Depuración de Node.js

Tiempo estimado: 30min

La depuración es algo que todo usuario debe conocer para desenvolverse con soltura en Node y así resolver problemas de programación fácil y rápidamente. No hay una buena base de Node sin conocer un depurador con el que encontrar fallos en el software. En este capítulo, centramos nuestra atención en el inspector V8, el depurador de facto desarrollado por Google para su motor JavaScript que, recordemos, es el que usa Node.

La lección comienza con una introducción a la depuración, proceso mediante el cual se detecta y corrige un fallo de software. También se introduce los componentes o elementos que todo buen depurador debe tener: los puntos de interrupción, la ejecución paso a paso, los observadores y la pila de llamadas. Concluimos, presentando el inspector V8 y cómo trabajar con él.

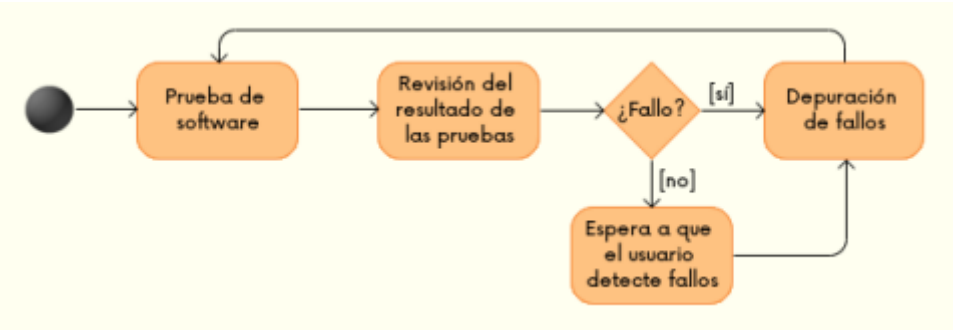
Al finalizar la lección, el estudiante sabrá:

- Qué es la depuración y para qué sirve.
- Qué es el inspector V8.
- Qué son los puntos de interrupción y cómo definirlos mediante el inspector.
- Qué es la ejecución paso a paso y cómo trabajar con ella en el inspector.
- Qué son los observadores y cómo definirlos en el inspector.
- Qué es la pila de llamadas y cómo visualizarla en el inspector.
- Cómo ocultar módulos de JavaScript de cara a la depuración mediante el *blackboxing*.

Introducción

Formalmente, la depuración (*debugging*) es el proceso mediante el cual se corrige uno o más defectos o fallos en el software. Por un lado, se busca la causa del problema, es decir, por qué se está dando el error y, una vez encontrado, se resuelve o corrige. Una vez finalizada la depuración, se vuelve a someter el software a la batería de pruebas para validar que realmente se ha resuelto el fallo sin añadir ninguno nuevo, es decir, que la corrección del defecto no tiene efectos colaterales en otras partes del software que no los presentaba antes.

A continuación, se muestra un diagrama de estado que muestra el ciclo de vida del software, en cuanto a fallos se refiere:



Tal como se puede observar, nunca se da por finalizada la prueba, más pronto o más tarde, aparecerá un problema. Como buenos desarrolladores, nuestro objetivo es intentar que llegue el menor número de defectos al usuario, para que así no nos saquen los colores y nuestra organización o el cliente no se vea perjudicada por un fallo importante.

Depurador

El depurador (*debugger*) o herramienta de depuración (*debugging tool*) es el programa utilizado para

ayudar a depurar software, en nuestro caso particular, código de JavaScript en la plataforma Node.

Los depuradores principalmente proporcionan las siguientes características:

- Ejecución paso a paso (*stepping*). Permite ejecutar una proposición cada vez con objeto de ver cómo quedan las cosas tras la ejecución de la proposición.
- Puntos de interrupción (*breakpoints*). Permite indicar puntos en los que el depurador detendrá la ejecución.

