



Depurando JS con DevTools

UD1: Tecnologías para el desarrollo de
interfaces





Objetivos de aprendizaje

- Aprender a usar las principales funcionalidades de la consola de Chrome para depurar JavaScript



DevTools

Shift+Ctrl+J o F12



- Herramientas de desarrollo de Chrome
- Facilitan la depuración y control de errores de las tecnologías de lado cliente

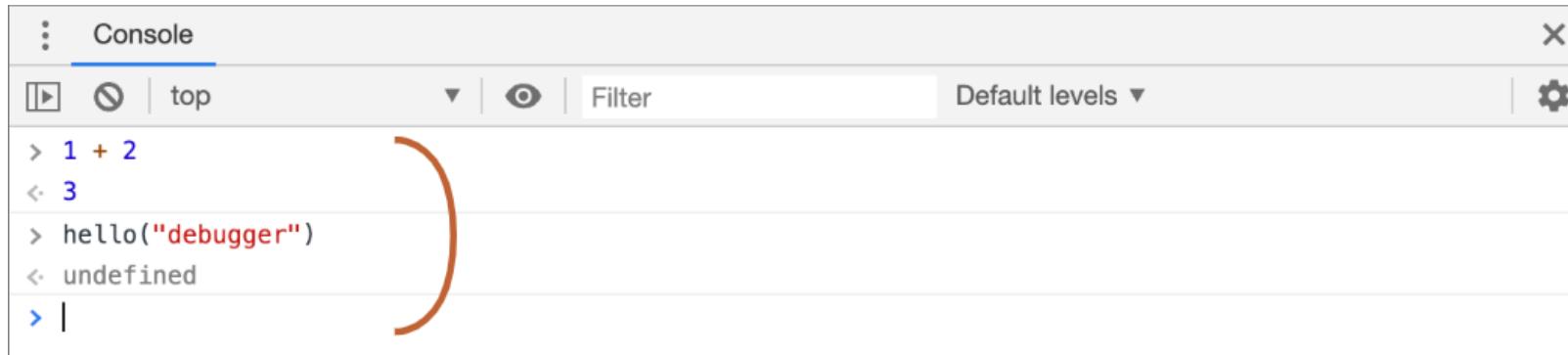
The screenshot shows the Google Chrome DevTools interface with the 'Console' tab selected. The console output displays several error messages related to the 'Permissions-Policy' header and DevTools source maps. The errors include:

- Error with Permissions-Policy header: Unrecognized feature: 'ch-ua-form-factor'.
 - This document requires 'TrustedScript' assignment.
- Error with Permissions-Policy header: Unrecognized feature: 'ch-ua-form-factor'.
- Error with Permissions-Policy header: Unrecognized feature: 'ch-ua-form-factor'.
- DevTools failed to load source map: Could not load content for https://www.google.com/_mss/k=boo-search/_ss/k=boo-search.VisualFrontendUi...AAAAAAAAbB3/d=1/ed=1/rs=AH7-fc567FnP401gSoHUEjTF1kb8WUg50/chrome.css.map: HTTP error: status code 404, net::ERR_HTTP_RESPONSE_CODE_FAILURE
- DevTools failed to load source map: Could not load content for https://www.google.com/_mss/k=boo-one-google/_ss/k=boo-one-google.OneGoogle_0/am=CADMGw/d=1/ed=1/rs=AM-SdItSzqkgnXNzIycjPj9sKu09HC9AV0/chrome.css.map: HTTP error: status code 404, net::ERR_HTTP_RESPONSE_CODE_FAILURE
- [Violation] Added non-passive event listener to a scroll-blocking <some> event. Consider marking event handler as 'passive' to make the page more responsive. See <URL>



La consola

- En el menú superior “Console” (atajo con esc)



```
> 1 + 2
<- 3
> hello("debugger")
<- undefined
> |
```

- Muy útil para probar “sobre la marcha” el resultado de una expresión

Opciones útiles de la consola

Función	Utilidad	Ejemplo
<code>console.log()</code>	Mostrar mensajes de información	<code>console.log('Hola');</code> <code>console.log(variable);</code>
<code>console.warn()</code> <code>console.error()</code> <code>console.debug()</code>	Muestra mensajes de advertencia / error / debugging	<code>console.warn('Cuidado!');</code> <code>console.warn('Error!');</code> <code>console.deb('Registro guardado');</code>
<code>console.table()</code>	Muestra información de manera tabular Se usa para tablas o para objetos	<code>console.table(["apples", "oranges", "bananas"]);</code>



Panel “Sources”

- Partes:
 - 1. Zona de recursos
 - 2. Código fuente de los archivos
 - 3. Zona con opciones de debugging

The screenshot shows the Chrome DevTools interface with the "Sources" tab selected. The top navigation bar includes "Elements", "Console", "Sources", "Network", "Performance", "Memory", "Application", and other developer tools.

