;`. Each one is a separate cookie.

To find a particular cookie, we can split `document.cookie` by `;`, and then find the right name. We can use either a regular expression or array functions to do that.

We leave it as an exercise for the reader. Also, at the end of the chapter you'll find helper functions to manipulate cookies.

## Writing to `document.cookie`

We can write to `document.cookie`. But it's not a data property, it's an accessor (getter/setter). An assignment to it is treated specially.

**A write operation to `document.cookie` updates only cookies mentioned in it, but doesn't touch other cookies.**

For instance, this call sets a cookie with the name `user` and value `John`:

```
document.cookie = "user=John"; // update only cookie named 'user'
alert(document.cookie); // show all cookies
```

If you run it, then probably you'll see multiple cookies. That's because the `document.cookie=` operation does not overwrite all cookies. It only sets the mentioned cookie `user`.

Technically, name and value can have any characters. To keep the valid formatting, they should be escaped using a built-in `encodeURIComponent` function:

```
// special characters (spaces), need encoding
let name = "my name";
let value = "John Smith"

// encodes the cookie as my%20name=John%20Smith
document.cookie = encodeURIComponent(name) + '=' + encodeURIComponent(value);

alert(document.cookie); // ...; my%20name=John%20Smith
```

### Limitations

There are few limitations:

- The `name=value` pair, after `encodeURIComponent`, should not exceed 4KB. So we can't store anything huge in a cookie.
- The total number of cookies per domain is limited to around 20+, the exact limit depends on the browser.

Cookies have several options, many of them are important and should be set.

The options are listed after `key=value`, delimited by `;`, like this:

```
document.cookie = "user=John; path=/; expires=Tue, 19 Jan 2038 03:14:07 GMT"
```

## path

- `path=/mypath`

The url path prefix must be absolute. It makes the cookie accessible for pages under that path. By default, it's the current path.

If a cookie is set with `path=/admin`, it's visible at pages `/admin` and `/admin/something`, but not at `/home` or `/adminpage`.

Usually, we should set `path` to the root: `path=/` to make the cookie accessible from all website pages.

## domain

- `domain=site.com`

A domain defines where the cookie is accessible. In practice though, there are limitations. We can't set any domain.

By default, a cookie is accessible only at the domain that set it. So, if the cookie was set by `site.com`, we won't get it at `other.com`.

...But what's more tricky, we also won't get the cookie at a subdomain `forum.site.com`!

```
// at site.com
document.cookie = "user=John"

// at forum.site.com
alert(document.cookie); // no user
```

**There's no way to let a cookie be accessible from another 2nd-level domain, so `other.com` will never receive a cookie set at `site.com`.**

It's a safety restriction, to allow us to store sensitive data in cookies, that should be available only on one site.

...But if we'd like to allow subdomains like `forum.site.com` to get a cookie, that's possible. When setting a cookie at `site.com`, we should explicitly set the `domain` option to the root domain: `domain=site.com`:

```
// at site.com
// make the cookie accessible on any subdomain *.site.com:
document.cookie = "user=John; domain=site.com"

// later

// at forum.site.com
alert(document.cookie); // has cookie user=John
```

For historical reasons, `domain=.site.com` (a dot before `site.com`) also works the same way, allowing access to the cookie from subdomains. That's an old notation and should be used if we need to support very old browsers.

So, the `domain` option allows to make a cookie accessible at subdomains.

## expires, max-age

By default, if a cookie doesn't have one of these options, it disappears when the browser is closed. Such cookies are called "session cookies"

To let cookies survive a browser close, we can set either the `expires` or `max-age` option.

- `expires=Tue, 19 Jan 2038 03:14:07 GMT`

The cookie expiration date defines the time, when the browser will automatically delete it.

The date must be exactly in this format, in the GMT timezone. We can use `date.toUTCString()` to get it. For instance, we can set the cookie to expire in 1 day:

```
// +1 day from now
let date = new Date(Date.now() + 86400e3);
date = date.toUTCString();
document.cookie = "user=John; expires=" + date;
```

If we set `expires` to a date in the past, the cookie is deleted.

- `max-age=3600`

Is an alternative to `expires` and specifies the cookie's expiration in seconds from the current moment.

If set to zero or a negative value, the cookie is deleted:

```
// cookie will die in +1 hour from now
document.cookie = "user=John; max-age=3600";
```



```
// delete cookie (let it expire right now)
document.cookie = "user=John; max-age=0";
```

## secure

- **secure**

The cookie should be transferred only over HTTPS.

**By default, if we set a cookie at `http://site.com`, then it also appears at `https://site.com` and vice versa.**

That is, cookies are domain-based, they do not distinguish between the protocols.

With this option, if a cookie is set by `https://site.com`, then it doesn't appear when the same site is accessed by HTTP, as `http://site.com`. So if a cookie has sensitive content that should never be sent over unencrypted HTTP, the `secure` flag is the right thing.

```
// assuming we're on https:// now
// set the cookie to be secure (only accessible over HTTPS)
document.cookie = "user=John; secure";
```

## samesite

That's another security attribute `samesite`. It's designed to protect from so-called XSRF (cross-site request forgery) attacks.

To understand how it works and when it's useful, let's take a look at XSRF attacks.

### XSRF attack

Imagine, you are logged into the site `bank.com`. That is: you have an authentication cookie from that site. Your browser sends it to `bank.com` with every request, so that it recognizes you and performs all sensitive financial operations.

Now, while browsing the web in another window, you accidentally come to another site `evil.com`. That site has JavaScript code that submits a form `<form action="https://bank.com/pay">` to `bank.com` with fields that initiate a transaction to the hacker's account.

The browser sends cookies every time you visit the site `bank.com`, even if the form was submitted from `evil.com`. So the bank recognizes you and actually performs the payment.



That's a so-called "Cross-Site Request Forgery" (in short, XSRF) attack.

Real banks are protected from it of course. All forms generated by `bank.com` have a special field, a so-called "XSRF protection token", that an evil page can't generate or extract from a remote page. It can submit a form there, but can't get the data back. The site `bank.com` checks for such token in every form it receives.

Such a protection takes time to implement though. We need to ensure that every form has the required token field, and we must also check all requests.

### Enter cookie `samesite` option

The cookie `samesite` option provides another way to protect from such attacks, that (in theory) should not require "xsrp protection tokens".

It has two possible values:

- **`samesite=strict` (same as `samesite` without value)**

A cookie with `samesite=strict` is never sent if the user comes from outside the same site.

In other words, whether a user follows a link from their mail or submits a form from `evil.com`, or does any operation that originates from another domain, the cookie is not sent.

If authentication cookies have the `samesite` option, then a XSRF attack has no chances to succeed, because a submission from `evil.com` comes without cookies. So `bank.com` will not recognize the user and will not proceed with the payment.

The protection is quite reliable. Only operations that come from `bank.com` will send the `samesite` cookie, e.g. a form submission from another page at `bank.com`.

Although, there's a small inconvenience.

When a user follows a legitimate link to `bank.com`, like from their own notes, they'll be surprised that `bank.com` does not recognize them. Indeed, `samesite=strict` cookies are not sent in that case.

We could work around that by using two cookies: one for "general recognition", only for the purposes of saying: "Hello, John", and the other one for data-changing operations with `samesite=strict`. Then, a person coming from outside of the site will see a welcome, but payments must be initiated from the bank's website, for the second cookie to be sent.

- **`samesite=lax`**

A more relaxed approach that also protects from XSRF and doesn't break the user experience.

Lax mode, just like `strict`, forbids the browser to send cookies when coming from outside the site, but adds an exception.

A `samesite=lax` cookie is sent if both of these conditions are true:

1. The HTTP method is "safe" (e.g. GET, but not POST).

The full list of safe HTTP methods is in the [RFC7231 specification](#). Basically, these are the methods that should be used for reading, but not writing the data. They must not perform any data-changing operations. Following a link is always GET, the safe method.

2. The operation performs a top-level navigation (changes URL in the browser address bar).

That's usually true, but if the navigation is performed in an `<iframe>`, then it's not top-level. Also, JavaScript methods for network requests do not perform any navigation, hence they don't fit.

So, what `samesite=lax` does, is to basically allow the most common "go to URL" operation to have cookies. E.g. opening a website link from notes that satisfy these conditions.

But anything more complicated, like a network request from another site or a form submission, loses cookies.

If that's fine for you, then adding `samesite=lax` will probably not break the user experience and add protection.

Overall, `samesite` is a great option.

There's a drawback:

- `samesite` is ignored (not supported) by very old browsers, year 2017 or so.

**So if we solely rely on `samesite` to provide protection, then old browsers will be vulnerable.**

But we surely can use `samesite` together with other protection measures, like xsrf tokens, to add an additional layer of defence and then, in the future, when old browsers die out, we'll probably be able to drop xsrf tokens.

## httpOnly

This option has nothing to do with JavaScript, but we have to mention it for completeness.

The web-server uses the `Set-Cookie` header to set a cookie. Also, it may set the `httpOnly` option.

This option forbids any JavaScript access to the cookie. We can't see such a cookie or manipulate it using `document.cookie`.

That's used as a precaution measure, to protect from certain attacks when a hacker injects his own JavaScript code into a page and waits for a user to visit that page. That shouldn't be possible at all, hackers should not be able to inject their code into our site, but there may be bugs that let them do it.

Normally, if such a thing happens, and a user visits a web-page with hacker's JavaScript code, then that code executes and gains access to `document.cookie` with user cookies containing authentication information. That's bad.

But if a cookie is `httpOnly`, then `document.cookie` doesn't see it, so it is protected.

## Appendix: Cookie functions

Here's a small set of functions to work with cookies, more convenient than a manual modification of `document.cookie`.

There exist many cookie libraries for that, so these are for demo purposes. Fully working though.

### getCookie(name)

The shortest way to access a cookie is to use a [regular expression](#).

The function `getCookie(name)` returns the cookie with the given `name`:

```
// returns the cookie with the given name,
// or undefined if not found
function getCookie(name) {
 let matches = document.cookie.match(new RegExp(
 "(?:^|;)" + name.replace(/[.\$?*[\]()\|\\\/\+^]/g, '\\\\$1') + "=([^\s;]*)"
));
 return matches ? decodeURIComponent(matches[1]) : undefined;
}
```

Here `new RegExp` is generated dynamically, to match `; name=<value>`.

Please note that a cookie value is encoded, so `getCookie` uses a built-in `decodeURIComponent` function to decode it.

### setCookie(name, value, options)

Sets the cookie's `name` to the given `value` with `path=/` by default (can be modified to add other defaults):

```
function setCookie(name, value, options = {}) {

 options = {
 path: '/',
 // add other defaults here if necessary
 ...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;
}

// Example of use:
setCookie('user', 'John', {secure: true, 'max-age': 3600});
```

### deleteCookie(name)

To delete a cookie, we can call it with a negative expiration date:

```
function deleteCookie(name) {
 setCookie(name, "", {
 'max-age': -1
 })
}
```

#### Updating or deleting must use same path and domain

Please note: when we update or delete a cookie, we should use exactly the same path and domain options as when we set it.

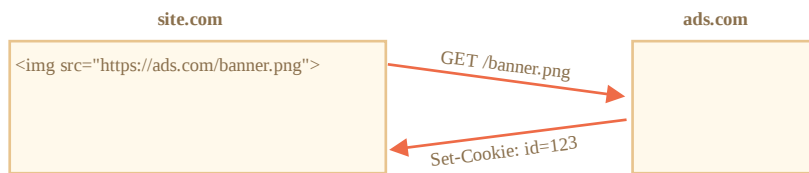
Together: [cookie.js](#).

## Appendix: Third-party cookies

A cookie is called “third-party” if it’s placed by a domain other than the page the user is visiting.

For instance:

1. A page at `site.com` loads a banner from another site: ``.
2. Along with the banner, the remote server at `ads.com` may set the `Set-Cookie` header with a cookie like `id=1234`. Such a cookie originates from the `ads.com` domain, and will only be visible at `ads.com`:



3. Next time when `ads.com` is accessed, the remote server gets the `id` cookie and recognizes the user:



4. What’s even more important is, when the user moves from `site.com` to another site `other.com`, which also has a banner, then `ads.com` gets the cookie, as it belongs to `ads.com`, thus recognizing the visitor and tracking him as he moves between sites:



Third-party cookies are traditionally used for tracking and ads services, due to their nature. They are bound to the originating domain, so `ads.com` can track the same user between different sites, if they all access it.

Naturally, some people don’t like being tracked, so browsers allow to disable such cookies.

Also, some modern browsers employ special policies for such cookies:

- Safari does not allow third-party cookies at all.
- Firefox comes with a “black list” of third-party domains where it blocks third-party cookies.

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

If we load a script from a third-party domain, like `<script src="https://google-analytics.com/analytics.js">`, and that script uses `document.cookie` to set a cookie, then such cookie is not third-party.

If a script sets a cookie, then no matter where the script came from – the cookie belongs to the domain of the current webpage.

## **Appendix: GDPR**

This topic is not related to JavaScript at all, just something to keep in mind when setting cookies.

There’s a legislation in Europe called GDPR, that enforces a set of rules for websites to respect the users’ privacy. One of these rules is to require an explicit permission for tracking cookies from the user.

Please note, that's only about tracking/identifying/authorizing cookies.

So, if we set a cookie that just saves some information, but neither tracks nor identifies the user, then we are free to do it.

But if we are going to set a cookie with an authentication session or a tracking id, then a user must allow that.

Websites generally have two variants of following GDPR. You must have seen them both already in the web:

1. If a website wants to set tracking cookies only for authenticated users.

To do so, the registration form should have a checkbox like “accept the privacy policy” (that describes how cookies are used), the user must check it, and then the website is free to set auth cookies.

2. If a website wants to set tracking cookies for everyone.

To do so legally, a website shows a modal “splash screen” for newcomers, and requires them to agree to the cookies. Then the website can set them and let people see the content. That can be disturbing for new visitors though. No one likes to see such “must-click” modal splash screens instead of the content. But GDPR requires an explicit agreement.

GDPR is not only about cookies, it's about other privacy-related issues too, but that's too much beyond our scope.

## Summary

`document.cookie` provides access to cookies

- write operations modify only cookies mentioned in it.
- name/value must be encoded.
- one cookie must not exceed 4KB, 20+ cookies per site (depends on the browser).

Cookie options:

- `path=/` , by default current path, makes the cookie visible only under that path.
- `domain=site.com` , by default a cookie is visible on the current domain only. If the domain is set explicitly, the cookie becomes visible on subdomains.
- `expires` or `max-age` sets the cookie expiration time. Without them the cookie dies when the browser is closed.
- `secure` makes the cookie HTTPS-only.
- `samesite` forbids the browser to send the cookie with requests coming from outside the site. This helps to prevent XSRF attacks.

Additionally:

- Third-party cookies may be forbidden by the browser, e.g. Safari does that by default.
- When setting a tracking cookie for EU citizens, GDPR requires to ask for permission.

## 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 navegadores permiten almacenar, como mínimo, 2 megabytes de datos (o más) y tienen opciones para configurar éstos límites.
- El servidor no puede manipular los objetos de almacenaje via cabeceras HTTP, todo se hace via 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.

Ámbos 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(índice)` – 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 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 cojer el ítem que hemos guardado de éste 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;

// coje 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, mientras que el acceso tipo 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.