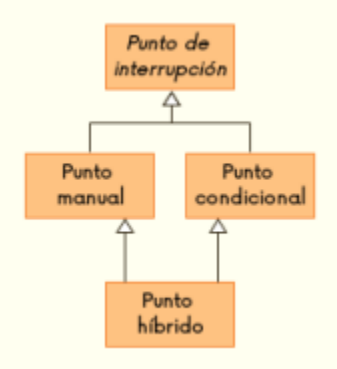
  En ese punto, podremos comprobar los valores de las variables, decidir seguir ejecutando paso a paso o hasta el siguiente punto de interrupción, comprobar la pila de llamadas o finalizar la ejecución.
- Observadores (*watchers*). Permite visualizar el valor de una determinada variable o expresión.
- Consola de consulta (*drawer*). Permite ejecutar expresiones de JavaScript usando como variables el contexto actual de ejecución.
- Pila de llamadas (*call stack*). Permite visualizar qué funciones hay abiertas en el punto bajo depuración.

La idea que se esconde tras un depurador es muy, pero que muy sencilla. Ejecutar un *script* de Node y, entonces, ejecutarlo proposición a proposición analizando si su flujo de ejecución es el esperado, comprobando los valores de las variables.

Puntos de interrupción

Un punto de interrupción (*breakpoint*) es una línea del programa en el que el depurador debe pausar la ejecución, permitiéndonos así inspeccionar la situación en la que nos encontramos como, por ejemplo, la pila de llamadas y los valores de las variables. Durante una sesión de depuración, podemos fijar tantos puntos de interrupción como sea necesario.

Los puntos de interrupción se clasifican en manuales, condicionales e híbridos.



Un punto de interrupción manual (*manual breakpoint*) es aquel que se asocia a una determinada línea y detiene la ejecución cuando se alcanza ésta. Generalmente, se usa este tipo de punto de interrupción cuando tenemos sospechas de que el problema se encuentra en esa línea o cerca.

En cambio, un punto de interrupción condicional (*conditional breakpoint*) tiene asociada una condición que actúa como interruptor de detención. Cuando se cumple la condición, el punto de interrupción se alcanza, independientemente del punto del programa en el que nos encontramos.

Un punto de interrupción híbrido (*hybrid breakpoint*) está asociado a una determinada línea, al igual que los manuales, pero además tienen asociada una condición. De tal manera que la detención del flujo de ejecución sólo se produce si se alcanza la línea y además se cumple la condición asociada.

Cuantos más tipos de puntos de interrupción soporte nuestro depurador, mejor. Mayor flexibilidad tendremos disponible.

Sentencia debugger

En JavaScript, generalmente, los puntos de interrupción se pueden fijar a nivel de línea, a través de la interfaz de usuario proporcionada por el depurador o bien mediante la sentencia debugger del

lenguaje. Cuando se ejecuta esta sentencia, si el motor de JavaScript se encuentra en modo depuración, detendrá la ejecución. Si no lo está, simplemente la ignorará.

Su sintaxis es muy sencilla:

```
debugger ;
```

Esta sentencia es por lo tanto intrusiva, pues hay que insertarla en el código fuente de la aplicación, a diferencia de otros modos de inserción de puntos de interrupción que no lo son, pues se fijan a nivel del depurador, no del código fuente. Pero aún así, es muy útil saber su existencia y utilizarla cuando es necesario, por ejemplo, en combinación con una sentencia if para simular un punto de interrupción híbrido.

Ejecución paso a paso

La ejecución paso a paso (*stepping*) es la capacidad del depurador de permitir al usuario ejecutar las proposiciones poco a poco, según nuestra necesidad, tras un punto de interrupción. Cuando se detiene la ejecución del programa en un determinado punto de interrupción, el depurador nos permite continuar la ejecución de varias formas:

- Continuar (*continue* o *resume*). Cuando reanudamos la ejecución con el comando continue o resume, lo que le estamos diciendo al depurador es: *continua la ejecución hasta que te encuentres con un punto de interrupción o el final del programa*.
- Pasar por encima (*step over*). Si usamos step over, lo que hacemos es decirle que ejecute la línea actual y se detenga de nuevo tras ella.

  Así pues, el depurador se detendrá en la siguiente línea de código, ejecutando todo aquello que sea necesario de la línea en curso. Si ésta tiene llamadas a función, las ejecutará sin pararse en ellas.
- Entrar (*step into*). En ocasiones, la línea actual tiene llamadas a funciones. Si deseamos detener el flujo de ejecución en la primera línea del cuerpo de cualquier función invocada en la línea en curso, debemos utilizar el comando step in.

