

---

## Desarrollando tu primera aplicación

---

Incluso el viaje más largo comienza con un primer paso.  
—Confucio

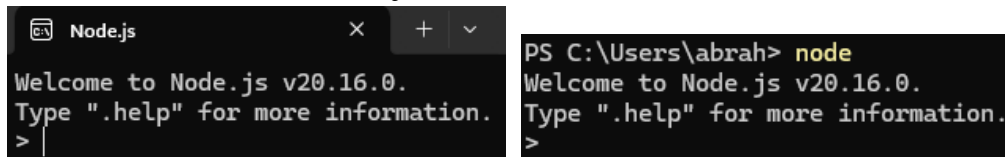
Cuando llegues aquí, ya deberías tener una instalación funcional de la plataforma Node.js en tu sistema. Para ejecutar los ejemplos de este documento, debes usar la línea de comandos de tu sistema operativo. No importa si usas Windows, Linux o macOS. Los ejemplos funcionan independientemente del sistema operativo. Puedes usar Node.js de dos maneras diferentes.

Para experimentos simples, puedes usar el shell interactivo. Alternativamente, puedes ejecutar una aplicación pasando el nombre del archivo inicial al comando node. En este caso, normalmente no se requiere ninguna interacción adicional del usuario.

### Modo interactivo

Puedes llegar al modo interactivo de Node.js, como puedes ver en el listado 1, ingresando el comando node en la línea de comandos.

Listado 1: Modo interactivo de Node.js.



The image shows two terminal windows side-by-side. The left window has a title bar that says 'Node.js' and contains the text: 'Welcome to Node.js v20.16.0.', 'Type ".help" for more information.', and a prompt '>' with a cursor. The right window has a title bar that says 'PS C:\Users\abrah>' and contains the text: 'node', 'Welcome to Node.js v20.16.0.', 'Type ".help" for more information.', and a prompt '>' with a cursor.

En el modo interactivo, puedes ingresar directamente el código JavaScript en la línea de comandos y ejecutarlo. Este tipo de interfaz de usuario se conoce como ciclo de lectura-evaluación-impresión (REPL, Read-Eval-Print Loop), lo que significa que los comandos se leen en la línea de comandos y se evalúan, y luego el resultado se muestra en la línea de comandos.

El modo no está pensado para implementar y ejecutar aplicaciones completas. En cambio, esta interfaz de Node.js se utiliza para probar el comportamiento y la funcionalidad de fragmentos de código individuales. La figura 1 muestra cómo funciona el modo interactivo.

### Uso general

El listado 2 muestra cómo se pueden emitir comandos en el REPL de Node.js.

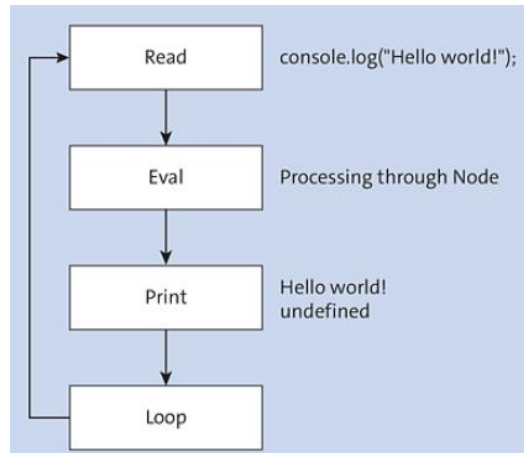


Figura 1: Modo interactivo de Node.js.

Listado 2: Ejecución de comandos en el REPL de Node.js.

```

> console.log('Hello World!');
Hello World!
undefined
>
  
```

Los comandos de JavaScript también terminan con un punto y coma en el REPL de Node.js. Un salto de línea finaliza la entrada del comando actual y envía el comando al motor de JavaScript. En el ejemplo, se evalúa el método `console.log` con el argumento `Hello World!` y el resultado `Hello World!` se muestra en la línea de comandos.

Como puedes ver en el listado 2, el valor `undefined` se muestra además del esperado `Hello World!`. Esto se debe a que se representa el valor de retorno de la función `console.log`; en este caso, es `undefined`. Además de ejecutar la salida, en REPL, también puedes ejecutar todos los comandos JavaScript disponibles en Node.js, como se muestra en el listado 3.

Listado 3: Definición de una función REPL simple de Node.js.

```

> function greet(name) {
... console.log(`Hello ${name}`);
... }; greet('World');
Hello ${name}
undefined
  
```

El Listado 3 muestra cómo definir una función en Node.js REPL que recibe un nombre como parámetro y cómo enviarlo a la consola a medida que avanza la función. Después de la definición, se llama a la función `greet` con el argumento `world` como nombre. La salida consta de `Hello world` y `undefined`. Si deseas ejecutar diferentes comandos en Node.js REPL que se basen entre sí, puedes hacerlo fácilmente porque el contexto se conserva dentro de una sesión. Por ejemplo, si defines funciones o realizas una asignación de variable, estas se conservarán

incluso después de que se evalúe la línea de comando. En el Listado 4, puedes ver cómo funciona esto en términos concretos.

Listado 4: Líneas de comando interdependientes.

```
> const sayHello = 'Hello World!';
undefined
> console.log(sayHello);
Hello World!
undefined
```

En el ejemplo del listado 4, asignas un valor a la constante sayHello. Esta constante se usa nuevamente en la salida en la declaración posterior. Entre las características más importantes de Node.js REPL se encuentra la función de autocompletado. Si presionas la tecla (Tab) sin ingresar nada más, obtendrás una lista de todos los objetos disponibles. Si utilizas la función de autocompletar junto con una o más letras, solo se mostrarán las sugerencias que coincidan.

Otra característica que vale la pena mencionar son los comandos multilínea. Como puedes ver en el listado 3, el código de la función no se ingresa en una sola línea sino en varias líneas. Un comando incompleto se indica con tres puntos en lugar del signo mayor que como símbolo del sistema. Los comandos multilínea se pueden lograr, por ejemplo, con bloques de código incompletos o un operador más para el cálculo o la concatenación de cadenas al final de la línea.

## Otros comandos REPL

Una de las características especiales del REPL de Node.js es que te proporciona algunos comandos más para controlar el REPL además del conjunto de comandos de JavaScript. Los comandos siempre comienzan con un punto y no tienen que terminar con un punto y coma. La tabla 1 contiene una descripción general de estos comandos.

Hay dos opciones disponibles para salir del REPL: usar el comando .exit o presionar (Ctrl) + (D), que también terminará el proceso inmediatamente. Alternativamente, puedes presionar (Ctrl) (C) dos veces.

Los comandos .break y .clear se usan cuando la línea de comando está bloqueada por una entrada incorrecta. El listado 5 muestra un caso de uso para estos comandos.

Listado 5 Uso de “.break”.

```
> function greet(name) {
... .break
>
```

Tabla 1: Comandos REPL disponibles.

Comando	Descripción
<code>.break</code>	Termina la entrada actual. <code>.break</code> es especialmente útil para comandos multilínea.
<code>.clear</code>	Sirve como alias para <code>.break</code> .
<code>.exit</code>	Termina el <code>node.js</code> REPL.
<code>.help</code>	Emite una lista de comandos disponibles.
<code>.load</code> <code>&lt;file&gt;</code>	Carga una sesión guardada de un archivo en el REPL.
<code>.save</code> <code>&lt;file&gt;</code>	Guarda los comandos de la sesión REPL actual en un archivo.
<code>.editor</code>	Abre el modo editor donde puedes definir un bloque de instrucciones. (Ctrl) + (D) ejecuta el bloque, y (ctrl) + (C) sale del modo editor sin ejecutar nada.

En el ejemplo del listado 5, has comenzado a formular una función, pero no deseas terminar de escribirla, prefieres cancelar la entrada. No puedes terminar la entrada actual presionando la tecla (Enter), ya que esto simplemente insertaría un salto de línea. Si descubres que has cometido un error similar en tu entrada, puedes terminar la entrada actual usando el comando `.break` e ingresar tu comando nuevamente. El mismo efecto se puede lograr mediante el atajo (Ctrl) + (C). El REPL de Node.js ofrece la opción de navegar por el historial de los comandos más recientes.