Cualquier otro tipo, como un número o un objeto, se convierte 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 ésta razón `sessionStorage` se usa relativamente poco.

## Evento storage

Cuando los datos se actualizan en `localStorage` o en `sessionStorage`, el evento se dispara [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 => { // igual que en 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, cadenas de texto.
- El límite es de más de 5mb+; depende 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)` – coje el valor de una clave.
- `removeItem(clave)` – borra una clave con su valor.
- `clear()` – borra todo.
- `key(índice)` – coje 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:

Write here

Clear

[Abrir un entorno controlado para la tarea.](#)

[A solución](#)

## IndexedDB

IndexedDB is a database that is built into a browser, much more powerful than `localStorage`.

- Stores almost any kind of values by keys, multiple key types.
- Supports transactions for reliability.
- Supports key range queries, indexes.

- Can store much bigger volumes of data than `localStorage`.

That power is usually excessive for traditional client-server apps. IndexedDB is intended for offline apps, to be combined with ServiceWorkers and other technologies.

The native interface to IndexedDB, described in the specification <https://www.w3.org/TR/IndexedDB>, is event-based.

We can also use `async/await` with the help of a promise-based wrapper, like <https://github.com/jakearchibald/idb>. That's pretty convenient, but the wrapper is not perfect, it can't replace events for all cases. So we'll start with events, and then, after we gain an understanding of IndexedDB, we'll use the wrapper.

#### Where's the data?

Technically, the data is usually stored in the visitor's home directory, along with browser settings, extensions, etc. Different browsers and OS-level users have each their own independent storage.

## Open database

To start working with IndexedDB, we first need to `open` (connect to) a database.

The syntax:

```
let openRequest = indexedDB.open(name, version);
```

- `name` – a string, the database name.
- `version` – a positive integer version, by default `1` (explained below).

We can have many databases with different names, but all of them exist within the current origin (domain/protocol/port). Different websites can't access each other's databases.

The call returns `openRequest` object, we should listen to events on it:

- `success`: database is ready, there's the "database object" in `openRequest.result`, we should use it for further calls.
- `error`: opening failed.
- `upgradeneeded`: database is ready, but its version is outdated (see below).

### IndexedDB has a built-in mechanism of "schema versioning", absent in server-side databases.

Unlike server-side databases, IndexedDB is client-side, the data is stored in the browser, so we, developers, don't have full-time access to it. So, when we have published a new version of our app, and the user visits our webpage, we may need to update the database.

If the local database version is less than specified in `open`, then a special event `upgradeneeded` is triggered, and we can compare versions and upgrade data structures as needed.

The `upgradeneeded` event also triggers when the database doesn't yet exist (technically, its version is `0`), so we can perform the initialization.

Let's say we published the first version of our app.

Then we can open the database with version `1` and perform the initialization in an `upgradeneeded` handler like this:

```
let openRequest = indexedDB.open("store", 1);

openRequest.onupgradeneeded = function() {
 // triggers if the client had no database
 // ...perform initialization...
};

openRequest.onerror = function() {
 console.error("Error", openRequest.error);
};

openRequest.onsuccess = function() {
 let db = openRequest.result;
 // continue working with database using db object
};
```

Then, later, we publish the 2nd version.

We can open it with version `2` and perform the upgrade like this:

```
let openRequest = indexedDB.open("store", 2);

openRequest.onupgradeneeded = function(event) {
 // the existing database version is less than 2 (or it doesn't exist)
 let db = openRequest.result;
 switch(event.oldVersion) { // existing db version
 case 0:
 // version 0 means that the client had no database
 // perform initialization
 case 1:
 // client had version 1
 // update
 }
};
```

Please note: as our current version is `2`, the `onupgradeneeded` handler has a code branch for version `0`, suitable for users that are accessing for the first time and have no database, and also for version `1`, for upgrades.

And then, only if `onupgradeneeded` handler finishes without errors, `openRequest.onsuccess` triggers, and the database is considered successfully opened.

To delete a database:

```
let deleteRequest = indexedDB.deleteDatabase(name)
// deleteRequest.onsuccess/onerror tracks the result
```

#### ⚠ We can't open an older version of the database

If the current user database has a higher version than in the `open` call, e.g. the existing DB version is `3`, and we try to `open(...2)`, then that's an error, `openRequest.onerror` triggers.

That's rare, but such a thing may happen when a visitor loads outdated JavaScript code, e.g. from a proxy cache. So the code is old, but his database is new.

To protect from errors, we should check `db.version` and suggest a page reload. Use proper HTTP caching headers to avoid loading the old code, so that you'll never have such problems.

### Parallel update problem

As we're talking about versioning, let's tackle a small related problem.

Let's say:

1. A visitor opened our site in a browser tab, with database version `1`.
2. Then we rolled out an update, so our code is newer.
3. And then the same visitor opens our site in another tab.

So there's a tab with an open connection to DB version `1`, while the second one attempts to update it to version `2` in its `upgradeneeded` handler.

The problem is that a database is shared between two tabs, as it's the same site, same origin. And it can't be both version `1` and `2`. To perform the update to version `2`, all connections to version `1` must be closed, including the one in the first tab.

In order to organize that, the `versionchange` event triggers on the "outdated" database object. We should listen for it and close the old database connection (and probably suggest a page reload, to load the updated code).

If we don't listen for the `versionchange` event and don't close the old connection, then the second, new connection won't be made. The `openRequest` object will emit the `blocked` event instead of `success`. So the second tab won't work.

Here's the code to correctly handle the parallel upgrade. It installs the `onversionchange` handler, that triggers if the current database connection becomes outdated (db version is updated elsewhere) and closes the connection.

```
let openRequest = indexedDB.open("store", 2);

openRequest.onupgradeneeded = ...;
```

```

openRequest.onerror = ...;

openRequest.onsuccess = function() {
 let db = openRequest.result;

 db.onversionchange = function() {
 db.close();
 alert("Database is outdated, please reload the page.")
 };

 // ...the db is ready, use it...
};

openRequest.onblocked = function() {
 // this event shouldn't trigger if we handle onversionchange correctly

 // it means that there's another open connection to the same database
 // and it wasn't closed after db.onversionchange triggered for it
};

```

...In other words, here we do two things:

1. The `db.onversionchange` listener informs us about a parallel update attempt, if the current database version becomes outdated.
2. The `openRequest.onblocked` listener informs us about the opposite situation: there's a connection to an outdated version elsewhere, and it doesn't close, so the newer connection can't be made.

We can handle things more gracefully in `db.onversionchange`, prompt the visitor to save the data before the connection is closed and so on.

Or, an alternative approach would be to not close the database in `db.onversionchange`, but instead use the `onblocked` handler (in the new tab) to alert the visitor, tell him that the newer version can't be loaded until they close other tabs.

These update collisions happen rarely, but we should at least have some handling for them, at least an `onblocked` handler, to prevent our script from dying silently.

## Object store

To store something in IndexedDB, we need an *object store*.

An object store is a core concept of IndexedDB. Counterparts in other databases are called “tables” or “collections”. It's where the data is stored. A database may have multiple stores: one for users, another one for goods, etc.

Despite being named an “object store”, primitives can be stored too.

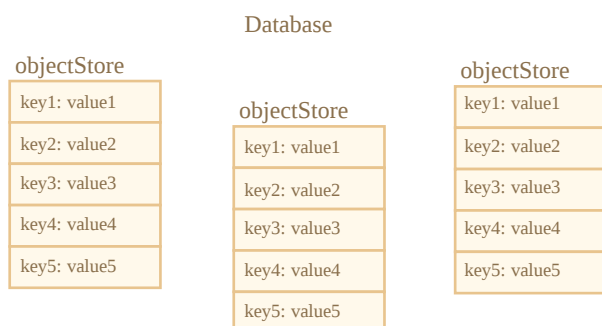
**We can store almost any value, including complex objects.**

IndexedDB uses the [standard serialization algorithm](#) to clone-and-store an object. It's like `JSON.stringify`, but more powerful, capable of storing much more datatypes.

An example of an object that can't be stored: an object with circular references. Such objects are not serializable. `JSON.stringify` also fails for such objects.

**There must be a unique key for every value in the store.**

A key must be one of these types – number, date, string, binary, or array. It's a unique identifier, so we can search/remove/update values by the key.



As we'll see very soon, we can provide a key when we add a value to the store, similar to `localStorage`. But when we store objects, IndexedDB allows setting up an object property as the key, which is much more convenient. Or we can auto-generate keys.

But we need to create an object store first.

The syntax to create an object store:

```
db.createObjectStore(name[, keyOptions]);
```

Please note, the operation is synchronous, no `await` needed.

- `name` is the store name, e.g. "books" for books,
- `keyOptions` is an optional object with one of two properties:
  - `keyPath` – a path to an object property that IndexedDB will use as the key, e.g. `id`.
  - `autoIncrement` – if `true`, then the key for a newly stored object is generated automatically, as an ever-incrementing number.

If we don't supply `keyOptions`, then we'll need to provide a key explicitly later, when storing an object.

For instance, this object store uses `id` property as the key:

```
db.createObjectStore('books', {keyPath: 'id'});
```

**An object store can only be created/modified while updating the DB version, in `upgradeneeded` handler.**

That's a technical limitation. Outside of the handler we'll be able to add/remove/update the data, but object stores can only be created/removed/alterd during a version update.

To perform a database version upgrade, there are two main approaches:

1. We can implement per-version upgrade functions: from 1 to 2, from 2 to 3, from 3 to 4 etc. Then, in `upgradeneeded` we can compare versions (e.g. old 2, now 4) and run per-version upgrades step by step, for every intermediate version (2 to 3, then 3 to 4).
2. Or we can just examine the database: get a list of existing object stores as `db.objectStoreNames`. That object is a [DOMStringList](#) that provides `contains(name)` method to check for existence. And then we can do updates depending on what exists and what doesn't.

For small databases the second variant may be simpler.

Here's the demo of the second approach:

```
let openRequest = indexedDB.open("db", 2);

// create/upgrade the database without version checks
openRequest.onupgradeneeded = function() {
 let db = openRequest.result;
 if (!db.objectStoreNames.contains('books')) { // if there's no "books" store
 db.createObjectStore('books', {keyPath: 'id'}); // create it
 }
};
```

To delete an object store:

```
db.deleteObjectStore('books')
```

## Transactions

The term “transaction” is generic, used in many kinds of databases.

A transaction is a group of operations, that should either all succeed or all fail.

For instance, when a person buys something, we need to:

1. Subtract the money from their account.

2. Add the item to their inventory.

It would be pretty bad if we complete the 1st operation, and then something goes wrong, e.g. lights out, and we fail to do the 2nd. Both should either succeed (purchase complete, good!) or both fail (at least the person kept their money, so they can retry).

Transactions can guarantee that.

**All data operations must be made within a transaction in IndexedDB.**

To start a transaction:

```
db.transaction(store[, type]);
```

- `store` is a store name that the transaction is going to access, e.g. `"books"`. Can be an array of store names if we're going to access multiple stores.
- `type` – a transaction type, one of:
  - `readonly` – can only read, the default.
  - `readwrite` – can only read and write the data, but not create/remove/alter object stores.

There's also `versionchange` transaction type: such transactions can do everything, but we can't create them manually. IndexedDB automatically creates a `versionchange` transaction when opening the database, for `updateneeded` handler. That's why it's a single place where we can update the database structure, create/remove object stores.

#### Why are there different types of transactions?

Performance is the reason why transactions need to be labeled either `readonly` and `readwrite`.

Many `readonly` transactions are able to access the same store concurrently, but `readwrite` transactions can't. A `readwrite` transaction "locks" the store for writing. The next transaction must wait before the previous one finishes before accessing the same store.

After the transaction is created, we can add an item to the store, like this:

```
let transaction = db.transaction("books", "readwrite"); // (1)

// get an object store to operate on it
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("Book added to the store", request.result);
};

request.onerror = function() {
 console.log("Error", request.error);
};
```

There were basically four steps:

1. Create a transaction, mentioning all the stores it's going to access, at (1).
2. Get the store object using `transaction.objectStore(name)`, at (2).
3. Perform the request to the object store `books.add(book)`, at (3).
4. ...Handle request success/error (4), then we can make other requests if needed, etc.

Object stores support two methods to store a value:

- **put(value, [key])** Add the `value` to the store. The `key` is supplied only if the object store did not have `keyPath` or `autoIncrement` option. If there's already a value with the same key, it will be replaced.

- **add(value, [key])** Same as `put`, but if there's already a value with the same key, then the request fails, and an error with the name `"ConstraintError"` is generated.

Similar to opening a database, we can send a request: `books.add(book)`, and then wait for `success/error` events.

- The `request.result` for `add` is the key of the new object.
- The error is in `request.error` (if any).

## Transactions' autocommit

In the example above we started the transaction and made `add` request. But as we stated previously, a transaction may have multiple associated requests, that must either all succeed or all fail. How do we mark the transaction as finished, with no more requests to come?

The short answer is: we don't.

In the next version 3.0 of the specification, there will probably be a manual way to finish the transaction, but right now in 2.0 there isn't.

**When all transaction requests are finished, and the `microtasks queue` is empty, it is committed automatically.**

Usually, we can assume that a transaction commits when all its requests are complete, and the current code finishes.

So, in the example above no special call is needed to finish the transaction.

Transactions auto-commit principle has an important side effect. We can't insert an async operation like `fetch`, `setTimeout` in the middle of a transaction. IndexedDB will not keep the transaction waiting till these are done.

In the code below, `request2` in the line `(*)` fails, because the transaction is already committed, and can't make any request in it:

```
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
 };
 });
};
```

That's because `fetch` is an asynchronous operation, a macrotask. Transactions are closed before the browser starts doing macrotasks.

Authors of IndexedDB spec believe that transactions should be short-lived. Mostly for performance reasons.

Notably, `readwrite` transactions "lock" the stores for writing. So if one part of the application initiated `readwrite` on `books` object store, then another part that wants to do the same has to wait: the new transaction "hangs" till the first one is done. That can lead to strange delays if transactions take a long time.

So, what to do?

In the example above we could make a new `db.transaction` right before the new request `(*)`.

But it will be even better, if we'd like to keep the operations together, in one transaction, to split apart IndexedDB transactions and "other" async stuff.

First, make `fetch`, prepare the data if needed, afterwards create a transaction and perform all the database requests, it'll work then.

To detect the moment of successful completion, we can listen to `transaction.oncomplete` event:

```
let transaction = db.transaction("books", "readwrite");

// ...perform operations...

transaction.oncomplete = function() {
 console.log("Transaction is complete");
};
```

Only `complete` guarantees that the transaction is saved as a whole. Individual requests may succeed, but the final write operation may go wrong (e.g. I/O error or something).

To manually abort the transaction, call:

```
transaction.abort();
```

That cancels all modification made by the requests in it and triggers `transaction.onabort` event.

## Error handling

Write requests may fail.

That's to be expected, not only because of possible errors at our side, but also for reasons not related to the transaction itself. For instance, the storage quota may be exceeded. So we must be ready to handle such case.

**A failed request automatically aborts the transaction, canceling all its changes.**

In some situations, we may want to handle the failure (e.g. try another request), without canceling existing changes, and continue the transaction. That's possible. The `request.onerror` handler is able to prevent the transaction abort by calling `event.preventDefault()`.