  Si la línea actual no ejecuta ninguna llamada a función, tiene el mismo comportamiento que step over.
- Salir (*step out*). Si estamos dentro de una función y deseamos salir de su cuerpo y detenernos en la línea que provocó su llamada, se puede usar step out.

Generalmente, los comandos de ejecución paso a paso se suelen representar mediante las siguientes imágenes:

La primera representa el comando Resume. La segunda Step Over. La tercera Step Into. Y la cuarta Step Out. Como se puede observar, son bastante descriptivas y representativas de las operaciones que llevan a cabo.

Observadores

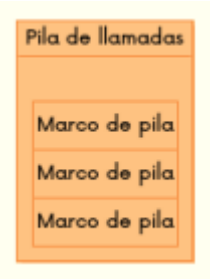
Un observador (*watcher*) es una funcionalidad del depurador mediante la cual se puede solicitar la monitorización de una determinada variable o expresión. Cada vez que se ejecuta una línea del programa, el observador muestra el valor de la variable o expresión bajo observación. Lo que permite visualizar su valor fácilmente.

Consola de consulta

Generalmente, los depuradores proporcionan una consola de consulta (*drawer*) a través de la cual ejecutar expresiones JavaScript con el contexto de ejecución que hay en el punto en el que nos encontramos detenidos. Estas expresiones pueden ser simples consultas del valor de una variable o de una determinada expresión, similar a un observador. Pero se pueden hacer interactivamente.

Pila de llamadas

Una función puede invocar otra función y ésta a su vez a otra y, así, sucesivamente. La pila de llamadas (*call stack*) es una estructura de datos, que utiliza internamente el motor, en la que se deposita las distintas llamadas que están activas actualmente. Cada vez que se invoca una función, se deposita su información en esta pila; a su vez, si la función invoca otra, la información de esta nueva llamada también se deposita en la pila, justo encima de la anterior. Así, es fácil para el motor recurrir a la pila de llamadas para determinar el orden de las invocaciones.



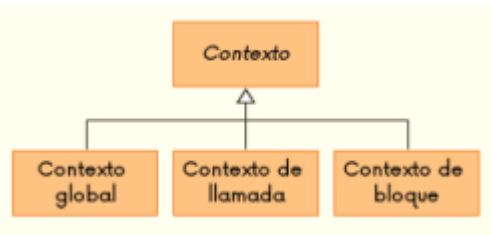
Cada apilamiento o entrada de la pila representa una invocación y se conoce formalmente como marco de pila (*stack frame*). Este marco contiene básicamente tres cosas: el objeto función invocado, la proposición que ha generado la llamada y un contexto de ejecución privado.

El objeto función invocado no es más que la función invocada.

La proposición que ha generado la llamada es eso, la proposición que ha invocado la función. Esta información permitirá al motor que, terminada la ejecución de la función activa, pueda volver atrás y continuar con el flujo de ejecución, es decir, con la proposición que invocó la función.

Finalmente, cada marco de pila contiene un contexto de ejecución privado (*private execution context*), también conocido como tabla de símbolos (*symbol table*), el conjunto de variables locales de la función invocada y los valores particulares que tiene cada una de ellas para esa invocación. Este contexto contiene los parámetros de la función y cada una de las variables locales definidas dentro de ella, junto con sus respectivos valores para esa invocación. Cuando la función finaliza, o lo que es lo mismo, cuando finaliza la invocación, el contexto de ejecución se destruye automáticamente. En algunos lenguajes, el contexto de ejecución se conoce también como ámbito (*scope*).

Básicamente, se puede distinguir tres tipos de contextos de ejecución: el global, el de llamada y el de bloque.



El contexto global (*global context*) o ámbito global (*global scope*) es aquel que contiene las variables globales, accesibles en cualquier función. Por su parte, un contexto de llamada (*call context*) o ámbito de llamada (*call scope*) es aquel que se crea con cada invocación para almacenar las variables específicas de esa llamada. Finalmente, el contexto de bloque (*block context*) o ámbito de bloque (*block scope*) es aquel que se crea cuando se utiliza un bloque para definir variables cuyo ámbito es el bloque que las define.