Con esta función, no es necesario que vuelvas a escribir el comando, sino que puedes utilizar las teclas de flecha (arriba) y (abajo) para navegar por el historial de comandos ingresados, recuperar la línea de comando correspondiente, corregirla e ingresarla nuevamente.

### Guardar y cargar en el REPL

Si deseas ejecutar pruebas más extensas en el REPL o registrar los resultados, puedes usar los comandos `.save` y `.load` para guardar los comandos ejecutados previamente en un archivo o cargar un archivo con instrucciones de JavaScript en el REPL actual.

El listado 6 muestra cómo puedes usar los comandos `.load` y `.save`. Para ilustrar tu uso, primero debes ingresar el comando `console.log`. Después de eso, la sesión actual se guarda en el archivo `myShell.js`. Este archivo contiene la línea de comando tal como la ingresaste en el REPL, pero no el resultado asociado. A continuación, debes volver a cargar este archivo en la sesión utilizando la declaración `.load`. El archivo se lee línea por línea, se ejecuta cada comando y se muestra el resultado correspondiente.

También puedes usar el comando `.load` para preparar una situación inicial específica para un experimento. Para ello, puedes formular un conjunto de comandos en un archivo, por ejemplo, para definir variables o funciones que necesites en el transcurso de una sesión en el REPL de Node.js.

Listado 6: Uso de “`.load`” y “`.save`”.

```
> console.log('Hello World!');
Hello World!
undefined
> .save myShell.js
Session saved to: myShell.js
> .load myshell.js
console.log('Hello World!');
Hello World!
undefined
```

## Contexto del REPL

Como es habitual en JavaScript, el REPL de Node.js proporciona un contexto global al que puede acceder desde cualquier parte de tu programa. En el REPL, algunas variables ya están registradas al principio en este contexto global, lo que facilita tu trabajo como desarrollador. De esta manera, todos los módulos principales de Node.js están disponibles para ti sin tener que cargarlos por separado a través del sistema de módulos. Por ejemplo, puedes utilizar `http.STATUS_CODES` para imprimir la lista predefinida de códigos de estado HTTP.

Además de estos módulos, puedes cargar archivos o paquetes de Node Package Manager (npm) a través del sistema de módulos utilizando la función `require`. En el ámbito global del REPL de Node.js, la variable `_` te proporciona otra característica especial: esta variable siempre contiene el valor del comando más reciente. Por ejemplo, si ingresas el comando `1 + 1`, `_` contendrá el valor 2 después. Si llamas a un método como `process.uptime()`, `_` contendrá su valor de retorno después.

## Historial de REPL

El REPL de Node.js tiene algunas variables de entorno especiales. Dos de ellas se relacionan con la historicización de las entradas. Probablemente ya conozcas esta característica del símbolo del sistema de tu sistema operativo. En el caso de Bash en sistemas Unix, hay un archivo llamado `.bash_history`, que almacena todos los comandos que se ingresaron. Existe una funcionalidad similar para el REPL de Node.js. En la configuración predeterminada, la entrada se almacena en el archivo `.node_repl_history` en el directorio raíz del usuario.

Puedes usar dos variables de entorno de tu sistema operativo para controlar la funcionalidad del historial. Con `NODE_REPL_HISTORY`, puedes cambiar la ubicación del historial. Si no se especifica ningún valor, se usa el valor predeterminado. La segunda variable de entorno, `NODE_REPL_HISTORY_SIZE`, determina cuántas líneas puedes contener el archivo de historial antes de sobrescribir los comandos anteriores. El valor predeterminado es 1000.

## Modo REPL

Puedes utilizar la variable de entorno `NODE_REPL_MODE` para determinar en qué modo deseas ejecutar el REPL de Node.js. Los tres valores posibles son los siguientes:

- **sloppy**  
El REPL se establece en modo no estricto. Se anulan las reglas del modo estricto de JavaScript. Este es el modo predeterminado para ejecutar el REPL.
- **strict**  
El valor `strict` activa el modo estricto. En este caso, por ejemplo, ya no puedes crear varias propiedades con el mismo nombre en un objeto y cambiar una constante devuelve un error. Puedes encontrar una descripción muy buena y detallada del modo estricto en [https://developer.mozilla.org/enUS/docs/Web/JavaScript/Reference/Strict\\_mode](https://developer.mozilla.org/enUS/docs/Web/JavaScript/Reference/Strict_mode).
- **magic**  
Este valor ahora está obsoleto y se utiliza como un alias de `sloppy`.

También puedes ejecutar Node.js directamente en modo estricto a través de la opción de línea de comandos `--use_strict`. En la mayoría de los casos, esto tiene sentido porque puede ahorrar el uso de especificaciones estrictas, y el modo estricto prohíbe muchos antipatrones en JavaScript.

Al usar el REPL de Node.js como herramienta, puedes probar fácilmente el código fuente para ver de forma interactiva cómo se comporta la plataforma Node.js en determinadas situaciones. El REPL de Node.js no es adecuado para aplicaciones extensas. En este caso, se utiliza una forma diferente de ejecutar Node.js.

## Búsqueda en el REPL

El REPL de Node.js te permite no solo navegar dentro del historial usando las teclas de flecha, sino también buscar líneas específicas. Puede usar `(Ctrl) + (R)` y `(Ctrl) + (S)` para realizar ejecuciones de búsqueda hacia atrás y hacia adelante en el historial.

Accedes a esta función principalmente cuando deseas ejecutar una determinada línea nuevamente pero no deseas ingresarla una segunda vez. La búsqueda encuentra el código ingresado ya sea que aparezca al principio o en cualquier lugar dentro de la línea. En el símbolo del sistema se muestra una coincidencia encontrada y tienes la opción de ajustar el código y luego ejecutarlo presionando la tecla (Enter). Si se encuentra un texto determinado varias veces, puedes saltar al siguiente resultado presionando nuevamente el atajo correspondiente o ir al resultado anterior presionando el otro atajo.

## Operaciones asíncronas en REPL

JavaScript proporciona varios medios para manejar la asincronía, como devoluciones de llamadas o promesas. Hace unos años, se introdujo el concepto `async-await`, que se analizará con mayor detalle más adelante en este curso. Puedes usar la palabra clave `await` para esperar una operación asíncrona sin registrar una función de devolución de llamada. El motor pausa la ejecución del bloque de código actual hasta que el resultado esté disponible. El resto de la aplicación sigue respondiendo y puedes continuar tu trabajo sin interrupciones.

Normalmente, debes marcar una función en la que deseas usar la palabra clave `await` con la palabra clave `async`. Las versiones más nuevas del estándar ECMAScript ofrecen una función llamada `await` de nivel superior, que te permite usar `await` incluso en el nivel superior de tu aplicación y, por lo tanto, sin una función asíncrona circundante. Esta función también está disponible en el REPL de Node.js y, desde la versión 16.6, también está habilitada de forma predeterminada y ya no está oculta detrás del indicador `—experimental-repl-await`.

El listado 7 muestra el ejemplo de un objeto de promesa creado con `Promise.resolve` para demostrar cómo puedes iniciar REPL con la opción `—experimental-repl-await` y cómo puedes usar la palabra clave `await`.

`Promise.resolve` es una de las formas más simples de simular una operación asíncrona mediante la creación de un objeto de promesa y su resolución inmediata.

Listado 7: Uso de un `Await` de nivel superior en el REPL.

```
> await Promise.resolve('Hello world!');  
'Hello world!'
```

## La primera aplicación

Cuando creas aplicaciones con Node.js, el código fuente de esa aplicación está contenido en uno o más archivos.

Al iniciar la aplicación, especifica el nombre del archivo inicial como una opción en la línea de comandos. A partir de este archivo, carga el resto de los componentes de tu aplicación a

través del sistema de módulos de Node.js. El motor de JavaScript lee y optimiza todo el código fuente. Como consecuencia, los cambios en el código fuente no afectan directamente a la aplicación en ejecución. Esto significa que debes salir y reiniciarla para que los cambios se activen. El listado 8 muestra cómo puedes ejecutar una aplicación con Node.js.

Listado 8: Ejecución de aplicaciones de Node.js.

```
PS C:\Users\abrah> node myshell.js
Hello World!
PS C:\Users\abrah>
```

El archivo myshell.js contiene solo la línea `console.log('Hello World!')`. La ventaja de este método de ejecución de una aplicación es que puedes ejecutar la aplicación tantas veces como desees y todo lo que necesitas hacer es ejecutar la línea de comandos del listado 8. Además, puedes ejecutar fácilmente el código fuente en otros sistemas o poner la aplicación a disposición de otros usuarios como software de código abierto.