In the example below a new book is added with the same key (`id`) as the existing one. The `store.add` method generates a `"ConstraintError"` in that case. We handle it without canceling the transaction:

```
let transaction = db.transaction("books", "readwrite");

let book = { id: 'js', price: 10 };

let request = transaction.objectStore("books").add(book);

request.onerror = function(event) {
 // ConstraintError occurs when an object with the same id already exists
 if (request.error.name === "ConstraintError") {
 console.log("Book with such id already exists"); // handle the error
 event.preventDefault(); // don't abort the transaction
 // use another key for the book?
 } else {
 // unexpected error, can't handle it
 // the transaction will abort
 }
};

transaction.onabort = function() {
 console.log("Error", transaction.error);
};
```

## Event delegation

Do we need `onerror`/`onsuccess` for every request? Not every time. We can use event delegation instead.

**IndexedDB events bubble: `request` → `transaction` → `database`.**

All events are DOM events, with capturing and bubbling, but usually only bubbling stage is used.

So we can catch all errors using `db.onerror` handler, for reporting or other purposes:

```
db.onerror = function(event) {
 let request = event.target; // the request that caused the error

 console.log("Error", request.error);
};
```

...But what if an error is fully handled? We don't want to report it in that case.

We can stop the bubbling and hence `db.onerror` by using `event.stopPropagation()` in `request.onerror`.

```
request.onerror = function(event) {
 if (request.error.name === "ConstraintError") {
```



```

 console.log("Book with such id already exists"); // handle the error
 event.preventDefault(); // don't abort the transaction
 event.stopPropagation(); // don't bubble error up, "chew" it
 } else {
 // do nothing
 // transaction will be aborted
 // we can take care of error in transaction.onabort
 }
};

```

## Searching

There are two main types of search in an object store:

1. By a key value or a key range. In our “books” storage that would be a value or range of values of `book.id`.
2. By another object field, e.g. `book.price`. This required an additional data structure, named “index”.

### By key

First let's deal with the first type of search: by key.

Searching methods support both exact key values and so-called “ranges of values” – [IDBKeyRange](#) objects that specify an acceptable “key range”.

`IDBKeyRange` objects are created using following calls:

- `IDBKeyRange.lowerBound(lower, [open])` means:  $\geq$ lower (or  $>$ lower if `open` is true)
- `IDBKeyRange.upperBound(upper, [open])` means:  $\leq$ upper (or  $<$ upper if `open` is true)
- `IDBKeyRange.bound(lower, upper, [lowerOpen], [upperOpen])` means: between `lower` and `upper`. If the open flags is true, the corresponding key is not included in the range.
- `IDBKeyRange.only(key)` – a range that consists of only one `key`, rarely used.

We'll see practical examples of using them very soon.

To perform the actual search, there are following methods. They accept a `query` argument that can be either an exact key or a key range:

- `store.get(query)` – search for the first value by a key or a range.
- `store.getAll([query], [count])` – search for all values, limit by `count` if given.
- `store.getKey(query)` – search for the first key that satisfies the query, usually a range.
- `store.getAllKeys([query], [count])` – search for all keys that satisfy the query, usually a range, up to `count` if given.
- `store.count([query])` – get the total count of keys that satisfy the query, usually a range.

For instance, we have a lot of books in our store. Remember, the `id` field is the key, so all these methods can search by `id`.

Request examples:

```

// get one book
books.get('js')

// get books with 'css' <= id <= 'html'
books.getAll(IDBKeyRange.bound('css', 'html'))

// get books with id < 'html'
books.getAll(IDBKeyRange.upperBound('html', true))

// get all books
books.getAll()

// get all keys, where id > 'js'
books.getAllKeys(IDBKeyRange.lowerBound('js', true))

```

### **i** Object store is always sorted

An object store sorts values by key internally.

So requests that return many values always return them in sorted by key order.

## By a field using an index

To search by other object fields, we need to create an additional data structure named "index".

An index is an "add-on" to the store that tracks a given object field. For each value of that field, it stores a list of keys for objects that have that value. There will be a more detailed picture below.

The syntax:

```
objectStore.createIndex(name, keyPath, [options]);
```

- **name** – index name,
- **keyPath** – path to the object field that the index should track (we're going to search by that field),
- **option** – an optional object with properties:
  - **unique** – if true, then there may be only one object in the store with the given value at the `keyPath`. The index will enforce that by generating an error if we try to add a duplicate.
  - **multiEntry** – only used if the value on `keyPath` is an array. In that case, by default, the index will treat the whole array as the key. But if `multiEntry` is true, then the index will keep a list of store objects for each value in that array. So array members become index keys.

In our example, we store books keyed by `id`.

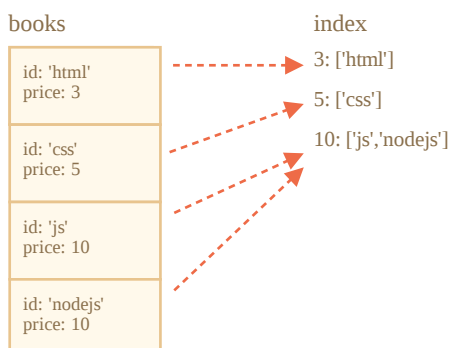
Let's say we want to search by `price`.

First, we need to create an index. It must be done in `upgradeneeded`, just like an object store:

```
openRequest.onupgradeneeded = function() {
 // we must create the index here, in versionchange transaction
 let books = db.createObjectStore('books', {keyPath: 'id'});
 let index = books.createIndex('price_idx', 'price');
};
```

- The index will track `price` field.
- The price is not unique, there may be multiple books with the same price, so we don't set `unique` option.
- The price is not an array, so `multiEntry` flag is not applicable.

Imagine that our `inventory` has 4 books. Here's the picture that shows exactly what the `index` is:



As said, the index for each value of `price` (second argument) keeps the list of keys that have that price.

The index keeps itself up to date automatically, we don't have to care about it.

Now, when we want to search for a given price, we simply apply the same search methods to the index:

```
let transaction = db.transaction("books"); // readonly
let books = transaction.objectStore("books");
```

```
let priceIndex = books.index("price_idx");

let request = priceIndex.getAll(10);

request.onsuccess = function() {
 if (request.result !== undefined) {
 console.log("Books", request.result); // array of books with price=10
 } else {
 console.log("No such books");
 }
};
```

We can also use `IDBKeyRange` to create ranges and looks for cheap/expensive books:

```
// find books where price <= 5
let request = priceIndex.getAll(IDBKeyRange.upperBound(5));
```

Indexes are internally sorted by the tracked object field, `price` in our case. So when we do the search, the results are also sorted by `price`.

## Deleting from store

The `delete` method looks up values to delete by a query, the call format is similar to `getAll`:

- **`delete(query)`** – delete matching values by query.

For instance:

```
// delete the book with id='js'
books.delete('js');
```

If we'd like to delete books based on a price or another object field, then we should first find the key in the index, and then call `delete`:

```
// find the key where price = 5
let request = priceIndex.getKey(5);

request.onsuccess = function() {
 let id = request.result;
 let deleteRequest = books.delete(id);
};
```

To delete everything:

```
books.clear(); // clear the storage.
```

## Cursors

Methods like `getAll/getAllKeys` return an array of keys/values.

But an object storage can be huge, bigger than the available memory. Then `getAll` will fail to get all records as an array.

What to do?

Cursors provide the means to work around that.

**A *cursor* is a special object that traverses the object storage, given a query, and returns one key/value at a time, thus saving memory.**

As an object store is sorted internally by key, a cursor walks the store in key order (ascending by default).

The syntax:

```
// like getAll, but with a cursor:
let request = store.openCursor(query, [direction]);
```

```
// to get keys, not values (like getAllKeys): store.openKeyCursor
```

- **query** is a key or a key range, same as for `getAll`.
- **direction** is an optional argument, which order to use:
  - "next" – the default, the cursor walks up from the record with the lowest key.
  - "prev" – the reverse order: down from the record with the biggest key.
  - "nextunique", "prevunique" – same as above, but skip records with the same key (only for cursors over indexes, e.g. for multiple books with price=5 only the first one will be returned).

The main difference of the cursor is that `request.onsuccess` triggers multiple times: once for each result.

Here's an example of how to use a cursor:

```
let transaction = db.transaction("books");
let books = transaction.objectStore("books");

let request = books.openCursor();

// called for each book found by the cursor
request.onsuccess = function() {
 let cursor = request.result;
 if (cursor) {
 let key = cursor.key; // book key (id field)
 let value = cursor.value; // book object
 console.log(key, value);
 cursor.continue();
 } else {
 console.log("No more books");
 }
};
```

The main cursor methods are:

- `advance(count)` – advance the cursor `count` times, skipping values.
- `continue([key])` – advance the cursor to the next value in range matching (or immediately after `key` if given).

Whether there are more values matching the cursor or not – `onsuccess` gets called, and then in `result` we can get the cursor pointing to the next record, or `undefined`.

In the example above the cursor was made for the object store.

But we also can make a cursor over an index. As we remember, indexes allow to search by an object field. Cursors over indexes do precisely the same as over object stores – they save memory by returning one value at a time.

For cursors over indexes, `cursor.key` is the index key (e.g. price), and we should use `cursor.primaryKey` property for the object key:

```
let request = priceIdx.openCursor(IDBKeyRange.upperBound(5));

// called for each record
request.onsuccess = function() {
 let cursor = request.result;
 if (cursor) {
 let primaryKey = cursor.primaryKey; // next object store key (id field)
 let value = cursor.value; // next object store object (book object)
 let key = cursor.key; // next index key (price)
 console.log(key, value);
 cursor.continue();
 } else {
 console.log("No more books");
 }
};
```

## Promise wrapper

Adding `onsuccess/onerror` to every request is quite a cumbersome task. Sometimes we can make our life easier by using event delegation, e.g. set handlers on the whole transactions, but `async/await` is much more convenient.

Let's use a thin promise wrapper <https://github.com/jakearchibald/idb> further in this chapter. It creates a global `idb` object with `promisified` IndexedDB methods.

Then, instead of `onsuccess/onerror` we can write like this:

```
let db = await idb.openDB('store', 1, db => {
 if (db.oldVersion == 0) {
 // perform the initialization
 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);
}
```

So we have all the sweet “plain async code” and “try...catch” stuff.

### Error handling

If we don't catch an error, then it falls through, till the closest outer `try...catch`.

An uncaught error becomes an “unhandled promise rejection” event on `window` object.

We can handle such errors like this:

```
window.addEventListener('unhandledrejection', event => {
 let request = event.target; // IndexedDB native request object
 let error = event.reason; // Unhandled error object, same as request.error
 ...report about the error...
});
```

### “Inactive transaction” pitfall

As we already know, a transaction auto-commits as soon as the browser is done with the current code and microtasks. So if we put a *macrotask* like `fetch` in the middle of a transaction, then the transaction won't wait for it to finish. It just auto-commits. So the next request in it would fail.

For a promise wrapper and `async/await` the situation is the same.

Here's an example of `fetch` in the middle of the transaction:

```
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
```

The next `inventory.add` after `fetch (*)` fails with an “inactive transaction” error, because the transaction is already committed and closed at that time.

The workaround is the same as when working with native IndexedDB: either make a new transaction or just split things apart.

1. Prepare the data and fetch all that's needed first.
2. Then save in the database.

## Getting native objects

Internally, the wrapper performs a native IndexedDB request, adding `onerror/onsuccess` to it, and returns a promise that rejects/resolves with the result.

That works fine most of the time. The examples are at the lib page <https://github.com/jakearchibald/idb>.

In few rare cases, when we need the original `request` object, we can access it as `promise.request` property of the promise:

```
let promise = books.add(book); // get a promise (don't await for its result)

let request = promise.request; // native request object
let transaction = request.transaction; // native transaction object

// ...do some native IndexedDB voodoo...

let result = await promise; // if still needed
```

## Summary

IndexedDB can be thought of as a “localStorage on steroids”. It’s a simple key-value database, powerful enough for offline apps, yet simple to use.

The best manual is the specification, [the current one](#) is 2.0, but few methods from 3.0 (it’s not much different) are partially supported.

The basic usage can be described with a few phrases:

1. Get a promise wrapper like [idb](#).
2. Open a database: `idb.openDb(name, version, onupgradeneeded)`
  - Create object storages and indexes in `onupgradeneeded` handler or perform version update if needed.
3. For requests:
  - Create transaction `db.transaction('books')` (readwrite if needed).
  - Get the object store `transaction.objectStore('books')`.
4. Then, to search by a key, call methods on the object store directly.
  - To search by an object field, create an index.
5. If the data does not fit in memory, use a cursor.

Here's a small demo app:

<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.

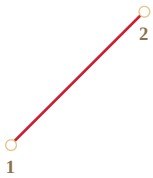
En realidad, son algo muy sencillo, vale la pena estudiarlos una vez y luego sentirse cómodo en el mundo de los gráficos vectoriales y las animaciones avanzadas.

## Puntos de control

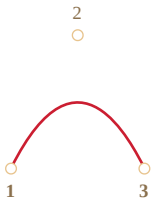
Una [curva de Bézier](#) está definida por puntos de control.

Puede haber 2, 3, 4 o más.