En todo momento, puede haber un único contexto de ejecución global, pero varios de llamada o de bloque. A la cadena particular de contextos, desde el más local al global, se le conoce como cadena de ámbitos (*scope chain*). Cuando el motor realiza una búsqueda de identificador (*identifier lookup*) para encontrar una variable, consulta la cadena de ámbitos actual, desde la más interna o local a la más externa o global. En caso de no encontrar ninguna variable tras consultar todos los ámbitos de la cadena, propagará un error.

Es importante entender el funcionamiento de la pila. En todo momento, un hilo de ejecución sólo puede tener una función activa, que consistirá en la última función invocada, o sea, la que se encuentra en lo alto de la pila. A esta función, se la conoce formalmente como función activa (*active function*); al resto

de funciones que se encuentran en la pila, como funciones en espera (*standby functions*).

Cuando la función activa finaliza, tanto si lo hace normal como anormalmente, el motor desapila o elimina su marco de pila, esto es, el marco que se encuentra en lo alto de la pila, y entonces continúa con la ejecución de la siguiente función, la cual se reconvertirá en la nueva activa. A esta operación de supresión del marco, se la conoce formalmente como limpieza de marco de pila (*stack frame clear*) o desapilamiento de marco de pila (*stack frame pop*) y conlleva, claro está, la reactivación de la función que invocó la función que acaba de terminar, para así continuar con su ejecución.

Inspector V8

El inspector V8 (*V8 inspector*) es un depurador gráfico para Node. Se apoya en Chrome DevTools, con el objeto de mejorar la experiencia de depuración, facilitando así su uso a los que ya trabajan con esta herramienta. Como la interfaz de usuario es el navegador, se puede depurar remotamente.

Su único requisito es Chrome.

Las principales características del depurador son:

- Es muy sencillo.
- Es gráfico.
- Permite definir puntos de interrupción manuales e híbridos.
- Permite la ejecución paso a paso.
- Permite definir observadores.
- Permite consultar la pila de llamadas.
- Proporciona una consola de consulta.

Arquitectura del inspector V8

El inspector V8 dispone, por una parte, de un comando con el que solicitar la ejecución de un módulo en modo depuración y, por otra parte, de una interfaz gráfica con la que interactuar con el depurador. El comando depurador es el propio node, pero hay que invocarlo con la opción --inspect. Mientras que la interfaz gráfica es Chrome DevTools.

Depuración de un módulo

Para ejecutar un módulo de Node en modo depuración, hay que ejecutarlo con el comando node y la opción --inspect. A continuación, habrá que abrir Chrome e ir al URL indicado para interactuar con el inspector V8.

Opción --inspect

Mediante la opción --inspect, le solicitamos a node que ejecute el módulo JavaScript indicado monitorizándolo con el inspector V8. El inspector es una aplicación web servidora para realizar la depuración remotamente mediante Chrome.

De manera predeterminada, el inspector escucha en el puerto 9229, pero podemos indicar otro cualquiera si fuera necesario. La sintaxis de esta opción es la siguiente:

```
--inspect
--inspect=puerto
```

He aquí un ejemplo de ejecución bajo el inspector V8:

```
$ node --inspect --debug-brk index.js
Debugger listening on port 9229.
Warning: This is an experimental feature and could change at any time.
To start debugging, open the following URL in Chrome:
chrome-
devtools://devtools/remote/serve_file/@60cd6e859b9f557d2312f5bf532f6aec5f284980/inspector.ht
ml?experiments=true&v8only=true&ws=localhost:9229/95072e64-ca46-4856-840f-2a7009f08129
```