---

### Sistemas de módulos de Node.js

Actualmente, Node.js admite dos sistemas de módulos diferentes. La implementación anterior, - el sistema de módulos CommonJS -, utiliza `module.exports` para exportar elementos y la función `require` para cargar elementos. Este sistema de módulos ha sido parte de la plataforma Node.js desde el principio.

Mientras tanto, otro sistema de módulos, el sistema de módulos ECMAScript, también se ha incluido en el ECMAScript estándar. Este sistema utiliza las palabras clave `import` y `export` y no es compatible con el sistema de módulos CommonJS.

Desde hace varios años, Node.js ha estado en una fase de transición desde el sistema de módulos CommonJS hacia el sistema de módulos ECMAScript.

Para que los módulos ECMAScript funcionen, es necesario que los archivos tengan la extensión `.mjs`. A lo largo de este curso, utilizaremos el sistema de módulos ECMAScript. Sin embargo, en el siguiente documento, también aprenderás más sobre el sistema de módulos CommonJS y cómo puedes trabajar con ambos sistemas.

---

Ten en cuenta que la salida de Hello world! aún no representa una aplicación. Sin embargo, con Node.js, puedes crear aplicaciones web dinámicas sin la necesidad de un servidor web independiente porque Node.js te permite crear tu propio servidor web a través del módulo `http` (en el listado anterior no usamos un archivo con extensión `.mjs`). Las siguientes secciones te guiarán paso a paso a través del ejemplo clásico de una aplicación Node.js: un servidor web muy liviano.

### Servidor web en Node.js



El servidor web que desarrolles en este ejemplo debe poder aceptar solicitudes de los navegadores y responderlas con una respuesta HTTP correcta y la salida de la cadena Hello world.

Comienza con un framework básico y lo amplías hasta que la aplicación cumpla con los requisitos. El listado 9 contiene la estructura básica de la aplicación. Debes guardar este código en un archivo independiente llamado server.mjs.

Listado 9: Estructura de un servidor web en Node.js.

```
import { createServer } from 'http';

const server = createServer();
server.listen(8080, () => {
  console.log(
    `Server is listening to
    http://localhost:${server.address().port}`
  );
});
```

---

### Plantilla de cadenas (Strings)

Con plantilla de cadenas, como las que se usan en el listado 9, existe una tercera forma de definir cadenas en JavaScript además de las comillas simples y dobles.

Sin embargo, se trata de una forma especial cuyo procesamiento es algo más lento que el de las cadenas simples. Con plantilla de cadenas definidas con el carácter de comillas invertidas (```), puedes usar `${}` para reemplazar variables o evaluar las expresiones de JavaScript dentro de una cadena. También es posible insertar saltos de línea sin concatenar la cadena con el operador `+`.

---

En realidad, un motor de JavaScript como V8 no tiene las capacidades para proporcionar un servidor web de una manera sencilla.

Por esta razón, existen módulos para Node.js que amplían la funcionalidad de la plataforma Node.js. En este ejemplo, necesitas la funcionalidad de un servidor web. Existe un módulo independiente para Node.js para este propósito y para resolver otras tareas relacionadas con HTTP.

La funcionalidad de los diversos módulos de Node.js está automáticamente disponible para ti. Como desarrollador, todo lo que necesitas hacer es cargar los módulos que necesitas para tu aplicación antes de usarlos. La carga de módulos y otros archivos en Node.js se realiza a través de la palabra clave `import`. Como puedes ver en el ejemplo, puedes abordar directamente las partes de la interfaz del módulo respectivo que necesitas para tu aplicación.

var, let y const

---

### **var, let y const**

En JavaScript, ahora hay tres formas disponibles para definir variables. Cada una de estas formas tiene un impacto en la forma en que desarrollas tus aplicaciones. Si defines tus variables anteponiéndoles la palabra clave var, esto tiene el efecto de que la variable es válida en la función actual y todas las subfunciones. Durante mucho tiempo, esta fue la única opción disponible.

La palabra clave let te permite definir variables a nivel de bloque. Por ejemplo, si defines una variable de contador en un ciclo for con let, esta variable es válida solo dentro del ciclo. Una variable definida con let en una función tiene las mismas propiedades que una definida con var. let tiene el mismo rango de funciones que var, excepto que tiene un mejor control sobre el alcance. En realidad, no hay muchas razones para seguir usando var. Sin embargo, ten cuidado de no mezclar var y let en tu aplicación, ya que esto puede provocar fácilmente errores que son difíciles de localizar.

La tercera forma de definir una variable es con la palabra clave const. Estas variables no son variables en el sentido estricto, sino constantes, lo que significa que no puedes cambiar el valor de la variable después de la asignación inicial. El hecho de que JavaScript funcione con referencias para valores no primitivos, como objetos o arreglos, vuelve a poner todo en perspectiva.

Con un objeto const, ya no puedes cambiar la referencia, pero puedes cambiar las propiedades del objeto en sí. Intenta usar const tanto como sea posible durante el desarrollo para evitar sobrescribir variables por error. Si realmente necesitas variables, debe usar let.

---

Con el módulo http, puedes crear un cliente para consultar otros servidores web además del servidor inicial.

Sin embargo, para tu aplicación de servidor web, solo necesitas el servidor. Puedes crearlo con la función createServer importada del módulo http. El valor de retorno de este método es el servidor HTTP, que está disponible para ti como un objeto para su uso posterior. El objeto de servidor recién creado no tiene actualmente ninguna funcionalidad, ni se ha abierto una conexión con el mundo exterior.

El siguiente paso es abrir esta misma conexión hacia los clientes para que puedan conectarse al servidor y recuperar datos. El servidor te proporciona el método listen, una forma de especificar un puerto y una dirección IP a través de los cuales sus usuarios pueden conectarse al servidor. Puedes pasar el número de puerto y la dirección IP a la que debe estar vinculado el servidor al método listen.

Normalmente, sin embargo, deberías especificar al menos el número de puerto, ya que de lo contrario se asignará cualquier puerto libre. El número de puerto se especifica como un entero, que debe estar entre 0 y 65,535. Hay dos cosas que tener en cuenta al elegir un puerto

para tu servidor web: el puerto utilizado no debe estar ya ocupado por otra aplicación, y no debe estar en el rango de puertos del sistema entre 0 y 1,023. Si utilizas un puerto 1,024 o superior, puedes ejecutar tu script de Node.js como un usuario normal; para puertos por debajo de ese valor, necesitas privilegios de administrador. Siempre debes ejecutar pruebas y ejemplos como un usuario normal porque como administrador tienes considerablemente más privilegios y puedes dañar seriamente el sistema.

Puedes especificar la dirección IP como una cadena, por ejemplo, '127.0.0.1'. Si no especificas ninguna dirección, se utilizará la dirección IPv6 :: o la dirección IPv4 0.0.0.0 de forma predeterminada. Esto significa que el servidor está vinculado a todas las interfaces del sistema. Por ejemplo, puedes acceder a tu servidor utilizando el nombre localhost.

Además de especificar la dirección y el puerto, al llamar al método listen del servidor HTTP se abre la conexión y hace que el servidor espere las solicitudes entrantes. También puedes pasar una función de devolución de llamada al método listen como último argumento. Esto se ejecuta tan pronto como se vincula el servidor. En el ejemplo, esta función muestra la información de que el servidor está listo para funcionar y en qué dirección puedes acceder a él. Si no te encargas de esa salida tú mismo, Node.js no muestra ninguna información adicional. Ahora debes guardar el script con el nombre server.mjs.

Puedes ejecutar el servidor web en tu sistema y probar el resultado. El listado 10 muestra cómo se ve el resultado de la prueba.

Listado 10: Ejecución del servidor web en Node.js.

```
PS C:\Users\abrah\Documents\ProyectosNode> node server.mjs
Server is listening to
http://localhost:8080
```

Si recibes el error create Server: listen EADDRINUSE:::8080 al ejecutar tu aplicación, significa que el puerto ya está ocupado por otra aplicación y debes elegir otro puerto para tu aplicación Node.js.