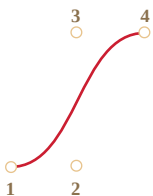
Por ejemplo, curva de dos puntos:



Curva de tres puntos:

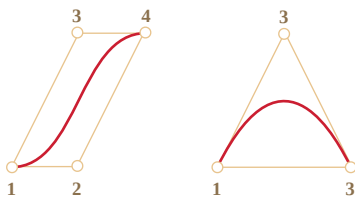


Curva de cuatro puntos:



Si observas detenidamente estas curvas, puedes notar inmediatamente que:

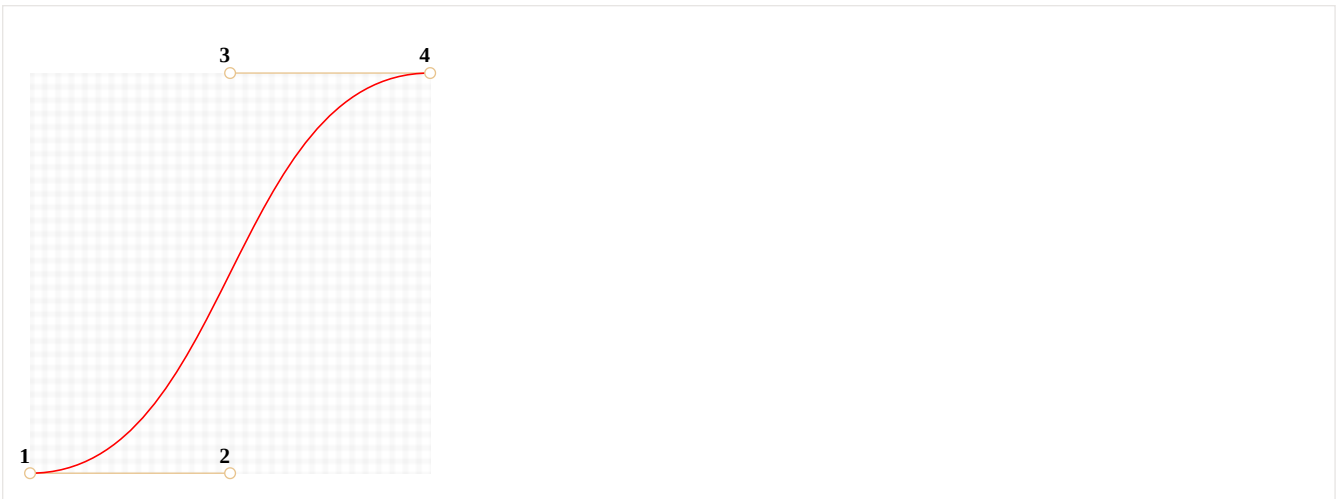
1. **Los puntos no siempre están en la curva.** Eso es perfectamente normal, luego veremos cómo se construye la curva.
2. **El orden de la curva es igual al número de puntos menos uno.** Para dos puntos tenemos una curva lineal (que es una línea recta), para tres puntos – curva cuadrática (parabólica), 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 cascos convexos no se intersecan, las curvas tampoco. Por tanto, comprobar primero la intersección de los cascos convexos puede dar un resultado “sin intersección” muy rápido. La comprobación de la intersección o los cascos 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.**

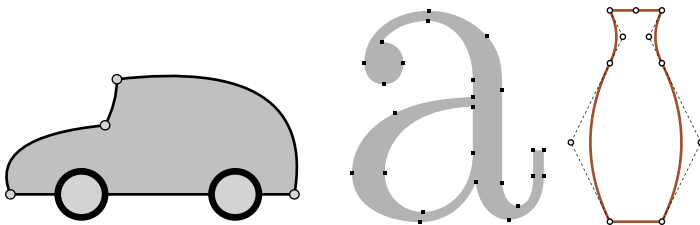
Intenta mover los puntos de control con el ratón en el siguiente ejemplo:



Como puedes observar, la curva se extiende a lo largo de las líneas tangenciales 1 → 2 y 3 → 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:



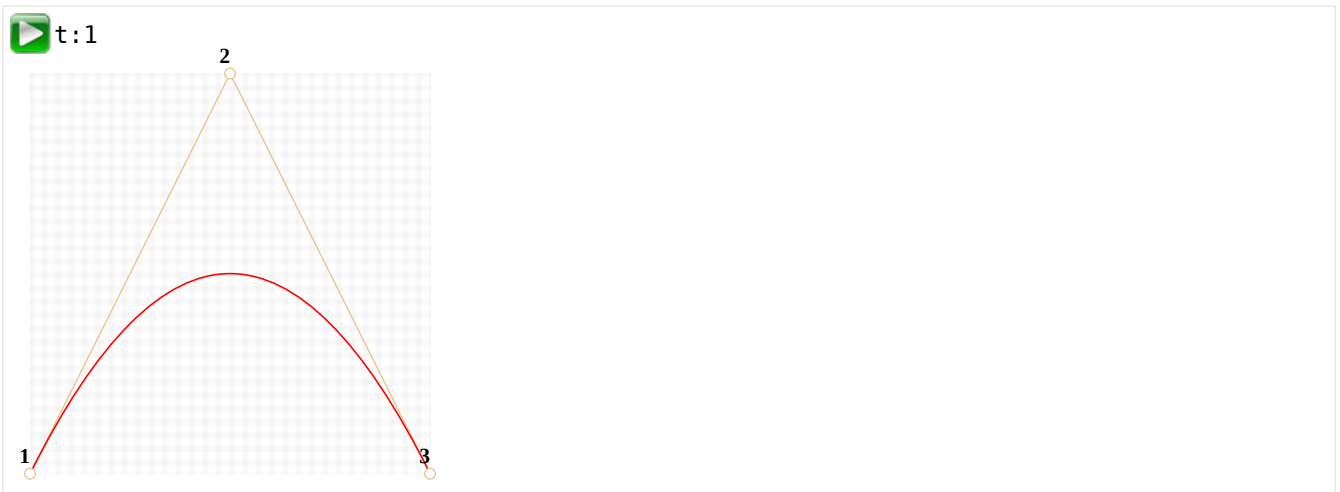
## Algoritmo de De Casteljau

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:**

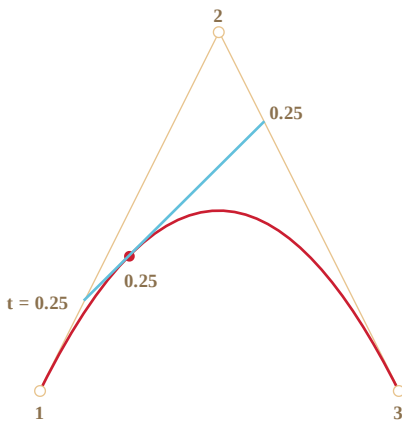
1. Dibujar puntos de control. En la demostración anterior están etiquetados: 1, 2, 3.
2. Construir segmentos entre los puntos de control 1 → 2 → 3. En la demo anterior son marrones.
3. El parámetro  $t$  se mueve de 0 a 1. En el ejemplo de arriba se usa el paso 0.05: el bucle pasa por 0, 0.05, 0.1, 0.15, ... 0.95, 1.

Para cada uno de estos valores de  $t$ :

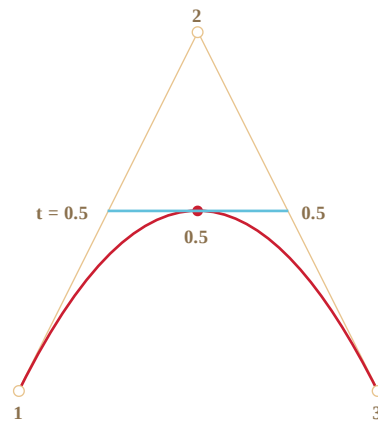


- En cada segmento **marrón** tomamos un punto ubicado en la distancia proporcional a  $t$  desde su comienzo. Como hay dos segmentos, tenemos dos puntos.  
Por ejemplo, para  $t=0$  – ambos puntos estarán al comienzo de los segmentos, y para  $t=0.25$  – en el 25% de la longitud del segmento desde el 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  $t=0.5$



- 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**.
- 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.

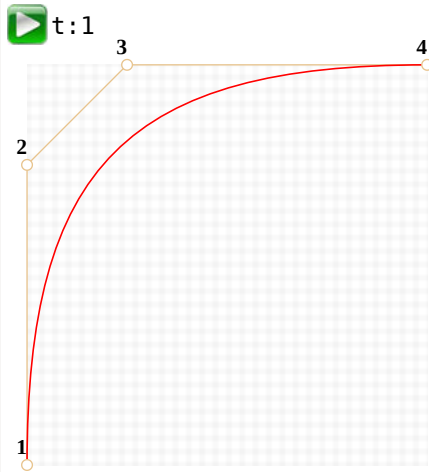
El algoritmo es recursivo y se puede generalizar para cualquier número de puntos de control.

Dados  $N$  de puntos de control:

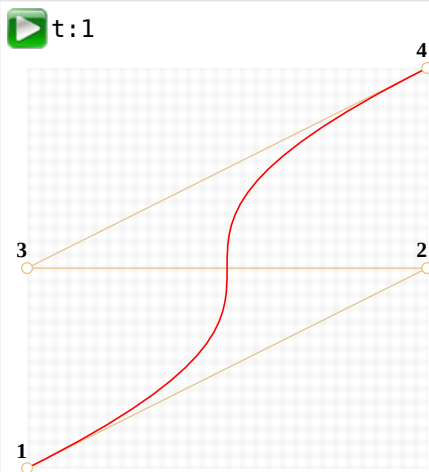
1. Los conectamos para obtener inicialmente N-1 segmentos.
2. Entonces, para cada  $t$  de  $0$  a  $1$ , tomamos un punto en cada segmento en la distancia proporcional a  $t$  y los conectamos. Habrá N-2 segmentos.
3. Repetimos el paso 2 hasta que solo quede un punto.

Estos puntos forman la curva.

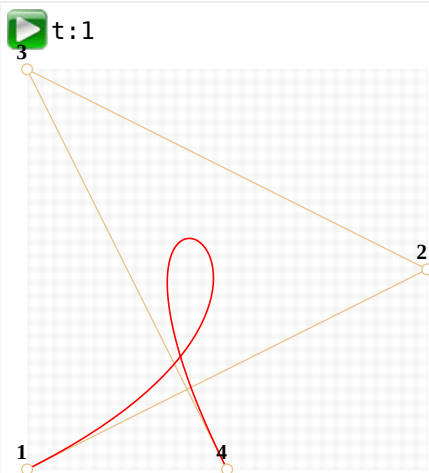
Una curva que se parece a  $y=1/t$  :



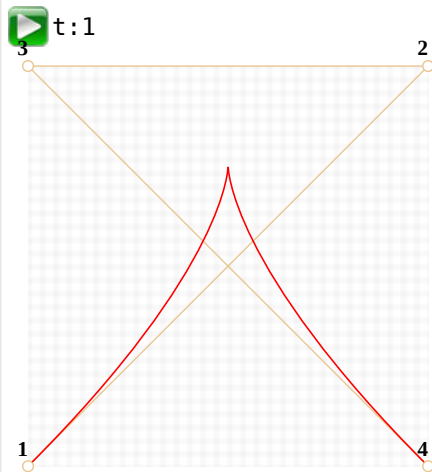
Los puntos de control en zig-zag también funcionan bien:



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.

### ❗ ¿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  $P_1$ : 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  $t$  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 abrirlo 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 mí</button>

<style>
 #color {
 transition-property: background-color;
 transition-duration: 3s;
 }
</style>

<script>
 color.onclick = function() {
 this.style.backgroundColor = 'red';
 };
</script>
```

Haz clic en mi

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 mi</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 mi

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. Entonces 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/3OpQT3blzipKDAp?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/Crob6aqjuQzNVyvd?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.

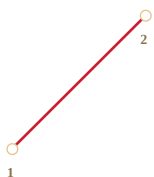
La sintaxis de una curva de Bézier en CSS: `cubic-bezier(x2, y2, x3, y3)`. Aquí necesitamos especificar solo los puntos de control segundo y tercero, porque el primero está fijado a `(0, 0)` y el cuarto es `(1, 1)`.

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/RIjV9tj7atIBrDE2?p=preview>

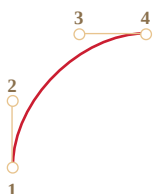
La `transition` de CSS se basa en esa curva:

```
.train {
 left: 0;
 transition: left 5s cubic-bezier(0, 0, 1, 1);
 /* JavaScript establece left a 450px */
}
```

... ¿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/9rIVfsxLkBooVvZI?p=preview>

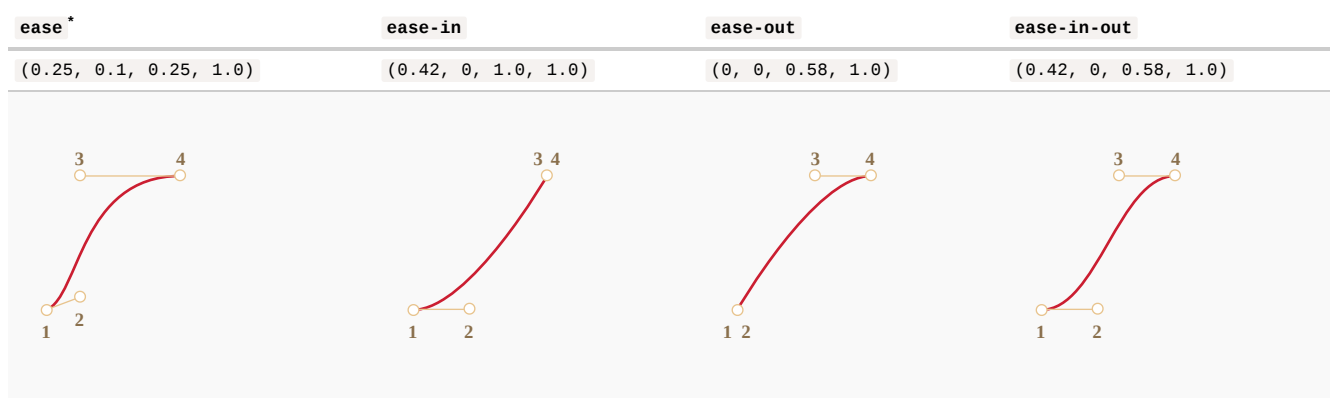
CSS:

```
.train {
 left: 0;
 transition: left 5s cubic-bezier(0, .5, .5, 1);
 /* JavaScript establece left a 450px */
}
```

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`:



\* – 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;
 /* 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);
 /* JavaScript establece left a 400px */
}
```

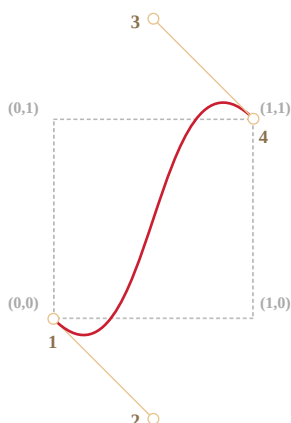
La propiedad `left` debería animarse de `100px` a `400px` .

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 e `y = 1` al valor final. Por lo tanto, los valores `y < 0` mueven la propiedad por debajo del `left` inicial e `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/> .

### Pasos

La función de temporización `steps(number of steps[, start/end])` permite dividir la animación en 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>

Haremos que los dígitos aparezcan de manera discreta haciendo invisible la parte de la lista fuera de la “ventana” roja y desplazando la lista a la izquierda con cada paso.

Habrà 9 pasos, un paso para cada dígito:



```
#stripe.animate {
 transform: translate(-90%);
 transition: transform 9s steps(9, start);
}
```

En acción:

<https://plnkr.co/edit/8YXVGHc6jf0G3FsJ?p=preview>

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.

Podemos observar eso durante la animación: cuando hacemos clic en el dígito, 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` – `-80%`
- (el último segundo muestra el valor final).

El valor alternativo 'end' significaría que el cambio debe aplicarse no al principio, sino al final de cada segundo.

Entonces el proceso sería así:

- `0s` – `0`
- `1s` – `-10%` (primer cambio al final del primer segundo)
- `2s` – `-20%`
- ...
- `9s` – `-90%`

Aquí está el `step(9, end)` en acción (observa la pausa entre el primer cambio de dígitos):

<https://plnkr.co/edit/A9eNaFJ1OXrOmmkd?p=preview>

También hay valores abreviados:

- `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`.

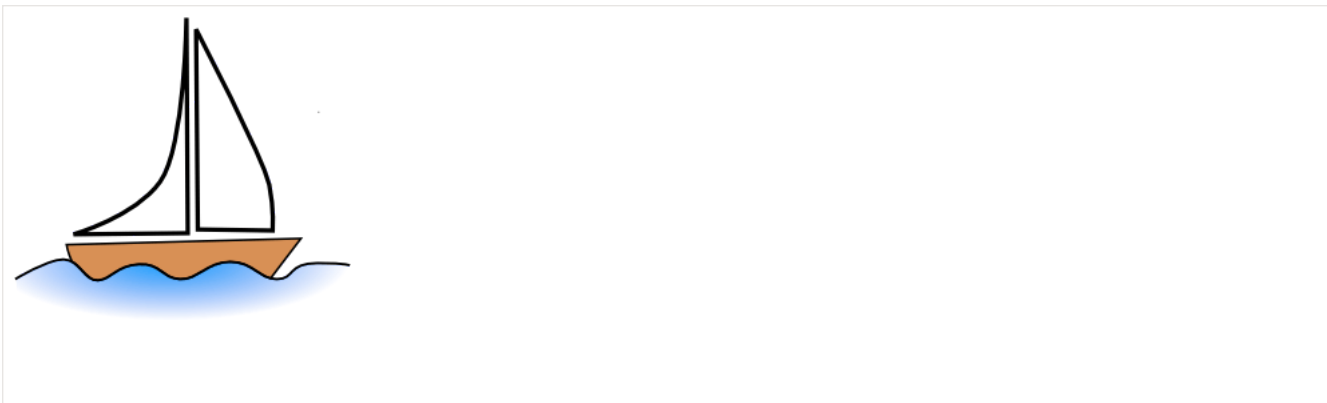
Estos valores rara vez se usan, porque eso no es realmente animación, sino un cambio de un solo paso.