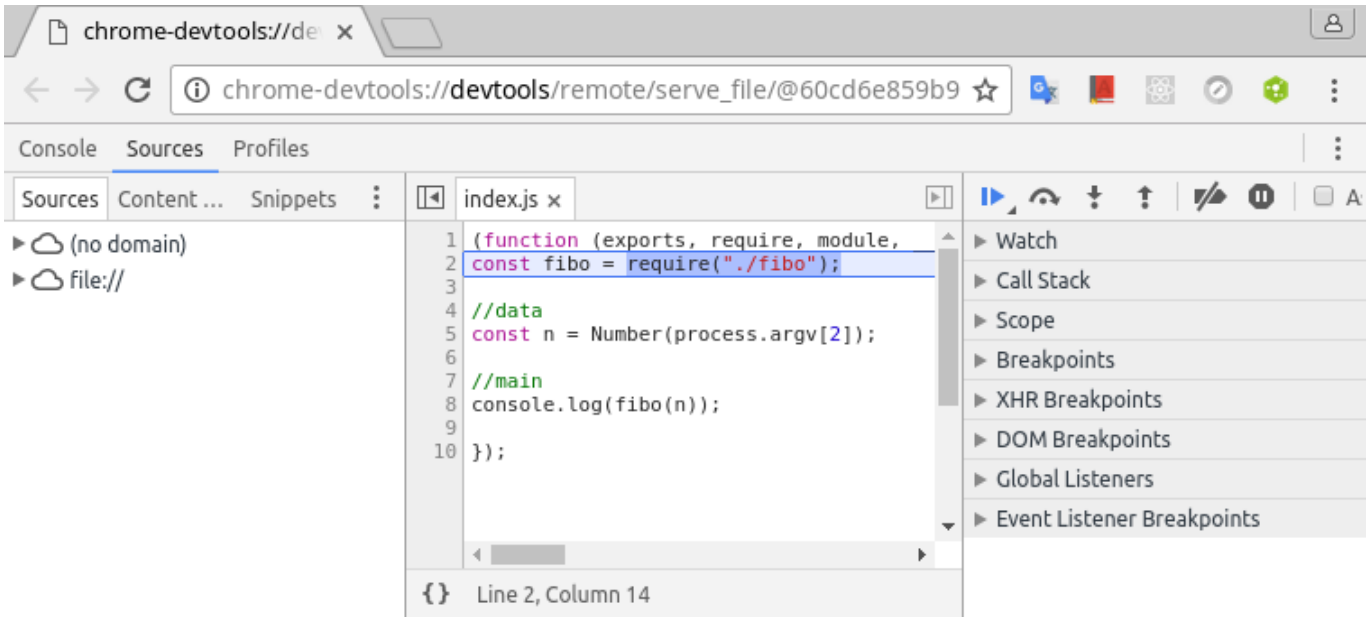
Observe que node indica el puerto al que hay que conectar y el URL a solicitar para interactuar con el inspector.

Opción --debug-brk

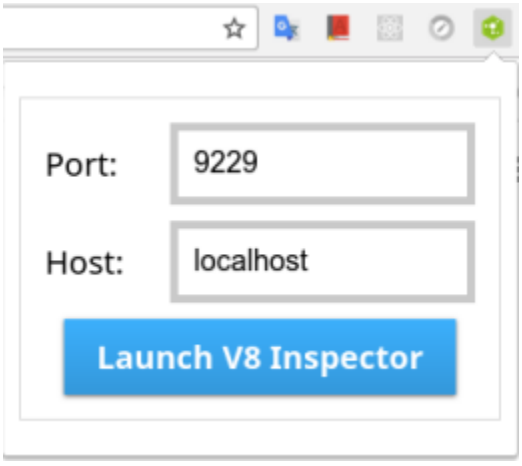
Por su parte, la opción --debug-brk lo que indica es que node no debe ejecutar nada. Debe detenerse en la primera línea del *script*. Si no la indicamos, comenzará a ejecutar hasta que finalice su ejecución.

Extensión Node.js V8 Inspector de Chrome

Una vez solicitada la ejecución del *script* mediante node y el inspector V8, lo siguiente es abrir Chrome y conectar al URL chrome-devtools indicado en la salida del comando. No hay más que copiarlo y pegarlo en la barra de direcciones y *voilà*:



Para facilitar la conexión al inspector V8, se puede utilizar la extensión Node.js V8 Inspector de Chrome. Una vez instalada, podremos usar la extensión y conectar al inspector simplemente indicando el puerto en el que escucha:



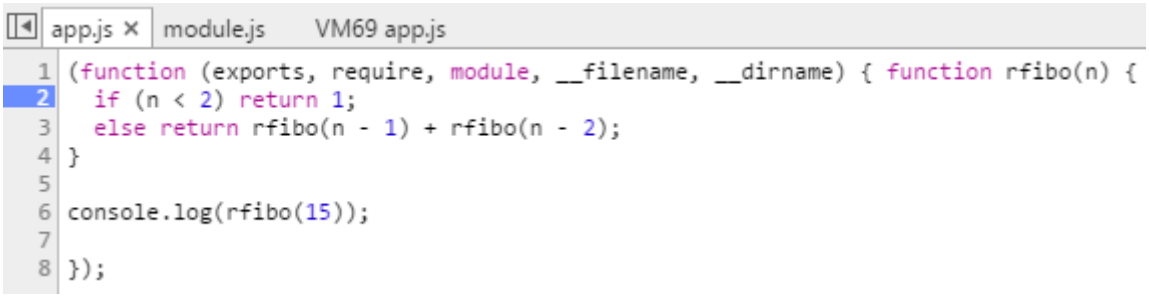
Definición de puntos de interrupción

Recordemos que podemos definir un punto de interrupción mediante la sentencia debugger de JavaScript o bien hacerlo directamente en el inspector v8. La primera opción es intrusiva porque se encuentra en el código fuente del software, mientras que la segunda no lo es.

Definición de puntos de interrupción manuales

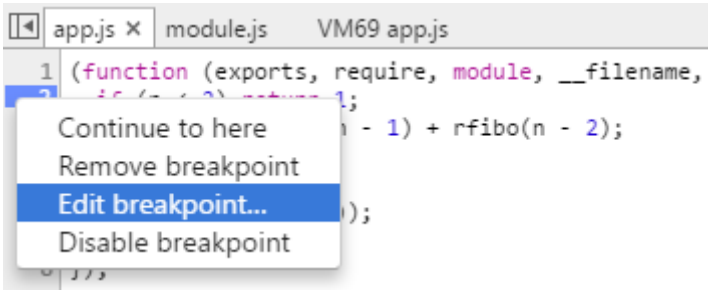
Para definir un punto de interrupción manual en el inspector, recordemos, aquellos que están asociados a una determinada línea de código, no hay más que abrir el archivo en el inspector y hacer clic en el número de la línea a la que asociar el punto de interrupción.

En el siguiente ejemplo, el punto de interrupción se encuentra en la línea 2:

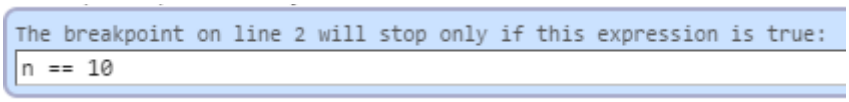


Definición de puntos de interrupción híbridos

En el inspector V8, es posible añadir puntos de interrupción híbridos, recordemos, aquellos que se asocian a una línea y además tienen asociada una condición. Para ello, primero hay que fijar un punto de interrupción manual. Después, mostrar su menú contextual y seleccionar Edit breakpoint...:



A continuación, indicar la condición que asociar al punto de interrupción:

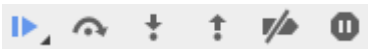


Ahora, el depurador sólo detendrá la ejecución cuando se alcance la línea y además se cumpla la condición.

El color de los puntos de interrupción manuales es distinto al de los híbridos. El primero es azul y el segundo naranja:

Ejecución paso a paso

Una vez sabemos definir puntos de interrupción en nuestra sesión de depuración, lo siguiente es explicar cómo ejecutar el código paso a paso. Para ello, disponemos de cuatro opciones: continuar (*resume*), pasar por encima (*step over*), entrar (*step into*) o salir (*step out*). Estos comandos se encuentran en el panel de ejecución, en la parte superior derecha:



Si no recuerda quién es quién, sitúese encima del botón y espere a que salga la ayuda. A pesar de todo, vamos a presentarlos formalmente:

- El primero es para continuar la ejecución hasta el siguiente punto de interrupción o el final del *script*.
- El segundo es para ejecutar sólo la línea actual, no entrando en modo depuración en las funciones invocadas en ella.
- El tercero para ejecutar la línea actual y detenerse en la primera línea del cuerpo de las funciones invocadas en ella.
- El cuarto para salir de la función actual, deteniendo el flujo de ejecución en la línea que invocó la función.