El comando node con el archivo que contiene el código fuente del servidor web como opción hace que se inicie un proceso Node.js que se conecta a la combinación especificada de dirección y puerto y luego espera las conexiones entrantes que pueda atender. Al ejecutar el script, la línea de comandos se bloquea y no puedes realizar más entradas en ella. Debido a la arquitectura de Node.js, que se basa en el principio impulsado por eventos, el script del servidor web crea muy poca carga porque Node.js no se bloquea cuando no tiene nada que hacer.

Si deseas cancelar el script, puedes hacerlo mediante el acceso directo (Ctrl) + (C), que devuelve un símbolo del sistema. Ahora puedes probar el servidor web con su navegador ingresando “http://localhost:8080” en la barra de direcciones.

El problema es que, aunque el servidor web está vinculado a la dirección y el puerto correctos, no tiene lógica para gestionar las solicitudes entrantes. Esto provoca principalmente que no se obtenga ningún resultado durante las pruebas y, si se deja la ventana del navegador abierta durante el tiempo suficiente, se produce un error de tiempo de espera. Por este motivo, en el siguiente paso se insertará el código fuente para garantizar que las solicitudes entrantes también se atiendan de forma significativa.

El servidor web que se crea aquí en Node.js difiere seriamente en algunas características de otras implementaciones en lenguajes de programación dinámicos como PHP. Aquí, cada solicitud se atiende por separado y el código fuente necesario se lee en el proceso. Con Node.js, el código de la aplicación se lee una vez y luego permanece en la memoria. La aplicación se ejecuta de forma permanente. Aquí es donde entra en juego un aspecto importante de Node.js: el procesamiento asíncrono de tareas. El servidor web responde a los eventos, en este caso, a las solicitudes de los clientes.

Para este fin, se define una función (una devolución de llamada) que se ejecuta tan pronto como se recibe una solicitud. Aunque este código se define en un bloque, como en otros lenguajes, el código fuente responsable de manejar las solicitudes no se ejecuta hasta que se recibe dicha solicitud.

## **Ampliación del servidor web**

En el listado 11 puedes ver el código fuente extendido del listado 9. Esta versión del servidor web también puedes manejar solicitudes correctamente.

Listado 11: Servidor web con devolución de llamada.

```
import { createServer } from 'http';

const server = createServer((request, response) => {
  response.writeHead(200, { 'content-type': 'text/plain; charset=utf-8'
});
  response.write('Hello ');
  response.end(' World\n');
});

server.listen(8080, () => {
  console.log(
    `Server is listening to`
```

```
http://localhost:${server.address().port}`,  
  );  
});
```

La única adaptación del ejemplo del listado 9 tiene lugar en la llamada a la función `createServer`. Aquí, Node.js recibe la devolución de llamada que especifica qué debe suceder cuando una solicitud de un cliente llega al servidor.

En este ejemplo simple, primero solo se muestra la cadena Hello world en el navegador del cliente.

Para lograr este resultado, primero debes observar la estructura de la función de devolución de llamada. Tiene dos parámetros, un objeto de solicitud y un objeto de respuesta, que representan la solicitud del cliente y la respuesta del servidor, respectivamente. En este ejemplo, primero debes ignorar la solicitud del cliente y concentrarte en la respuesta del servidor. El primer paso es preparar la información del encabezado HTTP que luego se enviará de vuelta al cliente. Esto se hace usando el método `writeHead`. El primer argumento de esta función consiste en un número que representa el código de estado HTTP. El segundo argumento es un objeto que contiene el encabezado HTTP real.

Los valores clave del objeto, como el valor `content-type` en el ejemplo, deben escribirse en minúsculas según la convención. En el ejemplo, puedes ver la especificación del tipo de contenido, que en este caso está configurado como `text/plain` para indicar al cliente que la respuesta del servidor contiene solo texto. Debido a que algunos navegadores como Firefox devuelven mensajes de error si la codificación de caracteres no se especifica correctamente, el tipo de contenido se amplía con la información `charset=utf-8` para informar al navegador que el cuerpo HTTP está codificado en UTF-8.

Puedes crear la respuesta visible para el usuario en el cuerpo HTTP, utilizando el método `write`. Cuando se llama a este método, se envían fragmentos de la respuesta, llamados 'chunks'. Puedes llamar a este método varias veces seguidas, lo que dará como resultado que las partes individuales se unan. Sin embargo, debes asegurarte de que antes de llamar a `write`, siempre se envíen los encabezados HTTP correctos con `writeHead`. Si no llamas al método `writeHead`, el servidor HTTP de Node.js envía implícitamente un encabezado HTTP con un código de estado de 200 y un tipo de contenido de `text/plain`, por lo que una respuesta sin proporcionar explícitamente información del encabezado también es válida. Llamar a `write` garantiza que se envíen partes de la respuesta al cliente.

Sin embargo, en este caso, el cliente no sabe cuándo el servidor termina de enviar la respuesta. Tú, como desarrollador, debes encargarte de esto utilizando el método `end` del objeto de respuesta (`response`). Opcionalmente, puedes proporcionar una cadena como

argumento. En este caso, end se comporta de la misma manera que write, ya que envía el fragmento especificado al cliente y luego finaliza la respuesta.

El método write tiene otras dos características que vale la pena mencionar. Por un lado, no solo puedes pasar cadenas como argumentos, sino también como objetos buffer. Un objeto buffer consiste en datos binarios que facilitan enormemente la transmisión de datos. Esta clase de objetos entra en juego principalmente cuando se utilizan transmisiones. La segunda característica consiste en especificar la codificación de la cadena, es decir, a través del segundo parámetro del método write. Esto es opcional y, si se omite, Node.js usa utf-8 como método de codificación predeterminado. Otros valores posibles son utf16le, ascii o hex. UTF-8 como método de codificación está permitido en este ejemplo, por lo que tampoco es necesario especificar una codificación de caracteres. Como alternativa a la combinación de múltiples llamadas write y una llamada end, también puedes almacenar toda la respuesta al cliente en una variable y enviarla en una sola llamada del método end.

Para que el ejemplo funcione correctamente, debes asegurarte de reiniciar el servidor web para que el código fuente personalizado de Node.js se lea correctamente. Para ello, lo mejor es finalizar la instancia del primer ejemplo que posiblemente aún esté en ejecución mediante el atajo (Ctrl) + (C) y reiniciar el servidor web llamando nuevamente al comando node con el nombre del archivo de tu código fuente. La figura 2 muestra el resultado de la solicitud.

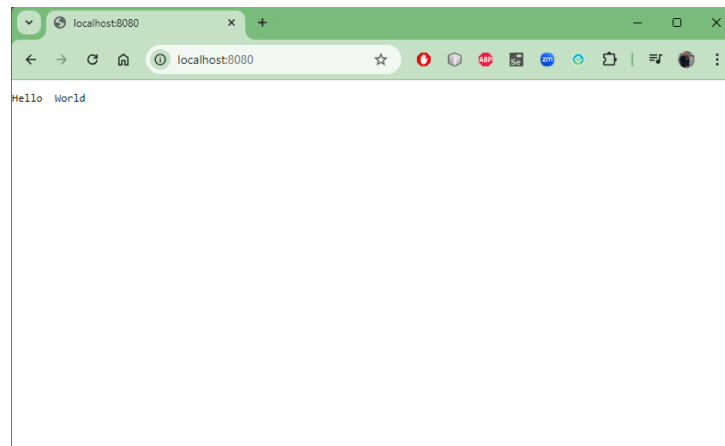


Figura 2: Respuesta del servidor web al cliente.

En este ejemplo, has visto cómo es posible, con solo unas pocas líneas de código JavaScript, crear un servidor web funcional que responda a la solicitud de un cliente con una respuesta HTTP correcta.

### Creación de una respuesta HTML

Sin embargo, en la realidad, rara vez tienes que lidiar con respuestas de servidores web en texto sin formato. Por lo tanto, ahora ampliaremos el ejemplo para que el servidor responda con una respuesta en HTML, tal como lo haría un servidor web normal. El listado 12 muestra los ajustes que debes realizar para esto.