The left sidebar lists resources:

- top
- javascript.info
- article/debugging-chrome/deb
- index.html
- hello.js** (highlighted with a blue bar)

Large orange numbers 1, 2, and 3 are overlaid on the interface to identify its components:

- 1** Points to the resource tree on the left.
- 2** Points to the code editor pane displaying the contents of `hello.js`:

```
function hello(name) {
  let phrase = `Hello, ${name}!`;
  say(phrase);
}

function say(phrase) {
  alert(`** ${phrase} **`);
}
```

Line 1, Column 1

The right sidebar contains a list of debugging features:

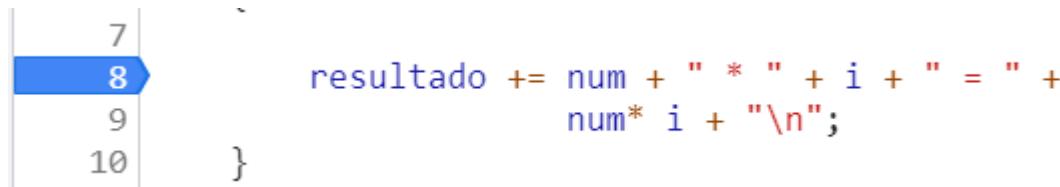
- Watch
- Call Stack
- Scope
- Breakpoints
- XHR/fetch Breakpoints
- DOM Breakpoints
- Global Listeners
- Event Listener Breakpoints

Large orange numbers 1, 2, and 3 are overlaid on the interface to identify its components:

- 1** Points to the resource tree on the left.
- 2** Points to the code editor pane displaying the contents of `hello.js`.
- 3** Points to the sidebar menu on the right.

Breakpoints

- Son puntos donde queremos que nuestro código se detenga en tiempo de ejecución
- Se insertan haciendo clic sobre el número de línea:



A screenshot of a code editor showing a line of code and a vertical margin on the left. The number 8 is highlighted in blue, indicating it is a breakpoint. The code is:

```
resultado += num + " * " + i + " = " +
    num * i + "\n";
```

- También se puede usar la instrucción **debugger**; en el código para forzar dicha detención
- Los siguientes botones permiten ir paso a paso decidiendo si entrar o no en el código interno de cada función:



Opciones de ejecución

- Reanudar, continua siguiente instrucción (si no hay más breakpoints, termina la ejecución): F8 
- Siguiente paso (ejecuta la siguiente sentencia): F9 
- Saltar paso (siguiente sentencia, sin entrar en funciones): F10 
- Continuar la ejecución hasta el final de la función actual: Shift + F11 



Si hay un error lo vas a encontrar

hello.js x

```
1 function hello(name) { name = "John"
2   let phrase = `Hello, ${name}!`; phrase = "Hello, John!"
3
4 say(phrase);
5 }
6
7 function say(phrase) {
8   alert(`** ${phrase} **`);
9 }
```

ver expresiones →

ver los detalles de la llamada externa →
variables actuales →

Paused on breakpoint

Watch 1 + C

No watch expressions

Call Stack 2

hello hello.js:4

(anonymous) index.html:10

Scope 3

Local

name: "John"
phrase: "Hello, John!"

this: Window

Global

Window

{ } Line 4, Column 3

¿Cómo?

1. La sección **Watch** permite mirar el valor y tipo de datos que van tomando las variables en tiempo de ejecución
 - El valor de las variables también se puede ver posando el ratón sobre la misma en el código
2. La sección **Call Stack** muestra las llamadas anidadas
3. En **Scope** están las variables activas, locales y globales



Ejemplo 1

- Realiza un script que pida un número entero por pantalla y a continuación genere su tabla de multiplicar del 1 al 10 (es decir, el resultado de multiplicar el número indicado sucesivamente por los números del 1 al 10)
- El resultado se presentará en 1 mensaje por consola
- Para resolver el ejercicio crea el menos la siguiente función:
 - `tablaMultiplicar(entero):cadena`



Ejemplo 1 - BIS

- Introduce un error en el código, por ejemplo cambiando el operador de acumular por el de asignar:

```
resultado += num + " * " + i + " = " + num* i + "\n";
```

```
resultado = m + " * " + i + " = " + num* i + "\n";
```

- Inserta un ***breakpoint*** en esa línea y traza el código, comprobando en todo momento que valores van tomando las variables resultado, num e i

Snippets

- En la consola, sección **Source**, parte izquierda, existe la posibilidad de crear **Snippets** de código
 - Son fragmentos de código con una funcionalidad concreta
- Se pueden ejecutar con **Run** y llamarlos desde la consola para probar su funcionalidad



Tabla de multiplicar..

```
function tablaMultiplicar(num){  
    let resultado = "\n";  
  
    for(let i=1; i<=10; i++){  
        resultado += num + " * " + i + " = " +  
            num* i + "\n";  
    }  
    return resultado;  
}
```



Ejemplo de snippet

- Crea un **Snippet** “Tabla de multiplicar” con el código para generar la tabla de multiplicar de un número que se pasa como parámetro.
- Ejecútalo y comprueba que funciona generando la tabla del 4

```
console.log(tablaMultiplicar(4))
```

4 * 1 = 4
4 * 2 = 8
4 * 3 = 12
4 * 4 = 16
4 * 5 = 20
4 * 6 = 24
4 * 7 = 28
4 * 8 = 32
4 * 9 = 36
4 * 10 = 40

Ejemplos 3 y 4

- Pon en práctica todo lo aprendido hoy para depurar los dos scripts ejemplo3 y ejemplo4 que contienen errores de diversa índole