## Evento transitionend

Cuando finaliza la animación CSS, se dispara el evento `transitionend`.

Es ampliamente utilizado para hacer una acción después 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.

## Resumen

Las animaciones CSS permiten animar suavemente (o no) 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.

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**

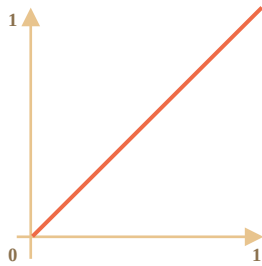
Tiempo total de animación. Como: `1000` .

### **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>

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**

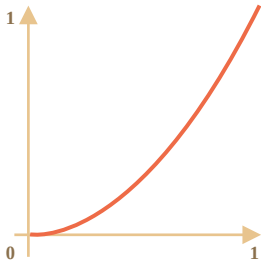
Si queremos acelerar la animación, podemos usar `progress` en la potencia `n`.

Por ejemplo, una curva parabólica:

```
function quad(timeFraction) {
 return Math.pow(timeFraction, 2)
}
```

La gráfica:



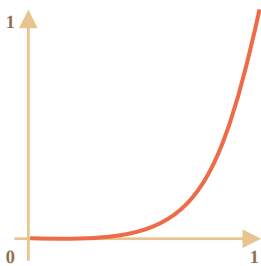


Velo en acción (haz clic para activar):



...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:

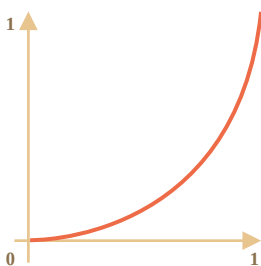


## El arco

Función:

```
function circ(timeFraction) {
 return 1 - Math.sin(Math.acos(timeFraction));
}
```

La gráfica:



## Back: tiro con arco

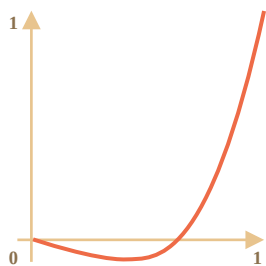
Esta función realiza el “tiro con arco”. Primero “tiramamos de la cuerda del arco”, y luego “disparamos”.

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, result; 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)
 }
 }
}
```

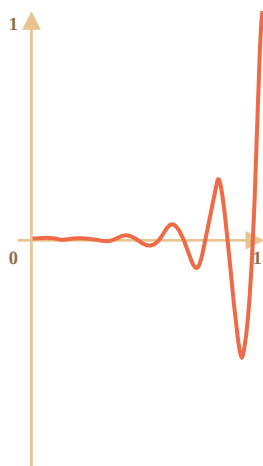
En acción:



### 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);
 }
}
```

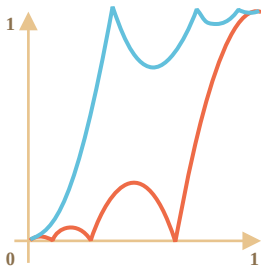
Por ejemplo, podemos tomar la función `bounce` descrita anteriormente y aplicarla:

```
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 “easeInOut”.

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);

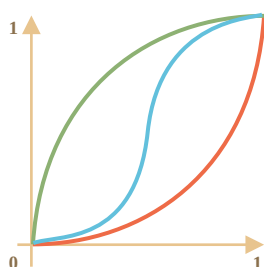
```

En acción, `bounceEaseInOut` :

<https://plnkr.co/edit/Vh0Q3ShC7IOHEyQN?p=preview>

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>

## 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 repintado. 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.

[A solución](#)

## 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):



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.

## Custom elements

We can create custom HTML elements, described by our class, with its own methods and properties, events and so on.

Once a custom element is defined, we can use it on par with built-in HTML elements.

That's great, as HTML dictionary is rich, but not infinite. There are no `<easy-tabs>`, `<sliding-carousel>`, `<beautiful-upload>` ... Just think of any other tag we might need.

We can define them with a special class, and then use as if they were always a part of HTML.

There are two kinds of custom elements:

1. **Autonomous custom elements** – “all-new” elements, extending the abstract `HTMLElement` class.
2. **Customized built-in elements** – extending built-in elements, like a customized button, based on `HTMLButtonElement` etc.

First we'll cover autonomous elements, and then move to customized built-in ones.

To create a custom element, we need to tell the browser several details about it: how to show it, what to do when the element is added or removed to page, etc.

That's done by making a class with special methods. That's easy, as there are only few methods, and all of them are optional.

Here's a sketch with the full list:

```
class MyElement extends HTMLElement {
 constructor() {
 super();
 // element created
 }
}
```



```

connectedCallback() {
 // browser calls this method when the element is added to the document
 // (can be called many times if an element is repeatedly added/removed)
}

disconnectedCallback() {
 // browser calls this method when the element is removed from the document
 // (can be called many times if an element is repeatedly added/removed)
}

static get observedAttributes() {
 return ['* array of attribute names to monitor for changes *'];
}

attributeChangedCallback(name, oldValue, newValue) {
 // called when one of attributes listed above is modified
}

adoptedCallback() {
 // called when the element is moved to a new document
 // (happens in document.adoptNode, very rarely used)
}

// there can be other element methods and properties
}

```

After that, we need to register the element:

```

// let the browser know that <my-element> is served by our new class
customElements.define("my-element", MyElement);

```

Now for any HTML elements with tag `<my-element>`, an instance of `MyElement` is created, and the aforementioned methods are called. We also can `document.createElement('my-element')` in JavaScript.

#### Custom element name must contain a hyphen -

Custom element name must have a hyphen -, e.g. `my-element` and `super-button` are valid names, but `myelement` is not.

That's to ensure that there are no name conflicts between built-in and custom HTML elements.

### Example: “time-formatted”

For example, there already exists `<time>` element in HTML, for date/time. But it doesn't do any formatting by itself.

Let's create `<time-formatted>` element that displays the time in a nice, language-aware format:

```

<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. The class has only one method `connectedCallback()` – the browser calls it when `<time-formatted>` element is added to page (or when HTML parser detects it), and it uses the built-in [Intl.DateTimeFormat](#) data formatter, well-supported across the browsers, to show a nicely formatted time.
2. We need to register our new element by `customElements.define(tag, class)`.
3. And then we can use it everywhere.

### **i Custom elements upgrade**

If the browser encounters any `<time-formatted>` elements before `customElements.define`, that's not an error. But the element is yet unknown, just like any non-standard tag.

Such “undefined” elements can be styled with CSS selector `:not(:defined)`.

When `customElement.define` is called, they are “upgraded”: a new instance of `TimeFormatted` is created for each, and `connectedCallback` is called. They become `:defined`.

To get the information about custom elements, there are methods:

- `customElements.get(name)` – returns the class for a custom element with the given `name`,
- `customElements.whenDefined(name)` – returns a promise that resolves (without value) when a custom element with the given `name` becomes defined.

### **i Rendering in `connectedCallback`, not in `constructor`**

In the example above, element content is rendered (created) in `connectedCallback`.

Why not in the `constructor`?

The reason is simple: when `constructor` is called, it's yet too early. The element is created, but the browser did not yet process/assign attributes at this stage: calls to `getAttribute` would return `null`. So we can't really render there.

Besides, if you think about it, that's better performance-wise – to delay the work until it's really needed.

The `connectedCallback` triggers when the element is added to the document. Not just appended to another element as a child, but actually becomes a part of the page. So we can build detached DOM, create elements and prepare them for later use. They will only be actually rendered when they make it into the page.

## **Observing attributes**

In the current implementation of `<time-formatted>`, after the element is rendered, further attribute changes don't have any effect. That's strange for an HTML element. Usually, when we change an attribute, like `a.href`, we expect the change to be immediately visible. So let's fix this.

We can observe attributes by providing their list in `observedAttributes()` static getter. For such attributes, `attributeChangedCallback` is called when they are modified. It doesn't trigger for other, unlisted attributes (that's for performance reasons).

Here's a new `<time-formatted>`, that auto-updates when attributes change:

```
<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>

```

3:02:28 PM

1. The rendering logic is moved to `render()` helper method.
2. We call it once when the element is inserted into page.
3. For a change of an attribute, listed in `observedAttributes()`, `attributeChangedCallback` triggers.
4. ...and re-renders the element.
5. At the end, we can easily make a live timer.

## Rendering order

When HTML parser builds the DOM, elements are processed one after another, parents before children. E.g. if we have `<outer><inner></inner></outer>`, then `<outer>` element is created and connected to DOM first, and then `<inner>`.

That leads to important consequences for custom elements.

For example, if a custom element tries to access `innerHTML` in `connectedCallback`, it gets nothing:

```

<script>
customElements.define('user-info', class extends HTMLElement {

 connectedCallback() {
 alert(this.innerHTML); // empty (*)
 }

});
</script>

<user-info>John</user-info>

```

If you run it, the `alert` is empty.

That's exactly because there are no children on that stage, the DOM is unfinished. HTML parser connected the custom element `<user-info>`, and is going to proceed to its children, but just didn't yet.

If we'd like to pass information to custom element, we can use attributes. They are available immediately.

Or, if we really need the children, we can defer access to them with zero-delay `setTimeout`.

This works:

```
<script>
customElements.define('user-info', class extends HTMLElement {
 connectedCallback() {
 setTimeout(() => alert(this.innerHTML)); // John (*)
 }
});
</script>

<user-info>John</user-info>
```

Now the `alert` in line `(*)` shows "John", as we run it asynchronously, after the HTML parsing is complete. We can process children if needed and finish the initialization.

On the other hand, this solution is also not perfect. If nested custom elements also use `setTimeout` to initialize themselves, then they queue up: the outer `setTimeout` triggers first, and then the inner one.

So the outer element finishes the initialization before the inner one.

Let's demonstrate that on example:

```
<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>
```

Output order:

1. outer connected.
2. inner connected.
3. outer initialized.
4. inner initialized.

We can clearly see that the outer element finishes initialization `(3)` before the inner one `(4)`.

There's no built-in callback that triggers after nested elements are ready. If needed, we can implement such thing on our own. For instance, inner elements can dispatch events like `initialized`, and outer ones can listen and react on them.

## Customized built-in elements

New elements that we create, such as `<time-formatted>`, don't have any associated semantics. They are unknown to search engines, and accessibility devices can't handle them.

But such things can be important. E.g, a search engine would be interested to know that we actually show a time. And if we're making a special kind of button, why not reuse the existing `<button>` functionality?

We can extend and customize built-in HTML elements by inheriting from their classes.

For example, buttons are instances of `HTMLButtonElement`, let's build upon it.

1. Extend `HTMLButtonElement` with our class:

```
class HelloButton extends HTMLButtonElement { /* custom element methods */ }
```

2. Provide the third argument to `customElements.define`, that specifies the tag:

```
customElements.define('hello-button', HelloButton, {extends: 'button'});
```

There may be different tags that share the same DOM-class, that's why specifying `extends` is needed.

3. At the end, to use our custom element, insert a regular `<button>` tag, but add `is="hello-button"` to it:

```
<button is="hello-button">...</button>
```

Here's a full example:

```
<script>
// The button that says "hello" on click
class HelloButton extends HTMLElement {
 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>
```

Click me Disabled

Our new button extends the built-in one. So it keeps the same styles and standard features like `disabled` attribute.

## References

- HTML Living Standard: <https://html.spec.whatwg.org/#custom-elements> .
- Compatibility: <https://caniuse.com/#feat=custom-elementsv1> .

## Summary

Custom elements can be of two types:

1. “Autonomous” – new tags, extending `HTMLElement` .

Definition scheme:

```
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. “Customized built-in elements” – extensions of existing elements.

Requires one more `.define` argument, and `is="..."` in HTML:

```
class MyButton extends HTMLButtonElement { /*...*/ }
customElements.define('my-button', MyElement, {extends: 'button'});
/* <button is="my-button"> */
```

Custom elements are well-supported among browsers. There's a polyfill

<https://github.com/webcomponents/polyfills/tree/master/packages/webcomponentsjs> .

## ✓ Tareas

### Live timer element

We already have `<time-formatted>` element to show a nicely formatted time.

Create `<live-timer>` element to show the current time:

1. It should use `<time-formatted>` internally, not duplicate its functionality.
2. Ticks (updates) every second.
3. For every tick, a custom event named `tick` should be generated, with the current date in `event.detail` (see chapter [Envío de eventos personalizados](#)).

Usage:

```
<live-timer id="elem"></live-timer>

<script>
 elem.addEventListener('tick', event => console.log(event.detail));
</script>
```

Demo:

3:02:28 PM

[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-runable-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, composition](#).

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](#), [composition](#) veremos cómo se componen.

## Elemento template

El elemento incorporado `<template>` sirve como almacenamiento para plantillas de HTML markup. 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="tpl">
 <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(tmp1.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="tpl">
 <style> p { font-weight: bold; } </style>
 <p id="message"></p>
</template>

<div id="elem">Haz clic sobre mi</div>

<script>
 elem.onclick = function() {
 elem.attachShadow({mode: 'open'});

 elem.shadowRoot.append(tmp1.content.cloneNode(true)); // (*)

 elem.shadowRoot.getElementById('message').innerHTML = "¡Saludos desde las sombras!";
 };
</script>
```

Haz clic sobre mi

En la línea `(*)` , cuando clonamos e insertamos `tmp1.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, composition

Many types of components, such as tabs, menus, image galleries, and so on, need the content to render.

Just like built-in browser `<select>` expects `<option>` items, our `<custom-tabs>` may expect the actual tab content to be passed. And a `<custom-menu>` may expect menu items.

The code that makes use of `<custom-menu>` can look like this:

```
<custom-menu>
 <title>Candy menu</title>
 <item>Lollipop</item>
 <item>Fruit Toast</item>
 <item>Cup Cake</item>
</custom-menu>
```

...Then our component should render it properly, as a nice menu with given title and items, handle menu events, etc.

How to implement it?

We could try to analyze the element content and dynamically copy-rearrange DOM nodes. That's possible, but if we're moving elements to shadow DOM, then CSS styles from the document do not apply in there, so the visual styling may be lost. Also that requires some coding.

Luckily, we don't have to. Shadow DOM supports `<slot>` elements, that are automatically filled by the content from light DOM.

## Named slots

Let's see how slots work on a simple example.

Here, `<user-card>` shadow DOM provides two slots, filled from light DOM:

```
<script>
customElements.define('user-card', class extends HTMLElement {
 connectedCallback() {
 this.attachShadow({mode: 'open'});
 this.shadowRoot.innerHTML = `
 <div>Name:
 <slot name="username"></slot>
 </div>
 <div>Birthday:
 <slot name="birthday"></slot>
 </div>
 `;
 }
});
</script>

<user-card>
 John Smith
 01.01.2001
</user-card>
```

Name: John Smith  
Birthday: 01.01.2001

In the shadow DOM, `<slot name="X">` defines an "insertion point", a place where elements with `slot="X"` are rendered.

Then the browser performs "composition": it takes elements from the light DOM and renders them in corresponding slots of the shadow DOM. At the end, we have exactly what we want – a component that can be filled with data.

Here's the DOM structure after the script, not taking composition into account:

```
<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>
```

We created the shadow DOM, so here it is, under `#shadow-root`. Now the element has both light and shadow DOM.

For rendering purposes, for each `<slot name="...">` in shadow DOM, the browser looks for `slot="..."` with the same name in the light DOM. These elements are rendered inside the 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>
```



The result is called “flattened” DOM:

```
<user-card>
 #shadow-root
 <div>Name:
 <slot name="username">
 <!-- slotted element is inserted into the slot -->
 John Smith
 </slot>
 </div>
 <div>Birthday:
 <slot name="birthday">
 01.01.2001
 </slot>
 </div>
</user-card>
```

...But the flattened DOM exists only for rendering and event-handling purposes. It's kind of “virtual”. That's how things are shown. But the nodes in the document are actually not moved around!