Listado 12: Respuesta HTML del servidor web.

```
import { createServer } from 'http';

const server = createServer((request, response) => {
  response.writeHead(200, { 'content-type': 'text/html; charset=utf-8'
});

const body = `<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Node.js Demo</title>
</head>
<body>
<h1 style="color:green">Hello World</h1>
</body>
</html>`;

response.end(body);
});

server.listen(8080, () => {
  console.log(
    `Server is listening to
http://localhost:${server.address().port}`,
  );
});
```

El único cambio que debes realizar en el código fuente del ejemplo es ajustar el tipo de contenido, que ahora es text/html en lugar de text/plain. Además, se eliminó la escritura y el cuerpo HTTP se envía completamente utilizando el método end. El valor pasado al método end contiene una cadena HTML que refleja la estructura de la página web que se mostrará. Como esta cadena es bastante grande, es mejor externalizarla a la constante de cuerpo independiente y usar una plantilla de cadena para crear una cadena clara de varias líneas con medios simples. Finalmente, debes pasar la constante de cuerpo (body) al método final. Una vez que haya realizado estos cambios, todo lo que necesitas hacer es reiniciar el proceso Node.js que ejecuta tu servidor web para que los cambios surtan efecto.

Cuando vuelvas a cargar la página en tu navegador, deberías ver un resultado similar al que se muestra en la figura 3.

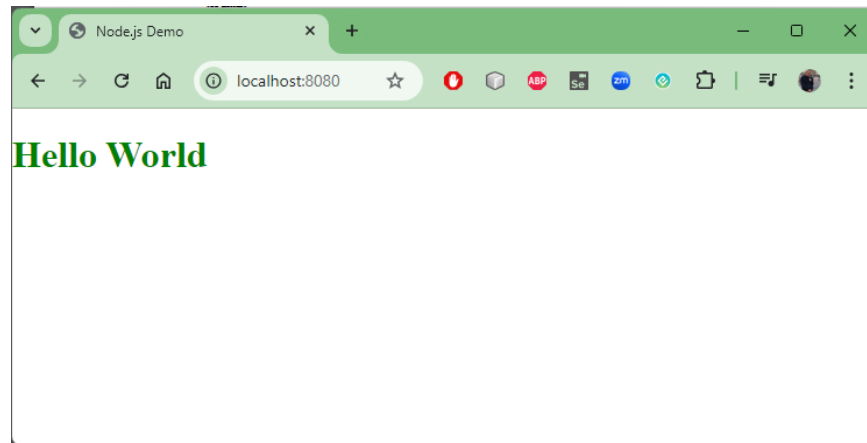


Figura 3: Salida de una página HTML.

Hasta este punto, te has preocupado principalmente por el objeto de respuesta, es decir, la respuesta al cliente. Ahora es el momento de analizar más de cerca el objeto de solicitud, que es la solicitud real. Este objeto te permite leer la información enviada por el cliente e incluirla en la generación de la respuesta.

### Generación de respuestas dinámicas

En las aplicaciones web clásicas, la información se envía desde el navegador utilizando los métodos HTTP GET y POST. En su mayoría, esto se hace a través de formularios o parámetros que están codificados en la URL. Ahora ampliarás el ejemplo y mostrarás una cadena especificada por el usuario en la URL en la página de salida.

Puedes utilizar el código fuente del listado 12 como base. El código fuente adaptado se muestra en el listado 13, seguido de las explicaciones correspondientes de los cambios.

Listado 13: Manipulación de páginas web mediante parámetros.

```
import { createServer } from 'http';

const server = createServer((request, response) => {
  response.writeHead(200, { 'content-type': 'text/html; charset=utf-8'
});

const url = new URL(request.url, 'http://localhost:8080');

const body = `<!DOCTYPE html>
<html>
```



```

    <head>
      <meta charset="utf-8">
      <title>Node.js Demo</title>
    </head>
    <body>
      <h1 style="color:green">Hello ${url.searchParams.get('name')}
      </h1>
    </body>
  </html>`;

  response.end(body);
});
server.listen(8080, () => {
  console.log(
    `Server is listening to
    http://localhost:${server.address().port}`,
  );
});

```

La adaptación más importante del código fuente es que lee la URL que el cliente ha solicitado en el código fuente y escribe partes de ella en la respuesta. En el objeto de solicitud, la información sobre qué URL especificó el usuario en su navegador está presente en la propiedad `url`.

Por ejemplo, si supones que el usuario escribió la URL `http://localhost:8080/?name=Besides` en la barra de direcciones de tu navegador, la propiedad `url` del objeto de solicitud contiene el valor `/name=visitor`.

Tu objetivo ahora es generar la cadena de caracteres `Hello visitor`. Para ello, debes extraer la cadena, en este caso `visitor`, de la propiedad `url`. Puedes hacerlo, por ejemplo, dividiendo la cadena con la función de cadena de JavaScript `split` en el signo igual y utilizando el segundo elemento del arreglo resultante. Sin embargo, esta variante solo funciona siempre que el usuario pase solo un parámetro en la URL o este parámetro esté en la primera posición en el caso de varios parámetros.

Una mejor manera de manejar las URL es utilizar la API de URL WHATWG, que ahora es una parte nativa de la plataforma Node.js; por lo tanto, no se requiere una importación separada para la clase URL. Entre otras cosas, esta API te permite analizar las URL y, por lo tanto, descomponerlas en sus componentes individuales. Para ello, debes crear una nueva instancia de la clase URL y pasar al constructor la ruta relativa de la llamada contenida en la propiedad `url` del objeto de solicitud y la URL base de tu aplicación, en este caso, `http://localhost:8080/`. El objeto recién creado representa la URL solicitada con todos sus componentes. Puedes encontrar los parámetros de consulta individuales que se transfirieron

en la propiedad `searchParams`. Puedes leerlos utilizando el método `get` y transferir el nombre del parámetro deseado. En nuestro ejemplo, se trata de la cadena `name`. Aquí, el número y el orden de los parámetros en la URL ya no juegan ningún papel, ya que puedes acceder al valor a través del nombre del parámetro. Por lo tanto, puedes acceder a la cadena que el usuario ha ingresado en la barra de direcciones del navegador a través de `url.searchParams.get('name')`. Esto significa que tienes todos los componentes que necesitas para lograr tu objetivo.

Después de haber realizado los ajustes en el código fuente y reiniciado el servidor web, puedes probar el resultado llamando a la página nuevamente. La figura 4 muestra el resultado que obtienes cuando llamas a la página `http://localhost:8080/?name=user`.

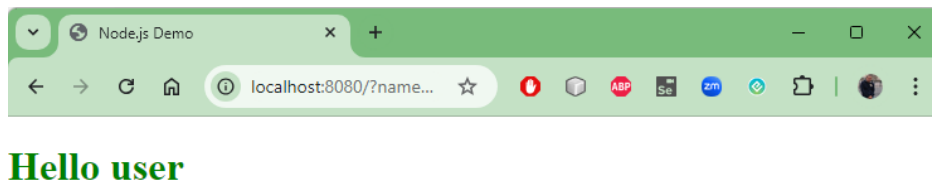


Figura 3.4 El primer sitio web dinámico en Node.js.

## Depuración de aplicaciones Node.js

Incluso al implementar aplicaciones más pequeñas, tarde o temprano, llegarás al punto en el que necesitarás encontrar y corregir un error en tu propio código fuente. La mayoría de estos errores son causados por variables asignadas incorrectamente o errores lógicos dentro de la aplicación. Por este motivo, al solucionar problemas, normalmente te preocupan los valores de ciertas variables y el flujo de la lógica de la aplicación.

En el caso más simple, insertas instrucciones `console.log` en ciertos puntos del código fuente, lo que te ayuda a generar los valores de las variables y a verificar los pasos de flujo individuales de tu código fuente. En el listado 14, puedes ver cómo se puede ver el ejemplo del servidor web con esas salidas de depuración. Además, como puedes ver en el código, `console.log` es bastante flexible. Puedes pasar uno o más argumentos a este método. Si pasas varios argumentos, `console.log` se encarga de la unión y la salida.

## Listado 14: Depuración con “console.log”

```
import { createServer } from 'http';

console.log('createServer');
const server = createServer((request, response) => {
  console.log('createServer callback');

  response.writeHead(200, { 'content-type': 'text/html; charset=utf-8'
});

  const url = new URL(request.url, 'http://localhost:8080');
  console.log(url);

  console.log('Name: ', url.searchParams.get('name'));

  const body = `...`;
  response.end(body);
});

console.log('listen');
server.listen(8080, () => {
  console.log(
    `Server is listening to
http://localhost:${server.address().port}`,
  );
});
```

Cuando ejecutes el código fuente del listado 14, verás que primero `createServer`, `listen` y la cadena `Server is listening to http://localhost:8080` se envían a la consola. Tan pronto como inicies una consulta con tu navegador, se envían a la consola la cadena de devolución de llamada `createServer`, el objeto URL y el nombre de la cadena de consulta.