- El quinto desactiva todos los puntos de interrupción definidos.
- El sexto se utiliza para detener la ejecución si se propaga algún error o excepción.

La ejecución de un *script* en modo depuración comienza con la detención en la primera línea del programa. Por lo que habrá que hacer uso de los comandos de ejecución paso a paso para comenzar la depuración.

Definición de observadores

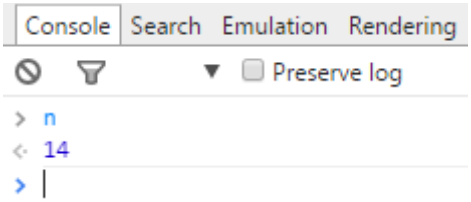
Con el inspector, podemos definir observadores, recordemos, monitores que permiten conocer el valor de una variable o expresión tras cada línea ejecutada. Se definen a través del panel Watch, ubicado bajo el panel de ejecución paso a paso:

Definición de observadores

Para definir un observador, hacer clic en el botón + y entonces indicar la expresión a monitorizar. Si sólo deseamos monitorizar una variable, indicar su nombre. Si deseamos el valor de una expresión, escribirla. He aquí unos ejemplos ilustrativos:

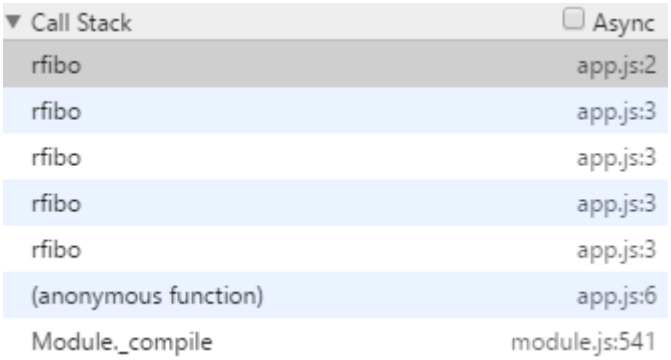
Ejecución de expresiones mediante la consola de consulta

Una característica muy interesante de los depuradores es la consola de consulta, mediante la cual podemos ejecutar expresiones JavaScript con el contexto actual de ejecución. Esta consola se muestra haciendo clic en la tecla Esc. Esto mostrará la consola, en la parte inferior, muy parecida a una sesión interactiva del comando node:



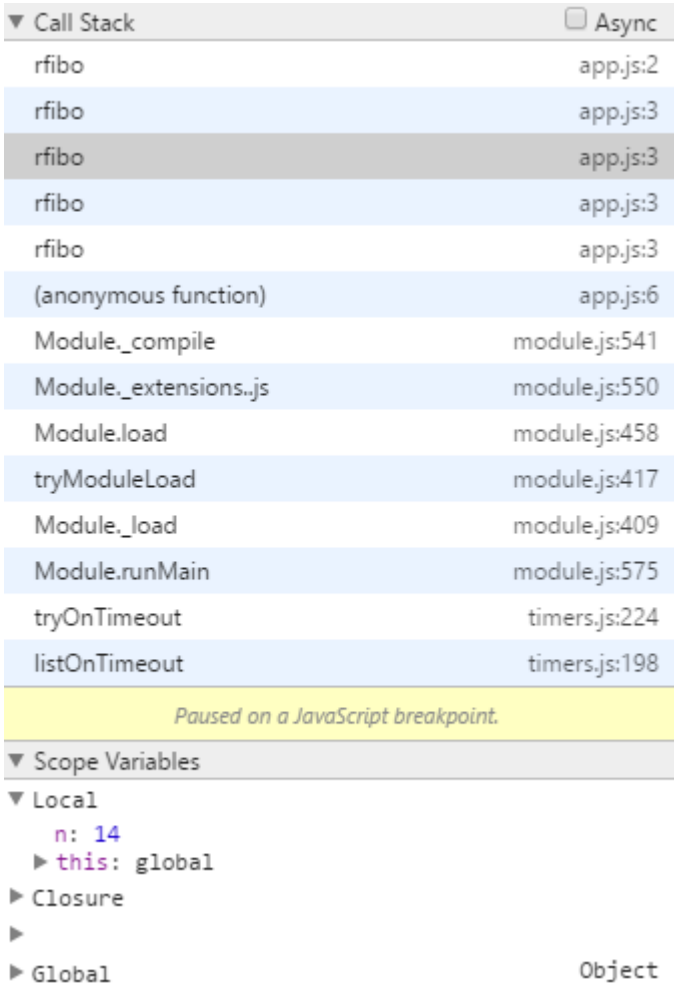
pila de llamadas

La pila de llamadas se puede visualizar mediante el panel Call Stack:



Para ayudar a leer mejor la pila, se recomienda asignar nombres a las funciones anónimas.

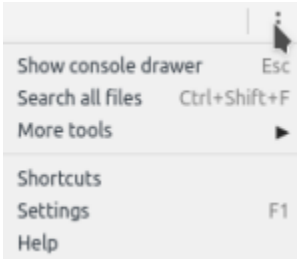
Para conocer el ámbito o contexto de ejecución de una determinada llamada, seleccionar la llamada e ir al panel Scope:



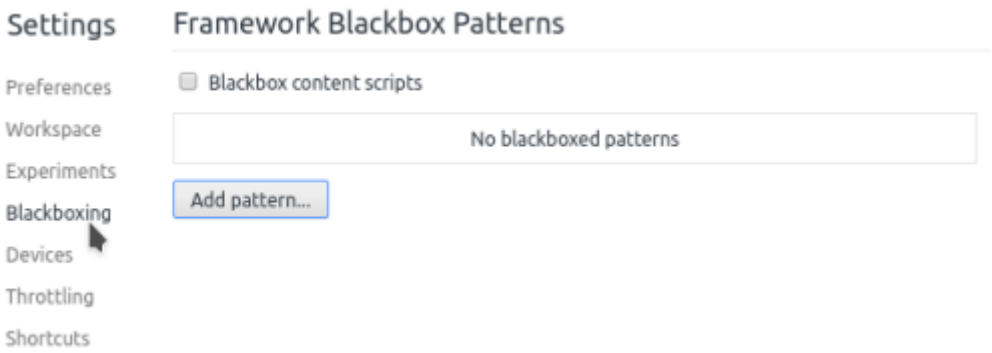
Ocultación

La ocultación (*blackboxing*) es la operación mediante la cual configuramos uno o más archivos para encubrirlos a la vista de cara a la depuración. Pero ojo, estos archivos siempre se ejecutan si es necesario, lo que ocurre que el inspector no se detendrá en su contenido.

Generalmente, el *blackboxing* se utiliza para ocultar módulos predefinidos o paquetes de terceros que no son de nuestro interés. Para ello, hay que pulsar F1 o ir a las opciones de Chrome DevTools:



A continuación, ir al panel Blackboxing:



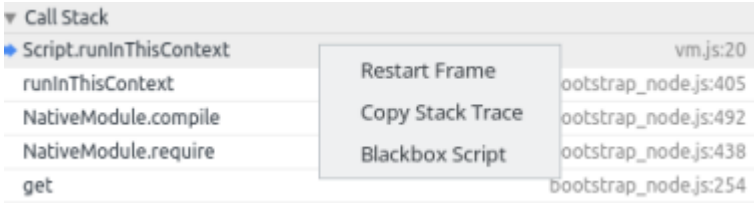
En esta página, disponemos de los patrones de los nombres de los archivos ocultos.

Añadidura de un patrón

Para añadir un patrón de invisibilidad, no hay más que hacer clic en Add pattern... y proporcionar los datos:

Añadidura de un archivo

No siempre tenemos todos los patrones añadidos al *blackboxing*. Nos damos cuenta durante la depuración cuando el inspector se detiene en una línea de un módulo que desearíamos estuviese oculto. Podemos añadirlo como hemos visto antes. O bien podemos ir al panel Call Stack, seleccionar el marco de pila que contiene la línea del módulo a ocultar, mostrar su menú contextual y seleccionar Blackbox Script:



Esto creará un patrón y lo añadirá a la lista de ocultación.