That can be easily checked if we run `querySelectorAll`: nodes are still at their places.

```
// light DOM nodes are still at the same place, under `<user-card>`
alert(document.querySelectorAll('user-card span').length); // 2
```

So, the flattened DOM is derived from shadow DOM by inserting slots. The browser renders it and uses for style inheritance, event propagation (more about that later). But JavaScript still sees the document “as is”, before flattening.

#### ⚠ Only top-level children may have slot="..." attribute

The `slot="..."` attribute is only valid for direct children of the shadow host (in our example, `<user-card>` element). For nested elements it's ignored.

For example, the second `<span>` here is ignored (as it's not a top-level child of `<user-card>`):

```
<user-card>
 John Smith
 <div>
 <!-- invalid slot, must be direct child of user-card -->
 01.01.2001
 </div>
</user-card>
```

If there are multiple elements in light DOM with the same slot name, they are appended into the slot, one after another.

For example, this:

```

<user-card>
 John
 Smith
</user-card>

```

Gives this flattened DOM with two elements in `<slot name="username">`:

```

<user-card>
 #shadow-root
 <div>Name:
 <slot name="username">
 John
 Smith
 </slot>
 </div>
 <div>Birthday:
 <slot name="birthday"></slot>
 </div>
</user-card>

```

## Slot fallback content

If we put something inside a `<slot>`, it becomes the fallback, “default” content. The browser shows it if there’s no corresponding filler in light DOM.

For example, in this piece of shadow DOM, `Anonymous` renders if there’s no `slot="username"` in light DOM.

```

<div>Name:
 <slot name="username">Anonymous</slot>
</div>

```

## Default slot: first unnamed

The first `<slot>` in shadow DOM that doesn’t have a name is a “default” slot. It gets all nodes from the light DOM that aren’t slotted elsewhere.

For example, let’s add the default slot to our `<user-card>` that shows all unslotted information about the user:

```

<script>
customElements.define('user-card', class extends HTMLElement {
 connectedCallback() {
 this.attachShadow({mode: 'open'});
 this.shadowRoot.innerHTML = `
 <div>Name:
 <slot name="username"></slot>
 </div>
 <div>Birthday:
 <slot name="birthday"></slot>
 </div>
 <fieldset>
 <legend>Other information</legend>
 <slot></slot>
 </fieldset>
 `;
 }
});
</script>

<user-card>
 <div>I like to swim.</div>
 John Smith
 01.01.2001
 <div>...And play volleyball too!</div>
</user-card>

```

Name: John Smith  
Birthday: 01.01.2001  
Other information  
I like to swim.  
...And play volleyball too!

All the unslotted light DOM content gets into the “Other information” fieldset.

Elements are appended to a slot one after another, so both unslotted pieces of information are in the default slot together.

The flattened DOM looks like this:

```
<user-card>
 #shadow-root
 <div>Name:
 <slot name="username">
 John Smith
 </slot>
 </div>
 <div>Birthday:
 <slot name="birthday">
 01.01.2001
 </slot>
 </div>
 <fieldset>
 <legend>Other information</legend>
 <slot>
 <div>I like to swim.</div>
 <div>...And play volleyball too!</div>
 </slot>
 </fieldset>
 </user-card>
```

## Menu example

Now let's back to `<custom-menu>`, mentioned at the beginning of the chapter.

We can use slots to distribute elements.

Here's the markup for `<custom-menu>`:

```
<custom-menu>
 Candy menu
 <li slot="item">Lollipop
 <li slot="item">Fruit Toast
 <li slot="item">Cup Cake
</custom-menu>
```

The shadow DOM template with proper slots:

```
<template id="tpl">
 <style> /* menu styles */ </style>
 <div class="menu">
 <slot name="title"></slot>
 <slot name="item"></slot>
 </div>
</template>
```

1. `<span slot="title">` goes into `<slot name="title">`.
2. There are many `<li slot="item">` in the template, but only one `<slot name="item">` in the template. So all such `<li slot="item">` are appended to `<slot name="item">` one after another, thus forming the list.

The flattened DOM becomes:

```
<custom-menu>
 #shadow-root
```

```

<style> /* menu styles */ </style>
<div class="menu">
 <slot name="title">
 Candy menu
 </slot>

 <slot name="item">
 <li slot="item">Lollipop
 <li slot="item">Fruit Toast
 <li slot="item">Cup Cake
 </slot>

</div>
</custom-menu>

```

One might notice that, in a valid DOM, `<li>` must be a direct child of `<ul>`. But that's flattened DOM, it describes how the component is rendered, such thing happens naturally here.

We just need to add a `click` handler to open/close the list, and the `<custom-menu>` is ready:

```

customElements.define('custom-menu', class extends HTMLElement {
 connectedCallback() {
 this.attachShadow({mode: 'open'});

 // tmpl is the shadow DOM template (above)
 this.shadowRoot.append(tmpl.content.cloneNode(true));

 // we can't select light DOM nodes, so let's handle clicks on the slot
 this.shadowRoot.querySelector('slot[name="title"]').onclick = () => {
 // open/close the menu
 this.shadowRoot.querySelector('.menu').classList.toggle('closed');
 };
 }
});

```

Here's the full demo:

```

❏Candy menu
 Lollipop
 Fruit Toast
 Cup Cake

```

Of course, we can add more functionality to it: events, methods and so on.

## Updating slots

What if the outer code wants to add/remove menu items dynamically?

**The browser monitors slots and updates the rendering if slotted elements are added/removed.**

Also, as light DOM nodes are not copied, but just rendered in slots, the changes inside them immediately become visible.

So we don't have to do anything to update rendering. But if the component code wants to know about slot changes, then `slotchange` event is available.

For example, here the menu item is inserted dynamically after 1 second, and the title changes after 2 seconds:

```

<custom-menu id="menu">
 Candy menu
</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 can't have event handlers, so using the first child
this.shadowRoot.firstChild.addEventListener('slotchange',
 e => alert("slotchange: " + e.target.name)
);
}
});

setTimeout(() => {
 menu.insertAdjacentHTML('beforeEnd', '<li slot="item">Lollipop')
}, 1000);

setTimeout(() => {
 menu.querySelector('[slot="title"]').innerHTML = "New menu";
}, 2000);
</script>
```

The menu rendering updates each time without our intervention.

There are two `slotchange` events here:

1. At initialization:

`slotchange: title` triggers immediately, as the `slot="title"` from the light DOM gets into the corresponding slot.

2. After 1 second:

`slotchange: item` triggers, when a new `<li slot="item">` is added.

Please note: there's no `slotchange` event after 2 seconds, when the content of `slot="title"` is modified. That's because there's no slot change. We modify the content inside the slotted element, that's another thing.

If we'd like to track internal modifications of light DOM from JavaScript, that's also possible using a more generic mechanism: [MutationObserver](#).

## Slot API

Finally, let's mention the slot-related JavaScript methods.

As we've seen before, JavaScript looks at the "real" DOM, without flattening. But, if the shadow tree has `{mode: 'open'}`, then we can figure out which elements assigned to a slot and, vise-versa, the slot by the element inside it:

- `node.assignedSlot` – returns the `<slot>` element that the `node` is assigned to.
- `slot.assignedNodes({flatten: true/false})` – DOM nodes, assigned to the slot. The `flatten` option is `false` by default. If explicitly set to `true`, then it looks more deeply into the flattened DOM, returning nested slots in case of nested components and the fallback content if no node assigned.
- `slot.assignedElements({flatten: true/false})` – DOM elements, assigned to the slot (same as above, but only element nodes).

These methods are useful when we need not just show the slotted content, but also track it in JavaScript.

For example, if `<custom-menu>` component wants to know, what it shows, then it could track `slotchange` and get the items from `slot.assignedElements`:

```
<custom-menu id="menu">
 Candy menu
 <li slot="item">Lollipop
 <li slot="item">Fruit Toast
</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>`;

 // triggers when slot content changes
```



```

 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);
 }
 });
 }
});

// items update after 1 second
setTimeout(() => {
 menu.insertAdjacentHTML('beforeEnd', '<li slot="item">Cup Cake')
}, 1000);
</script>

```

## Summary

Usually, if an element has shadow DOM, then its light DOM is not displayed. Slots allow to show elements from light DOM in specified places of shadow DOM.

There are two kinds of slots:

- Named slots: `<slot name="X">...</slot>` – gets light children with `slot="X"`.
- Default slot: the first `<slot>` without a name (subsequent unnamed slots are ignored) – gets unslotted light children.
- If there are many elements for the same slot – they are appended one after another.
- The content of `<slot>` element is used as a fallback. It's shown if there are no light children for the slot.

The process of rendering slotted elements inside their slots is called “composition”. The result is called a “flattened DOM”.

Composition does not really move nodes, from JavaScript point of view the DOM is still same.

JavaScript can access slots using methods:

- `slot.assignedNodes/Elements()` – returns nodes/elements inside the `slot`.
- `node.assignedSlot` – the reverse property, returns slot by a node.

If we'd like to know what we're showing, we can track slot contents using:

- `slotchange` event – triggers the first time a slot is filled, and on any add/remove/replace operation of the slotted element, but not its children. The slot is `event.target`.
- [MutationObserver](#) to go deeper into slot content, watch changes inside it.

Now, as we know how to show elements from light DOM in shadow DOM, let's see how to style them properly. The basic rule is that shadow elements are styled inside, and light elements – outside, but there are notable exceptions.

We'll see the details in the next chapter.

## 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="tpl">
 <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>

```

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 sobrescribirlos 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="tmp1">
 <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(tmp1.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>`.

## :host-context(selector)

Igual que `:host`, pero se aplica solo si el shadow host o cualquiera de sus ancestros en el documento exterior coinciden con el selector.

p. ej. `:host-context(.dark-theme)` coincide solo si hay una clase `dark-theme` en `<custom-dialog>` en cualquier lugar por encima de él:

```

<body class="dark-theme">
 <!--
 :host-context(.dark-theme) se aplica a los custom-dialogs dentro de .dark-theme
 -->
 <custom-dialog>...</custom-dialog>
</body>

```

Para resumir, podemos usar `:host`-familia de selectores para aplicar estilos al elemento principal del componente, según el contexto. Estos estilos (a menos que sea `!important`) pueden ser sobrescritos 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 esta 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="tpl">
 <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>
```

```

</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('tpl').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 `:host` -familia de pseudoclases,
- 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 and events

The idea behind shadow tree is to encapsulate internal implementation details of a component.

Let's say, a click event happens inside a shadow DOM of `<user-card>` component. But scripts in the main document have no idea about the shadow DOM internals, especially if the component comes from a 3rd-party library.

So, to keep the details encapsulated, the browser *retargets* the event.

**Events that happen in shadow DOM have the host element as the target, when caught outside of the component.**

Here's a simple example:

```

<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.firstElementChild.onclick =
 e => alert("Inner target: " + e.target.tagName);
 }
});

document.onclick =
 e => alert("Outer target: " + e.target.tagName);
</script>

```

Click me

If you click on the button, the messages are:

1. Inner target: `BUTTON` – internal event handler gets the correct target, the element inside shadow DOM.
2. Outer target: `USER-CARD` – document event handler gets shadow host as the target.

Event retargeting is a great thing to have, because the outer document doesn't have to know about component internals. From its point of view, the event happened on `<user-card>`.

**Retargeting does not occur if the event occurs on a slotted element, that physically lives in the light DOM.**

For example, if a user clicks on `<span slot="username">` in the example below, the event target is exactly this `span` element, for both shadow and light handlers:

```

<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.firstElementChild.onclick =
 e => alert("Inner target: " + e.target.tagName);
 }
});

userCard.onclick = e => alert(`Outer target: ${e.target.tagName}`);
</script>

```

Name: John Smith

If a click happens on `"John Smith"`, for both inner and outer handlers the target is `<span slot="username">`. That's an element from the light DOM, so no retargeting.

On the other hand, if the click occurs on an element originating from shadow DOM, e.g. on `<b>Name</b>`, then, as it bubbles out of the shadow DOM, its `event.target` is reset to `<user-card>`.

## Bubbling, `event.composedPath()`

For purposes of event bubbling, flattened DOM is used.

So, if we have a slotted element, and an event occurs somewhere inside it, then it bubbles up to the `<slot>` and upwards.

The full path to the original event target, with all the shadow elements, can be obtained using `event.composedPath()`. As we can see from the name of the method, that path is taken after the composition.

In the example above, the flattened DOM is:

```

<user-card id="userCard">
 #shadow-root
 <div>
 Name:
 <slot name="username">
 John Smith
 </slot>
 </div>
 </user-card>

```

So, for a click on `<span slot="username">`, a call to `event.composedPath()` returns an array: `[span, slot, div, shadow-root, user-card, body, html, document, window]`. That's exactly the parent chain from the target element in the flattened DOM, after the composition.

#### ⚠ Shadow tree details are only provided for `{mode: 'open'}` trees

If the shadow tree was created with `{mode: 'closed'}`, then the composed path starts from the host: `user-card` and upwards.

That's the similar principle as for other methods that work with shadow DOM. Internals of closed trees are completely hidden.

## event.composed

Most events successfully bubble through a shadow DOM boundary. There are few events that do not.

This is governed by the `composed` event object property. If it's `true`, then the event does cross the boundary. Otherwise, it only can be caught from inside the shadow DOM.

If you take a look at [UI Events specification](#), most events have `composed: true`:

- `blur`, `focus`, `focusin`, `focusout`,
- `click`, `dblclick`,
- `mousedown`, `mouseup`, `mousemove`, `mouseout`, `mouseover`,
- `wheel`,
- `beforeinput`, `input`, `keydown`, `keyup`.

All touch events and pointer events also have `composed: true`.

There are some events that have `composed: false` though:

- `mouseenter`, `mouseleave` (they do not bubble at all),
- `load`, `unload`, `abort`, `error`,
- `select`,
- `slotchange`.

These events can be caught only on elements within the same DOM, where the event target resides.

## Custom events

When we dispatch custom events, we need to set both `bubbles` and `composed` properties to `true` for it to bubble up and out of the component.

For example, here we create `div#inner` in the shadow DOM of `div#outer` and trigger two events on it. Only the one with `composed: true` makes it outside to the document:

```

<div id="outer"></div>

<script>
 outer.attachShadow({mode: 'open'});

 let inner = document.createElement('div');
 outer.shadowRoot.append(inner);

 /*
 div(id=outer)
 #shadow-dom

```



```



```

## Summary

Events only cross shadow DOM boundaries if their `composed` flag is set to `true`.

Built-in events mostly have `composed: true`, as described in the relevant specifications:

- UI Events <https://www.w3.org/TR/uievents>.
- Touch Events <https://w3c.github.io/touch-events>.
- Pointer Events <https://www.w3.org/TR/pointerevents>.
- ...And so on.

Some built-in events that have `composed: false`:

- `mouseenter`, `mouseleave` (also do not bubble),
- `load`, `unload`, `abort`, `error`,
- `select`,
- `slotchange`.

These events can be caught only on elements within the same DOM.

If we dispatch a `CustomEvent`, then we should explicitly set `composed: true`.

Please note that in case of nested components, one shadow DOM may be nested into another. In that case composed events bubble through all shadow DOM boundaries. So, if an event is intended only for the immediate enclosing component, we can also dispatch it on the shadow host and set `composed: false`. Then it's out of the component shadow DOM, but won't bubble up to higher-level DOM.

## 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 “regex”, 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”:

```

regex = 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 multilinea (cubierto en el capítulo [Modo multilinea 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(regex, replacement)` reemplaza las coincidencias encontradas usando `regex` 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
----------------------	------------------------------

Símbolos	Acción en la cadena de reemplazo
\$`	inserta una parte de la cadena antes de la coincidencia
\$'	inserta una parte de la cadena después de la coincidencia
\$n	si n es un número de 1-2 dígitos, entonces inserta el contenido de los paréntesis n-ésimo, más del tema en el capítulo <a href="#">Grupos de captura</a>
\$<name>	inserta el contenido de los paréntesis con el nombre dado, más del tema en el capítulo <a href="#">Grupos de captura</a>
\$\$	inserta el carácter \$

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, exploremos 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 guión bajo `_`. Las letras no latinas (como el cirílico o el hindi) 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\w\d/)); // ' HTML5'
```

La coincidencia (cada clase de carácter de la expresión regular tiene el carácter resultante correspondiente):

```

 \s \w \w \w \w \d
I l o v e H T M L 5
```

## 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, guión 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  $\chi$  (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
$\chi$	0x1d4b3	4
ℓ	0x1d4b4	4
😊	0x1f604	4

Entonces los caracteres como `a` e `≈` ocupan 2 bytes, mientras que los códigos para `χ`, `ℓ` 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 Ⴓ Ⴕ";

alert(str.match(/\p{L}/gu)); // A, Ⴓ, Ⴕ
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 (sustituido) `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) (`Alfa`), incluye letras `L` , más números de letras `Nl` (por ejemplo, `Xl` – un carácter para el número romano 12), y otros símbolos `Other_Alphabetic` (`0Alpha`).
- `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> .
- 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, como `xFF` , donde `F` es un dígito hexadecimal (`0...1` 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 esta 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/fin 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`;

alert(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`;

alert(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`;

alert(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 multilinea 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`.

¿Cual es la diferencia? Veamos un ejemplo.

Buscamos `\d\n` en lugar de `\d$`:

```
let str = `Winnie: 1
Piglet: 2
Eeyore: 3`;

alert(str.match(/\\d\\n/gm)); // 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 `pattern:$`.

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:

↓ ↓ ↓ ↓  
H e l l o , J a v a !

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(/bHel\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/fin).

```
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 guión 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. Se utilizan para hacer búsquedas más potentes. Aquí hay una lista completa de ellos: `[ \ ^ $ . | ? * + ( ) ]`.

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 escaparlos.

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 escaparlos:

```
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(regex)); // 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\.`”:

```
alert("\d\.\d"); // d.d
```

Las comillas de cadenas “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 dados”.

### 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 guión -, entonces el conjunto es [\w-].

También es posible combinar varias clases, p.ej.: [\s\d] significa "un carácter de espacio o un dígito".

**i 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}].

Descifrémoslo. 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 guión 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 compatibles con 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. `[а-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]` – cualquier 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 guión `-` 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 `xy`:

```
alert('xy'.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[<sup>^</sup>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.

## 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(/<\/?[a-z][a-z0-9]*>/gi)); // <h1>, </h1>
```

**i 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 ser `«Hola, mundo»`. Existen otras comillas, como `„Witam, świat!"` (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 `/" .+"/g` (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(regex)); // "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:

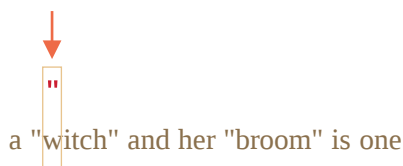
- Para cada posición en la cadena
  - Prueba si el patrón coincide en esta posición.
  - Si no hay coincidencia, ir a la siguiente posición.

Estas palabras comunes no son tan obvias para determinar por qué la regexp falla, así que elaboremos 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.

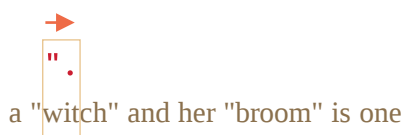
Entonces avanza: va a la siguiente posición en la cadena fuente y prueba encontrar el primer carácter del patrón allí, falla de nuevo, y finalmente encuentra la comilla doble en la 3ra posición:



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 “cualquiere 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:



a "witch" and her "broom" is one

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 `'\"'`) coincida:



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 esperabamos, 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: "repite el mínimo número de veces".

Podemos habilitarlo poniendo un signo de interrogación `'?'` después del cuantificador, entonces tendríamos `*?` o `+?` o incluso `??` para `'?'`.

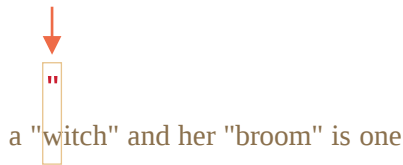
Aclarando las cosas: generalmente un signo de interrogación `'?'` es un cuantificador por si mismo (cero o uno), pero si se agrega *después de otro cuantificador (o incluso el mismo)* toma otro significado, alterna el modo de coincidencia de codicioso a perezoso.

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:



a "witch" and her "broom" is one

2. El siguiente paso también es similar: el motor encuentra una coincidencia para el punto `'.'`:



a "witch" and her "broom" is one

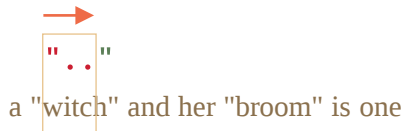
3. Y ahora la búsqueda es diferente. Porque tenemos el modo perezoso activado en `+?`, el motor no prueba coincidir un punto una vez más, se detiene y prueba coincidir el resto del patrón (`'\"'`) ahora mismo :



a "witch" and her "broom" is one

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:



a "witch" and her "broom" is one

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:



a "witch" and her "broom" is one

6. La próxima búsqueda inicia desde el final de la coincidencia actual y produce un resultado más:



a "witch" and her "broom" is one

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
```



1. 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 `' '`.
2. Luego hay un espacio en el patrón, coincide.
3. 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 `1234`.

#### **i 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`.

¿Cual expresión regular usamos?

La primera idea podría ser: `/<a href=".*" class="doc">/g`.

Veámoslo:

```
let str = '......';
let regexp = //g;

// ¡Funciona!
alert(str.match(regexp)); //
```

Funcionó. Pero veamos ¿que pasa si hay varios enlaces en el texto?

```
let str = '...... ...';
let regexp = //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="[^"]*" class="doc"`. 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:

### Codiciosa

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+?/`

¿Cual 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 `gooooooooo`.

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, no se permite guiones y puntos. 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):

```
1 [span class="my"]
<(([a-z]+)\s*([>]*))>
2 [span] 3 [class="my"]
```

En acción:

```
let str = '';

let regexp = /<(([a-z]+)\s*([>]*))>/;

let result = str.match(regexp);
alert(result[0]); //
alert(result[1]); // span class="my"
alert(result[2]); // span
alert(result[3]); // class="my"
```

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:

```
1 [span class="my"]
<(([a-z]+)\s*([>]*))>
2 [span] 3 [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/Ijharb/String.prototype.matchAll>.

Cuando buscamos todas las coincidencias (flag `g`), el método `match` no devuelve contenido para los grupos.

Por ejemplo, encontremos 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(regex)`.

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 convirtamos 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>
```

### ¿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 dateRegex = /(?!<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(dateRegex);

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(regex, replacement)` que reemplaza todas las coincidencias con `regex` 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 regex = /(\w+) (\w+)/;

alert(str.replace(regex, '$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 regex = /(?!<year>[0-9]{4})-(?!<month>[0-9]{2})-(?!<day>[0-9]{2})/g;

let str = "2019-10-30, 2020-01-01";

alert(str.replace(regex, '$<day>.$<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 regex = /(?:go)+ (\w+)/i;

let result = str.match(regex);

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 disponible 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(regex)); // "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 regex = /(["'])(.*?)\1/g;

alert(str.match(regex)); // "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.

#### 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 regex = /(?<quote>["'])(.*?)\k<quote>/g;

alert(str.match(regex)); // "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 regex = /html|php|css|java(script)?/gi;
let str = "Primera aparición de HTML, luego CSS, luego JavaScript";

alert(str.match(regex)); // '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`.
- `gr|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 que 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] [/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 doble barra invertida 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 pasarlas 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 and lookbehind

Sometimes we need to find only those matches for a pattern that are followed or preceded by another pattern.

There's a special syntax for that, called "lookahead" and "lookbehind", together referred to as "lookaround".

For the start, let's find the price from the string like `1 turkey costs 30€`. That is: a number, followed by `€` sign.

### Lookahead

The syntax is: `X(?=Y)`, it means "look for `X`, but match only if followed by `Y`". There may be any pattern instead of `X` and `Y`.

For an integer number followed by `€`, the regexp will be `\d+(?=.*€)`:

```
let str = "1 turkey costs 30€";

alert(str.match(/\d+(?=.*€)/)); // 30, the number 1 is ignored, as it's not followed by €
```

Please note: the lookahead is merely a test, the contents of the parentheses `(?=.*€)` is not included in the result `30`.

When we look for `X(?=Y)`, the regular expression engine finds `X` and then checks if there's `Y` immediately after it. If it's not so, then the potential match is skipped, and the search continues.

More complex tests are possible, e.g. `X(?=Y)(?=Z)` means:

1. Find `X`.
2. Check if `Y` is immediately after `X` (skip if isn't).
3. Check if `Z` is also immediately after `X` (skip if isn't).
4. If both tests passed, then the `X` is a match, otherwise continue searching.

In other words, such pattern means that we're looking for `X` followed by `Y` and `Z` at the same time.

That's only possible if patterns `Y` and `Z` aren't mutually exclusive.

For example, `\d+(?=\s)(?=. *30)` looks for `\d+` that is followed by a space `(?=\s)`, and there's `30` somewhere after it `(?=. *30)`:

```
let str = "1 turkey costs 30€";

alert(str.match(/\d+(?=\s)(?=. *30)/)); // 1
```

In our string that exactly matches the number `1`.

## Negative lookahead

Let's say that we want a quantity instead, not a price from the same string. That's a number `\d+`, NOT followed by `€`.

For that, a negative lookahead can be applied.

The syntax is: `X(?!Y)`, it means "search `X`, but only if not followed by `Y`".

```
let str = "2 turkeys cost 60€";

alert(str.match(/\d+\b(?!€)/g)); // 2 (the price is not matched)
```

## Lookbehind

Lookahead allows to add a condition for "what follows".

Lookbehind is similar, but it looks behind. That is, it allows to match a pattern only if there's something before it.

The syntax is:

- Positive lookbehind: `(?<=Y)X`, matches `X`, but only if there's `Y` before it.
- Negative lookbehind: `(?<!\Y)X`, matches `X`, but only if there's no `Y` before it.

For example, let's change the price to US dollars. The dollar sign is usually before the number, so to look for `$30` we'll use `(?<=\$)\d+` – an amount preceded by `$`:

```
let str = "1 turkey costs $30";

// the dollar sign is escaped \$
alert(str.match(/\(?<=\$)\d+/)); // 30 (skipped the sole number)
```

And, if we need the quantity – a number, not preceded by `$`, then we can use a negative lookbehind `(?<!\$)\d+`:

```
let str = "2 turkeys cost $60";

alert(str.match(/\(?<!\$)\b\d+/g)); // 2 (the price is not matched)
```

## Capturing groups

Generally, the contents inside lookahead parentheses does not become a part of the result.

E.g. in the pattern `\d+(?=€)`, the `€` sign doesn't get captured as a part of the match. That's natural: we look for a number `\d+`, while `(?=€)` is just a test that it should be followed by `€`.

But in some situations we might want to capture the lookahead expression as well, or a part of it. That's possible. Just wrap that part into additional parentheses.

In the example below the currency sign `(€|kr)` is captured, along with the amount:

```
let str = "1 turkey costs 30€";
let regexp = /\d+(?=(€|kr))/; // extra parentheses around €|kr

alert(str.match(regexp)); // 30, €
```

And here's the same for lookbehind:

```
let str = "1 turkey costs $30";
let regexp = /(?!<=($|£))\d+/;

alert(str.match(regexp)); // 30, $
```

## Summary

Lookahead and lookbehind (commonly referred to as "lookaround") are useful when we'd like to match something depending on the context before/after it.

For simple regexps we can do the similar thing manually. That is: match everything, in any context, and then filter by context in the loop.

Remember, `str.match` (without flag `g`) and `str.matchAll` (always) return matches as arrays with `index` property, so we know where exactly in the text it is, and can check the context.

But generally lookaround is more convenient.

Lookaround types:

Pattern	type	matches
<code>X(?:=Y)</code>	Positive lookahead	<u>X</u> if followed by <u>Y</u>
<code>X(?:!Y)</code>	Negative lookahead	<u>X</u> if not followed by <u>Y</u>
<code>(?&lt;=Y)X</code>	Positive lookbehind	<u>X</u> if after <u>Y</u>
<code>(?&lt;!Y)X</code>	Negative lookbehind	<u>X</u> if not after <u>Y</u>

## ✔ Tareas

### Find non-negative integers

There's a string of integer numbers.

Create a regexp that looks for only non-negative ones (zero is allowed).

An example of use:

```
let regexp = /your regexp/g;

let str = "0 12 -5 123 -18";

alert(str.match(regexp)); // 0, 12, 123
```

[A solución](#)

### Insert After Head

We have a string with an HTML Document.

Write a regular expression that inserts `<h1>Hello</h1>` immediately after `<body>` tag. The tag may have attributes.

For instance:

```
let regexp = /your regular expression/;

let str = `
<html>
 <body style="height: 200px">
 ...
 </body>
</html>
`;

str = str.replace(regexp, `<h1>Hello</h1>`);
```



After that the value of `str` should be:

```
<html>
 <body style="height: 200px"><h1>Hello</h1>
 ...
</body>
</html>
```

[A solución](#)

## Catastrophic backtracking

Some regular expressions are looking simple, but can execute a veeeeery long time, and even “hang” the JavaScript engine.

Sooner or later most developers occasionally face such behavior. The typical symptom – a regular expression works fine sometimes, but for certain strings it “hangs”, consuming 100% of CPU.