Esta forma de depurar una aplicación tiene varios problemas al mismo tiempo. Para recibir la salida, debes editar activamente el código fuente e incluir las declaraciones en los lugares relevantes. Esto significa que tendrás que realizar ajustes adicionales al código fuente para el análisis de errores y, por esta razón, tendrás que modificarlo aún más, lo que lo hace vulnerable a problemas adicionales. Otra dificultad de este método de depuración es que al usar las declaraciones `console.log`, en el tiempo de ejecución de la aplicación, no obtienes una imagen de todo el entorno sino solo de una sección muy específica, que puede distorsionarse aún más por ciertas influencias.

El depurador de Node.js ofrece una solución para estos problemas. La ventaja del depurador es que es una parte integral de la plataforma Node.js, por lo que no necesitas instalar ningún

otro software. Para iniciar el depurador, guarda el código fuente en un archivo, que en este ejemplo es el archivo `server.mjs`, con el código fuente del listado 13, y ejecute el comando `node inspect server.mjs` en la línea de comandos. El listado 15 muestra el resultado del comando.

#### Listado 15: Depuración con Node.js.

```
PS C:\Users\abrah\Documents\ProyectosNode> node inspect server.mjs
< Debugger listening on ws://127.0.0.1:9229/bc7293ad-8f5f-4114-b1d7-fa3e8c2e3be5
< For help, see: https://nodejs.org/en/docs/inspector
<
connecting to 127.0.0.1:9229 ... ok
< Debugger attached.
<
Break on start in server.mjs:1
> 1 import { createServer } from 'http';
  2
  3 const server = createServer((request, response) => {
debug>
```

Si agregas la opción `inspect` al llamar a tu aplicación, la aplicación se iniciará en modo de depuración interactiva. La primera información que obtienes es que el depurador está esperando conexiones entrantes a través de una conexión WebSocket. Además, la pantalla te indica en qué archivo y línea el depurador ha interrumpido la ejecución.

Finalmente, se muestran las primeras tres líneas del código de la aplicación. La declaración en la que se detuvo el depurador se resalta en verde.

### Navegación en el depurador

El mensaje `debug>` indica que el depurador está esperando su entrada. Puedes ver qué comandos están disponibles en la tabla 2.

En el ejemplo que se muestra en el listado 15, se utiliza el comando `n` o `next`, respectivamente, para saltar a la siguiente instrucción con el depurador. Esto se indica por el hecho de que la función `createServer` está resaltada en verde en la línea 3.

### Información en el depurador

Si avanzas en el depurador hasta la segunda línea, puedes visualizar el valor de la constante `http`. Para ello, debes introducir el comando `repl` en el depurador. Este comando inicia un shell interactivo desde el que puedes acceder al entorno del depurador. Por ejemplo, puedes introducir la cadena `http` y obtener la estructura de esta variable. El atajo (Ctrl) + (C) te permite cambiar del shell interactivo al depurador.

Tabla 2: Comandos para el depurador.

Comando	Descripción	Descripción
c	Continue	Continúa la ejecución de la aplicación hasta el siguiente punto de interrupción
n	Step next	Omite una subrutina
Add	Step in	Salta a una subrutina
o	Step out	Salta de una subrutina al siguiente nivel superior
pause	Pause	Pausa la ejecución

Después de eso, puedes utilizar los comandos de la tabla 2 para navegar como de costumbre. Si has creado una salida en el shell interactivo y luego has vuelto al modo de depuración, ya no puedes ver dónde se encuentra en tu código fuente. El comando `list` te ayuda a resolver este problema. Esta función garantiza que el depurador te muestre la línea de código fuente que se está ejecutando actualmente y una determinada cantidad de líneas antes y después de esta línea. Puedes especificar la cantidad de líneas que deseas ver como argumento. Si no especificas un número, se asume el valor 5. El listado 16 muestra un ejemplo de cómo utilizar el comando `list`.

Listado 16: Función “list” del depurador.

```
debug> list (1)
> 1 import { createServer } from 'http';
2
```

Otra función que te ofrece el depurador es la salida de un backtrace. Esto es especialmente útil si has saltado a varias subrutinas utilizando el comando `s` y ahora deseas averiguar cómo llegó a la ubicación actual. La función backtrace, o su variante más corta `bt`, te permite mostrar el backtrace de la ejecución actual. El listado 17 muestra la salida del backtrace en caso de que hayas saltado a la función de devolución de llamada de la función `createServer` en el ejemplo del servidor web.

Listado 17: Backtrace con el depurador.

```
debug> bt
#0 (anonymous) server.mjs:1:0
#1 _instantiate node:internal/modules/esm/module_job:132:8
```

Cuando recorres el código fuente de tu aplicación con el depurador, normalmente te interesan los valores de ciertas variables. Puedes determinar fácilmente estos valores a través del comando `repl` y el shell interactivo. Sin embargo, en muchos casos, esta variante no es muy práctica porque hay que volver al shell cada vez y solo se pueden leer los valores allí.

Otra forma de averiguar los valores de las variables es utilizar la función `watch` del depurador. En este caso, se pasa el nombre de la variable cuyo valor se desea observar como una cadena a esta función. Si ahora se recorre la aplicación, se verá el valor de esta variable en cada paso.

El listado 18 muestra cómo se puede utilizar la función `watch`. La salida de la estructura en este caso solo proporciona la pista de que la expresión `watch` es la función `createServer`. Sin embargo, no solo se pueden supervisar las estructuras nativas, sino también observar las variables durante la depuración. Pero si se establece un observador en un objeto más extenso con muchas propiedades, la salida no está estructurada.

Por lo tanto, los observadores ofrecen una ventaja principalmente para las variables con valores escalares u objetos pequeños. Para objetos más extensos, recomendamos utilizar el comando `repl`.

Listado 18: Observador con Node.js.

```
debug> watch('createServer')
debug> n
break in server.mjs:3
Watchers:
0: createServer = [Function: createServer]

1 import { createServer } from 'http';
2
> 3 const server = createServer((request, response) => {
4   response.writeHead(200, { 'content-type': 'text/html'; charset=utf-8'
5 });
```

Además de la función `watch`, hay otras dos funciones que puedes utilizar para supervisar estructuras. La función `unwatch` te permite eliminar observadores que ya se han configurado. Para ello, solo tienes que pasar el nombre de la variable como una cadena a la función y el observador se eliminará. La segunda función `watchers` no toma argumentos y enumera los observadores existentes y los valores asociados de las variables correspondientes. La tabla 3 resume nuevamente los comandos del depurador.

### Puntos de interrupción (breakpoints)

Con el estado actual, solo es posible recorrer el código fuente de tu aplicación paso a paso desde el principio hasta el punto en el que sospechas que hay un problema. Esto puede ser difícil con aplicaciones extensas, ya que puede llevar mucho tiempo llegar a la ubicación relevante. Para estos casos, el depurador de Node.js proporciona la función de puntos de interrupción. Un punto de interrupción representa un marcador en el código fuente en el que el depurador se detiene automáticamente.

Tabla 3: Comandos en el depurador.

Comando	Descripción
repl	Abre un shell interactivo en el depurador
list, list (n)	Muestra el código fuente actual del depurador
backtrace, bt	Genera el backtrace de la ejecución actual
watch (exp)	Muestra el valor de la expresión especificada en cada paso del depurador
unwatch (exp)	Elimina un observador
watchers	Enumera todos los observadores activos

Con la función `setBreakpoint` (o `sb` en su forma abreviada) del depurador, puedes definir un punto de interrupción. En el caso del listado 19, se utiliza `setBreakpoint` para establecer un punto de interrupción en la línea 7 al comienzo de la ejecución. Puedes utilizar el comando `c` para continuar la ejecución, que luego se detiene en el punto de interrupción.

`setBreakpoint` te permite establecer puntos de interrupción de varias maneras diferentes. Como ya has visto, puedes especificar una línea particular en la que deseas establecer el punto de interrupción. Si no especificas un valor, el punto de interrupción se establece en la línea actual. Además, puedes especificar una función como una cadena en cuya primera instrucción se detendrá la ejecución. Por último, en la última variante, especifica un nombre de archivo y un número de línea. Cuando la ejecución de la aplicación llega a este archivo y línea, la ejecución se interrumpirá. El comando `clearBreakpoint` o `cb` te permite eliminar un punto de interrupción establecido especificando el nombre de archivo y el número de línea.

#### Listado 19: Puntos de interrupción en Node.js.

```
PS C:\Users\abrah\Documents\ProyectosNode> node inspect server.mjs
< Debugger listening on ws://127.0.0.1:9229/29320364-3712-407b-95c2-9f21597ed396
< For help, see: https://nodejs.org/en/docs/inspector
<
connecting to 127.0.0.1:9229 ... ok
< Debugger attached.
<
Break on start in server.mjs:1
> 1 import { createServer } from 'http';
  2
  3 const server = createServer((request, response) => {
debug> setBreakpoint(8)
  4 response.writeHead(200, { 'content-type': 'text/html'; charset=utf-8'
  5 });
  6
```

```

7  const url = new URL(request.url, 'http://localhost:8080');
8
> 9  const body = `<!DOCTYPE html>
10    <html>
11    <head>
12      <meta charset="utf-8">
13      <title>Node.js Demo</title>
14    </head>
debug> c
< Server is listening to
< http://localhost:8080
<
debug>

```

Alternativamente, puedes establecer un punto de interrupción directamente en tu código fuente insertando la declaración del depurador. Sin embargo, la desventaja es que debes modificar tu código fuente para la depuración. En cualquier caso, debes asegurarte de eliminar todas las declaraciones del depurador después de la depuración. El listado 20 muestra cómo puedes usar la declaración del depurador en el ejemplo del servidor web.

Listado 20: Uso de la declaración “depurador”.

```

import { createServer } from 'http';

const server = createServer((request, response) => {
  response.writeHead(200, { 'content-type': 'text/html'; charset=utf-8'
});

const url = new URL(request.url, 'http://localhost:8080');
debugger;
const body = `<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Node.js Demo</title>
  </head>
  <body>
    <h1 style="color:green">Hello ${url.searchParams.get('name')}
  </h1>
  </body>
</html>`;

  response.end(body);
});
server.listen(8080, () => {

```



```
console.log(  
  `Server is listening to  
http://localhost:${server.address().port}`,  
);  
});
```

Si inicias tu script ahora, como se muestra en el listado 21, y continúas la ejecución utilizando el comando `c`, la aplicación está lista para aceptar solicitudes de los clientes.

Listado 21: Ejecución de una aplicación en modo de depuración.

```
PS C:\Users\abrah\Documents\ProyectosNode> node inspect server.mjs  
< Debugger listening on ws://127.0.0.1:9229/cda28c74-6357-4e87-94e1-6d0e7be6086f  
< For help, see: https://nodejs.org/en/docs/inspector  
<  
connecting to 127.0.0.1:9229 ... ok  
< Debugger attached.  
<  
Break on start in server.mjs:1  
> 1 import { createServer } from 'http';  
  2  
  3 const server = createServer((request, response) => {  
debug> c  
< Server is listening to  
< http://localhost:8080  
<  
debug>
```

Si ahora accedes al servidor web a través de un navegador web utilizando la URL `http://localhost:8080/name=user`, la ejecución se interrumpirá dentro de la función de devolución de llamada.

Luego, estarán disponibles las características familiares del depurador.

Si inicias tu aplicación en modo de depuración, se ejecutará hasta que el depurador alcance el primer punto de interrupción. Durante la ejecución inicial, solo tienes la opción de establecer un punto de interrupción a través de una declaración del depurador en el código.

La opción `—inspect-brk` brinda ayuda aquí al garantizar que el depurador se detenga en la primera línea; luego puedes conectarte a tus herramientas de desarrollador, establecer puntos de interrupción y ejecutar la aplicación.

## Depuración con las herramientas de desarrollador de Chrome

El inspector V8 está integrado en el depurador de Node.js y es responsable de abrir un WebSocket al mundo exterior a través del cual puedes conectar varias herramientas a tu sesión de depuración. Esto te permite no solo depurar en la línea de comandos, sino también usar herramientas gráficas. La variante más conveniente es usar Chrome en este contexto. La conexión se establece a través del Protocolo Chrome DevTools.

En lugar de iniciar el depurador a través de la opción `inspect` como antes, debes usar la opción `—inspect` para la depuración remota. El listado 22 muestra el resultado correspondiente.

Listado 22: Depuración remota.

```
PS C:\Users\abrah\Documents\ProyectosNode> node --inspect server.mjs
Debugger listening on ws://127.0.0.1:9229/f2d7e50e-6cb9-4181-a2b7-491cd715ac7a
For help, see: https://nodejs.org/en/docs/inspector
Server is listening to
http://localhost:8080
```

Como puedes ver en el resultado, la aplicación no se detiene; solo espera conexiones de depuración entrantes, así como solicitudes de cliente regulares. Para poder conectar tu navegador al proceso de depuración en ejecución, ahora debes ingresar `"chrome://inspect"` en la barra de direcciones del navegador. Luego verás una descripción general de todos los destinos remotos disponibles, como se muestra en la figura 5.

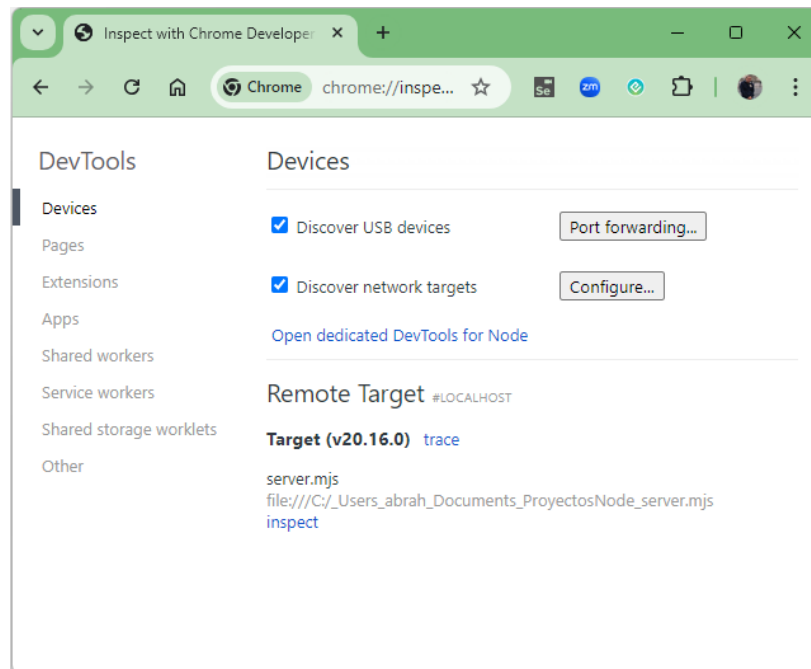


Figura 5: Lista de sesiones de depuración en Chrome.

Cuando ahora hagas clic en el enlace inspect, se abrirá otra ventana del navegador, que consta solo de las herramientas para desarrolladores.

Estas están asociadas con tu proceso Node.js. Ahora verás todos los mensajes de la consola también en la pestaña Consola de Chrome DevTools. Para inspeccionar tu código fuente, ve a la pestaña Fuentes. Allí puedes hacer selecciones de los archivos de tu aplicación en el panel de la izquierda.

Al hacer clic en uno de estos archivos se muestra el código fuente. En este punto, puedes establecer puntos de interrupción haciendo clic en el número de línea correspondiente. Tan pronto como la ejecución de la aplicación llega a esta línea, el proceso se detiene y tienes control sobre el entorno de tiempo de ejecución. La figura 6 muestra un ejemplo de una sesión de depuración de este tipo.

En este modo, al igual que en la consola, puedes crear expresiones de vigilancia, navegar por tu aplicación con el depurador o manipular el entorno a través de la consola. También tienes acceso a todos los ámbitos de variables disponibles y al backtrace actual. Además de estas funciones, puedes usar el generador de perfiles para crear perfiles de CPU y usar la pestaña Memory para analizar la utilización de la memoria de tu aplicación. Explora las opciones.