In such case a web-browser suggests to kill the script and reload the page. Not a good thing for sure.

For server-side JavaScript such a regexp may hang the server process, that’s even worse. So we definitely should take a look at it.

### Example

Let’s say we have a string, and we’d like to check if it consists of words `\w+` with an optional space `\s?` after each.

An obvious way to construct a regexp would be to take a word followed by an optional space `\w+\s?` and then repeat it with `*`.

That leads us to the regexp `^(\w+\s?)*$`, it specifies zero or more such words, that start at the beginning `^` and finish at the end `$` of the line.

In action:

```
let regexp = /^(\w+\s?)*$/;

alert(regexp.test("A good string")); // true
alert(regexp.test("Bad characters: $@#")); // false
```

The regexp seems to work. The result is correct. Although, on certain strings it takes a lot of time. So long that JavaScript engine “hangs” with 100% CPU consumption.

If you run the example below, you probably won’t see anything, as JavaScript will just “hang”. A web-browser will stop reacting on events, the UI will stop working (most browsers allow only scrolling). After some time it will suggest to reload the page. So be careful with this:

```
let regexp = /^(\w+\s?)*$/;
let str = "An input string that takes a long time or even makes this regexp hang!";

// will take a very long time
alert(regexp.test(str));
```

To be fair, let’s note that some regular expression engines can handle such a search effectively, for example V8 engine version starting from 8.8 can do that (so Google Chrome 88 doesn’t hang here), while Firefox browser does hang.

### Simplified example

What’s the matter? Why does the regular expression hang?

To understand that, let’s simplify the example: remove spaces `\s?`. Then it becomes `^(\w+)*$`.

And, to make things more obvious, let’s replace `\w` with `\d`. The resulting regular expression still hangs, for instance:

```
let regexp = /^(\d+)*$/;

let str = "012345678901234567890123456789z";

// will take a very long time (careful!)
alert(regexp.test(str));
```

So what's wrong with the regexp?

First, one may notice that the regexp `(\d+)*` is a little bit strange. The quantifier `*` looks extraneous. If we want a number, we can use `\d+`.

Indeed, the regexp is artificial; we got it by simplifying the previous example. But the reason why it is slow is the same. So let's understand it, and then the previous example will become obvious.

What happens during the search of `^(\d+)*$` in the line `123456789z` (shortened a bit for clarity, please note a non-digit character `z` at the end, it's important), why does it take so long?

Here's what the regexp engine does:

1. First, the regexp engine tries to find the content of the parentheses: the number `\d+`. The plus `+` is greedy by default, so it consumes all digits:

```
\d+.....
(123456789)z
```

After all digits are consumed, `\d+` is considered found (as `123456789`).

Then the star quantifier `(\d+)*` applies. But there are no more digits in the text, so the star doesn't give anything.

The next character in the pattern is the string end `$`. But in the text we have `z` instead, so there's no match:

```
 X
\d+.....$
(123456789)z
```

2. As there's no match, the greedy quantifier `+` decreases the count of repetitions, backtracks one character back.

Now `\d+` takes all digits except the last one (`12345678`):

```
\d+.....
(12345678)9z
```

3. Then the engine tries to continue the search from the next position (right after `12345678`).

The star `(\d+)*` can be applied – it gives one more match of `\d+`, the number `9`:

```
\d+.....\d+
(12345678)(9)z
```

The engine tries to match `$` again, but fails, because it meets `z` instead:

```
 X
\d+.....\d+
(12345678)(9)z
```

4. There's no match, so the engine will continue backtracking, decreasing the number of repetitions. Backtracking generally works like this: the last greedy quantifier decreases the number of repetitions until it reaches the minimum. Then the previous greedy quantifier decreases, and so on.

All possible combinations are attempted. Here are their examples.

The first number `\d+` has 7 digits, and then a number of 2 digits:

```
 X
\d+.....\d+
(1234567)(89)z
```

The first number has 7 digits, and then two numbers of 1 digit each:

```
 X
\d+.....\d+\d+
(1234567)(8)(9)z
```

The first number has 6 digits, and then a number of 3 digits:

```
 X
\d+.....\d+
(123456)(789)z
```

The first number has 6 digits, and then 2 numbers:

```
 X
\d+.....\d+ \d+
(123456)(78)(9)z
```

...And so on.

There are many ways to split a sequence of digits `123456789` into numbers. To be precise, there are  $2^n - 1$ , where  $n$  is the length of the sequence.

- For `123456789` we have  $n=9$ , that gives 511 combinations.
- For a longer sequence with  $n=20$  there are about one million (1048575) combinations.
- For  $n=30$  – a thousand times more (1073741823 combinations).

Trying each of them is exactly the reason why the search takes so long.

## Back to words and strings

The similar thing happens in our first example, when we look for words by pattern `^(\w+\s?)*$` in the string `An input that hangs!`.

The reason is that a word can be represented as one `\w+` or many:

```
(input)
(inpu)(t)
(inp)(u)(t)
(in)(p)(ut)
...
```

For a human, it's obvious that there may be no match, because the string ends with an exclamation sign `!`, but the regular expression expects a wordly character `\w` or a space `\s` at the end. But the engine doesn't know that.

It tries all combinations of how the regexp `(\w+\s?)*` can “consume” the string, including variants with spaces `(\w+\s)*` and without them `(\w+)*` (because spaces `\s?` are optional). As there are many such combinations (we've seen it with digits), the search takes a lot of time.

What to do?

Should we turn on the lazy mode?

Unfortunately, that won't help: if we replace `\w+` with `\w+?`, the regexp will still hang. The order of combinations will change, but not their total count.

Some regular expression engines have tricky tests and finite automations that allow to avoid going through all combinations or make it much faster, but most engines don't, and it doesn't always help.

## How to fix?

There are two main approaches to fixing the problem.

The first is to lower the number of possible combinations.

Let's make the space non-optional by rewriting the regular expression as `^(\w+\s)*\w*$` – we'll look for any number of words followed by a space `(\w+\s)*`, and then (optionally) a final word `\w*`.

This regexp is equivalent to the previous one (matches the same) and works well:

```
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
```

Why did the problem disappear?

That's because now the space is mandatory.

The previous regexp, if we omit the space, becomes `(\w+)*`, leading to many combinations of `\w+` within a single word

So input could be matched as two repetitions of `\w+`, like this:

```
\w+ \w+
(inp)(ut)
```

The new pattern is different: `(\w+\s)*` specifies repetitions of words followed by a space! The input string can't be matched as two repetitions of `\w+\s`, because the space is mandatory.

The time needed to try a lot of (actually most of) combinations is now saved.

## Preventing backtracking

It's not always convenient to rewrite a regexp though. In the example above it was easy, but it's not always obvious how to do it.

Besides, a rewritten regexp is usually more complex, and that's not good. Regexp's are complex enough without extra efforts.

Luckily, there's an alternative approach. We can forbid backtracking for the quantifier.

The root of the problem is that the regexp engine tries many combinations that are obviously wrong for a human.

E.g. in the regexp `(\d+)*$` it's obvious for a human, that + shouldn't backtrack. If we replace one `\d+` with two separate `\d+\d+`, nothing changes:

```
\d+
(123456789)!

\d+ . . . \d+ . . .
(1234)(56789)!
```

And in the original example `^(\w+\s?)*$` we may want to forbid backtracking in `\w+`. That is: `\w+` should match a whole word, with the maximal possible length. There's no need to lower the repetitions count in `\w+` or to split it into two words `\w+\w+` and so on.

Modern regular expression engines support possessive quantifiers for that. Regular quantifiers become possessive if we add + after them. That is, we use `\d++` instead of `\d+` to stop + from backtracking.

Possessive quantifiers are in fact simpler than "regular" ones. They just match as many as they can, without any backtracking. The search process without backtracking is simpler.

There are also so-called "atomic capturing groups" – a way to disable backtracking inside parentheses.

...But the bad news is that, unfortunately, in JavaScript they are not supported.

We can emulate them though using a "lookahead transform".

## Lookahead to the rescue!

So we've come to real advanced topics. We'd like a quantifier, such as `+` not to backtrack, because sometimes backtracking makes no sense.

The pattern to take as many repetitions of `\w` as possible without backtracking is: `(?=(\w+))\1`. Of course, we could take another pattern instead of `\w`.

That may seem odd, but it's actually a very simple transform.

Let's decipher it:

- Lookahead `?=` looks forward for the longest word `\w+` starting at the current position.
- The contents of parentheses with `?=...` isn't memorized by the engine, so wrap `\w+` into parentheses. Then the engine will memorize their contents
- ...And allow us to reference it in the pattern as `\1`.

That is: we look ahead – and if there's a word `\w+`, then match it as `\1`.

Why? That's because the lookahead finds a word `\w+` as a whole and we capture it into the pattern with `\1`. So we essentially implemented a possessive plus `+` quantifier. It captures only the whole word `\w+`, not a part of it.

For instance, in the word `JavaScript` it may not only match `Java`, but leave out `Script` to match the rest of the pattern.

Here's the comparison of two patterns:

```
alert("JavaScript".match(/w+Script/)); // JavaScript
alert("JavaScript".match(/(?=(w+))\1Script/)); // null
```

1. In the first variant `\w+` first captures the whole word `JavaScript` but then `+` backtracks character by character, to try to match the rest of the pattern, until it finally succeeds (when `\w+` matches `Java`).
2. In the second variant `(?=(\w+))` looks ahead and finds the word `JavaScript`, that is included into the pattern as a whole by `\1`, so there remains no way to find `Script` after it.

We can put a more complex regular expression into `(?=(\w+))\1` instead of `\w`, when we need to forbid backtracking for `+` after it.

**❗ Por favor tome nota:**

There's more about the relation between possessive quantifiers and lookahead in articles [Regex: Emulate Atomic Grouping \(and Possessive Quantifiers\) with LookAhead](#) and [Mimicking Atomic Groups](#).

Let's rewrite the first example using lookahead to prevent 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, works and fast!
```

Here `\2` is used instead of `\1`, because there are additional outer parentheses. To avoid messing up with the numbers, we can give the parentheses a name, e.g. `(?<word>\w+)`.

```
// parentheses are named ?<word>, referenced as \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
```

The problem described in this article is called “catastrophic backtracking”.

We covered two ways how to solve it:

- Rewrite the regexp to lower the possible combinations count.
- Prevent backtracking.

## 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)` devuelve coincidencias una después de la otra.

Un ejemplo (con el indicador `g`):

```
let str = 'let varName'; // encontremos 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 `regex.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 `regex.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 `regex` 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(regex)

El método `str.match(regex)` encuentra coincidencias para las expresiones regulares (`regex`) en la cadena (`str`).

Tiene 3 modos:

1. Si la expresión regular (`regex`) 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)
```

2. Si la expresión regular (`regex`) 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(regex) || [];
```

### str.matchAll(regex)

#### Una adición reciente

Esta es una adición reciente al lenguaje. Los navegadores antiguos pueden necesitar polyfills.

El método `str.matchAll(regex)` 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, no retorna `null`, pero sí un objeto iterable vacío.

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 una 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 <code>n</code> ésimo 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:

1. Si el primer argumento es un string, reemplaza *todas las ocurrencias* del string, mientras que `replace` solamente reemplaza la *primera ocurrencia*.
2. Si el primer argumento es una expresión regular sin 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 una 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, binary arrays

#### Concatenate typed arrays

```
function concat(arrays) {
 // sum of individual array lengths
 let totalLength = arrays.reduce((acc, value) => acc + value.length, 0);

 if (!arrays.length) return null;

 let result = new Uint8Array(totalLength);

 // for each array - copy it over result
 // next array is copied right after the previous one
 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 que 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.](#)

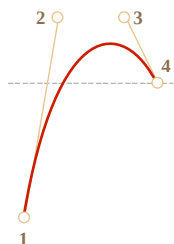
[A formulación](#)

### 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](#)

## Custom elements

### Live timer element

Please note:

1. We clear `setInterval` timer when the element is removed from the document. That's important, otherwise it continues ticking even if not needed any more. And the browser can't clear the memory from this element and referenced by it.
2. We can access current date as `elem.date` property. All class methods and properties are naturally element methods and properties.

[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 escaparlos.

### 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 escaparlos e insertarlos 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:#fddee #fd2"
```

```
alert(str.match(regex)); // #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 demosetremos 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-fA-F]{2}(:[0-9a-fA-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 espacios opcionales 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+)?)\s*/;

 let result = expr.match(regexp);

 if (!result) return [];
 result.shift();
}
```

```
 return result;
}

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(JavaScript)?|C(\+\+)?|PHP.

En acción:

```
let regexp = /Java(JavaScript)?|C(\+\+)?|PHP/g;

let str = "Java, JavaScript, PHP, C, C++";

alert(str.match(regexp)); // Java,JavaScript,PHP,C,C++
```

A formulación

### Encuentra la pareja bbttag

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(regex)); // [b]hello![/b],[quote][url]http://google.com[/url][/quote]
```

Tenga en cuenta que además de escapar `[` y `]`, 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 and lookbehind

#### Find non-negative integers

The regexp for an integer number is `\d+`.

We can exclude negatives by prepending it with the negative lookbehind: `(?<!\-)\d+`.

Although, if we try it now, we may notice one more “extra” result:

```
let regexp = /(?!-)\d+/g;

let str = "0 12 -5 123 -18";

console.log(str.match(regexp)); // 0, 12, 123, 8
```

As you can see, it matches `8`, from `-18`. To exclude it, we need to ensure that the regexp starts matching a number not from the middle of another (non-matching) number.

We can do it by specifying another negative lookbehind: `(?!-)(?!\d)\d+`. Now `(?!\d)` ensures that a match does not start after another digit, just what we need.

We can also join them into a single lookbehind here:

```
let regexp = /(?![-\d])\d+/g;

let str = "0 12 -5 123 -18";

alert(str.match(regexp)); // 0, 12, 123
```

## A formulaci3n

### Insert After Head

In order to insert after the `<body>` tag, we must first find it. We can use the regular expression pattern `<body.*?>` for that.

In this task we don't need to modify the `<body>` tag. We only need to add the text after it.

Here's how we can do it:

```
let str = '...<body style="...">...';
str = str.replace(/<body.*?>/, '$&<h1>Hello</h1>');

alert(str); // ...<body style="..."><h1>Hello</h1>...
```

In the replacement string `$&` means the match itself, that is, the part of the source text that corresponds to `<body.*?>`. It gets replaced by itself plus `<h1>Hello</h1>`.

An alternative is to use lookbehind:

```
let str = '...<body style="...">...';
str = str.replace(/(?=<body.*?>)/, '<h1>Hello</h1>');

alert(str); // ...<body style="..."><h1>Hello</h1>...
```

As you can see, there's only lookbehind part in this regexp.

It works like this:

- At every position in the text.
- Check if it's preceeded by `<body.*?>`.
- If it's so then we have the match.

The tag `<body.*?>` won't be returned. The result of this regexp is literally an empty string, but it matches only at positions preceeded by `<body.*?>`.

So it replaces the "empty line", preceeded by `<body.*?>`, with `<h1>Hello</h1>`. That's the insertion after `<body>`.

P.S. Regexp flags, such as `s` and `i` can also be useful: `/<body.*?>/si`. The `s` flag makes the dot `.` match a newline character, and `i` flag makes `<body>` also match `<BODY>` case-insensitively.

## A formulaci3n