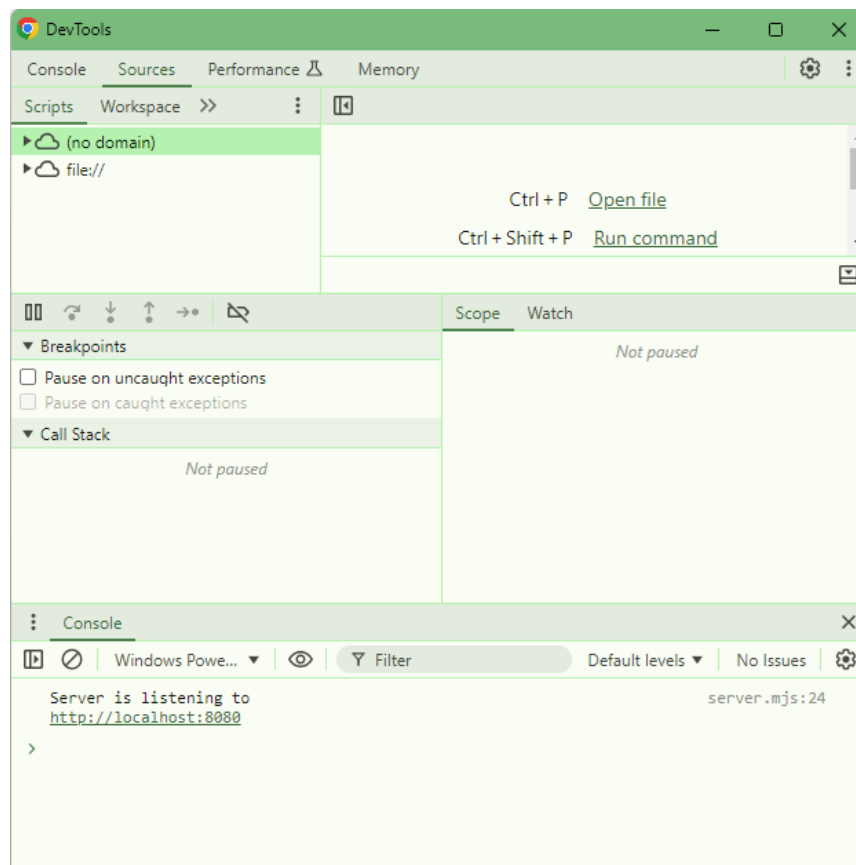


Figura 6: Sesión de depuración activa.

## Depuración en el entorno de desarrollo

Sin embargo, el proceso de desarrollo real no se lleva a cabo en la línea de comandos ni en el navegador, sino en tu entorno de desarrollo. La mayoría de los entornos de desarrollo utilizados en el desarrollo web, como Visual Studio Code o los entornos de desarrollo de JetBrains (por ejemplo, WebStorm), te ayudan a depurar aplicaciones Node.js. Los complementos correspondientes se incluyen de forma predeterminada o se pueden instalar fácilmente más tarde. Para obtener información sobre cómo configurar el depurador en tu entorno de desarrollo, debes consultar la documentación de tu entorno de desarrollo, que generalmente contiene instrucciones detalladas paso a paso.

Si usas el depurador en tu entorno de desarrollo, puedes acceder a las mismas funciones del depurador que ya era posible en la línea de comandos. Por lo tanto, puedes omitir subrutinas o profundizar en las estructuras. Pero también tienes la opción de establecer puntos de interrupción o leer los valores de ciertas variables.

Estas funciones, combinadas con las capacidades de un entorno de desarrollo gráfico, aumentan el nivel de conveniencia durante el desarrollo y mantenimiento de aplicaciones Node.js.

## Herramienta de desarrollo nodemon

Al desarrollar una aplicación Node.js, te encontrarás con un problema específico: debes introducir tu código fuente personalizado en el proceso en ejecución para que los cambios surtan efecto. En el caso más simple, escribes un bloque de código, lo guardas, cambias a la línea de comandos, cancelas el proceso actual usando el atajo (Ctrl) + (C) y lo reinicias. Luego cambias al navegador y compruebas el efecto de los cambios. Si la frecuencia de los cambios es alta, tendrás que realizar estos pasos muchas veces. En este punto, es importante reducir al mínimo las operaciones a realizar. Reiniciar el proceso ofrece especialmente el potencial de automatización. Hoy en día, hay una gran cantidad de herramientas disponibles que pueden hacer este trabajo por ti. Una de las más populares es nodemon.

---

### ¡Atención! No utilices Nodemon en modo de producción

Ten en cuenta que las herramientas como nodemon solo deben usarse durante el desarrollo, pero nunca en producción. Un reinicio accidental mientras los usuarios están conectados a tu aplicación puede causar graves problemas.

---

Puedes instalar nodemon a través de npm usando el comando `npm install -g nodemon`. Esto hará que nodemon esté disponible globalmente para su uso en tu sistema. En el caso más

simple, para ejecutar tu aplicación, debes reemplazar el comando `node` con `nodemon`. El listado 23 muestra la ejecución del ejemplo de servidor web a través de `nodemon`.

Listado 23: Ejecución de una aplicación con “`nodemon`”.

```
PS C:\Users\abrah\Documents\ProyectosNode> nodemon server.mjs
[nodemon] 3.1.4
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting `node server.mjs`
Server is listening to
http://localhost:8080
```

Si tienes problemas en la ejecución, puedes leer la siguiente página web para configurar correctamente:

[https://learn.microsoft.com/es-es/powershell/module/microsoft.powershell.core/about/about\\_execution\\_policies?view=powershell-7.4](https://learn.microsoft.com/es-es/powershell/module/microsoft.powershell.core/about/about_execution_policies?view=powershell-7.4)

Tan pronto como se inicia `nodemon` y modifica un archivo en tu proyecto, en este caso, el archivo `server.mjs`, el proceso se reiniciará automáticamente. Sin embargo, al usar `nodemon`, debes tener en cuenta que dicho reinicio hará que se pierda el estado actual, es decir, todas las asignaciones de variables.

Como se indica en la salida de `nodemon`, no necesariamente tienes que guardar un archivo para provocar un reinicio. También puedes ingresar “`rs`” en la consola donde se está ejecutando `nodemon`. Además de esta opción de control directo, también puedes manipular el comportamiento de `nodemon` a través de un archivo de configuración. En este contexto, se recomienda nombrar ese archivo `nodemon.json`. Este archivo se pasa con la opción `—config nodemon.json`. El listado 24 contiene un ejemplo de un archivo de configuración de este tipo.

Listado 24: Configuración de “`nodemon`”.

```
{
  "verbose": true,
  "ignore": ["*.spec.js"],
  "execMap": {
    "rb": "ruby"
  }
}
```

La propiedad `verbose` activa una salida de registro adicional en la línea de comandos. Con `ignore`, puedes especificar archivos cuya modificación no debe provocar un reinicio automático del proceso. Finalmente, `execMap` es una característica que te permite iniciar automáticamente no solo JavaScript, sino también cualquier otro script, como Ruby en este caso. La propiedad `execMap` contiene una asignación de la extensión del archivo al programa que se está ejecutando. Por ejemplo, si llamas a `nodemon index.rb` con esta configuración, el script se ejecutará en Ruby.

Como alternativa al archivo `nodemon.json`, también puedes almacenar la configuración de `nodemon` directamente en el archivo `package.json` de tu proyecto bajo la clave `nodemonConfig`.

Por último, pero no por ello menos importante, `nodemon` también admite la depuración de aplicaciones Node.js. Puedes utilizar las opciones `—inspect` o `—inspect-brk` para la depuración remota o `inspect` para la depuración interactiva en la línea de comandos.

## Resumen

En este documento, aprendiste a utilizar REPL para iniciar sesiones interactivas con Node.js para ejecutar experimentos ligeros. Sabes cómo iniciar aplicaciones Node.js normales en la línea de comandos pasando el nombre del archivo inicial. También se te presentó el módulo `http` en forma de un servidor web simple y lo utilizaste para desarrollar tu primera aplicación.

El depurador integrado de la plataforma Node.js te permite solucionar problemas dentro de una aplicación en ejecución. Puedes usarlo tanto de forma interactiva en la línea de comandos como de forma remota a través de Chrome DevTools o un entorno de desarrollo.

Con `nodemon`, finalmente conocerás una herramienta que reinicia automáticamente tu aplicación para permitir que los cambios en el código fuente surtan efecto.