

---

## Módulos de Node.js

---

El todo es más que la suma de sus partes.

—Aristóteles

La estructura de Node.js está inspirada en la filosofía Unix. En esencia, esta filosofía establece que el software debe constar de componentes que se centran en completar una sola tarea específica. Estos componentes trabajan juntos a través de interfaces para formar el sistema general.

Esta idea recorre toda la estructura de Node.js, desde las bibliotecas principales hasta su propia aplicación. Este documento está diseñado para ayudarte a comprender el sistema de módulos de Node.js y cómo puedes usarlo al crear tus aplicaciones.

### Estructura modular

La estructura de Node.js se puede comparar con la de una cebolla. La plataforma está formada por varias capas, cada una de las cuales cumple una función específica. La figura 1 ilustra esta estructura.

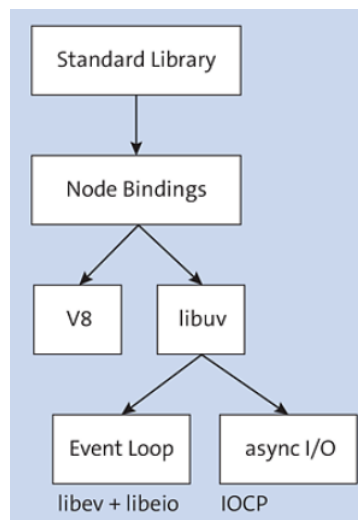


Figura 1: Estructura de Node.js.

El núcleo de Node.js es el motor V8 y algunas bibliotecas adicionales como libuv o openssl. Esta capa representa el entorno de ejecución y está escrita en C++. La mayoría de los desarrolladores que trabajan con Node.js no entran en contacto directo con estos componentes.

La capa que rodea el núcleo de la plataforma se denomina enlaces de nodos (node bindings) y representa la transición entre C++ y JavaScript. Esta parte de la plataforma permite que partes de Node.js se implementen en JavaScript y también se puedan usar directamente cuando implementas aplicaciones.

La capa final de la plataforma Node.js propiamente dicha -la biblioteca estándar- consta de los módulos centrales. Con esta interfaz, trabajas directamente en el desarrollo de tus aplicaciones. La tarea de los módulos centrales es darte acceso a las interfaces de tu sistema, para que puedas acceder al sistema de archivos o a la red, por ejemplo.

Además de la modularización, otro objetivo de los desarrolladores de Node.js es mantener la plataforma relativamente pequeña, o Node.js es solo una herramienta que proporciona la funcionalidad básica para tu aplicación. Tú tienes que encargarte de todo lo demás tú mismo, lo que significa, por ejemplo, que tendrías que implementar el acceso a una base de datos o al protocolo WebSocket tú mismo.

Pero esto contradice la filosofía de Node.js, que prevé muchos componentes pequeños y reutilizables. Por esta razón, existen administradores de paquetes para Node.js, especialmente Node Package Manager (npm), que está muy vinculado a la plataforma.

Los administradores de paquetes o los paquetes externos forman otra capa en la arquitectura de Node.js y te proporcionan casi todos los módulos imaginables que necesitas al implementar una aplicación. En el sentido más amplio de la palabra, un módulo es una biblioteca en Node.js que se puede cargar a través del sistema de módulos y luego usar en una aplicación. Un paquete consta de uno o más módulos que se han combinado en un solo paquete. Este paquete luego se puede instalar a través de un administrador de paquetes.

La gama de paquetes disponibles se extiende desde bibliotecas auxiliares como lodash y controladores de bases de datos hasta frameworks de aplicaciones completos o sistemas de gestión de contenido (CMS, content management systems). Y si no encuentras el paquete que necesitas, siempre puedes escribirlo y publicarlo tú mismo.

Finalmente, la última capa es tu propia aplicación. Aquí también debes adoptar un enfoque modular. Esto significa que tu aplicación consta de muchas partes pequeñas e independientes que interactúan entre sí a través de interfaces definidas.

La figura 2 contiene un ejemplo de la estructuración de una aplicación. Debes prestar especial atención a la denominación significativa de directorios y archivos, lo que hace que sea mucho más fácil encontrar el código fuente.

---

## Estructura de la aplicación

La modularización es la clave para una aplicación ampliable y mantenible. Sin embargo, esto no significa que cada aplicación Node.js tenga que ser un conjunto de microservicios. En cambio, debes elegir una forma arquitectónica adecuada para cada problema. Un monolito con una estructura modular también puede ser una estrategia de solución válida para un problema claramente definido.

Al modularizar, debes asegurarte de que cada módulo esté en su propio archivo y contenga solo una cantidad manejable de funciones o exactamente un objeto o clase concretos. También debes asegurarte de que el contenido de un módulo sea siempre muy similar temáticamente y resuelva solo una tarea muy específica a la vez. Asigna a las estructuras de tus módulos nombres significativos que sean consistentes en toda la aplicación. Luego, puedes organizar los archivos individuales en una estructura de directorio, donde un directorio debe contener nuevamente módulos del mismo tema.

Para mantener una aplicación organizada, no debes colocar demasiados archivos en un solo directorio. Como pauta segura, puedes usar entre 7 y 10 archivos por directorio. Si hay más, debes crear subdirectorios.

---

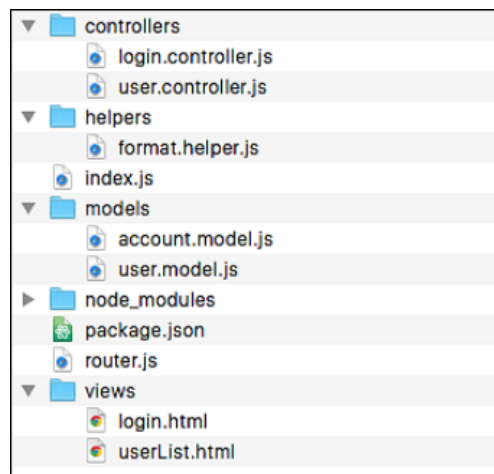


Figura 4.2 Estructura de una aplicación.

La modularización en el desarrollo de software tiene algunas ventajas decisivas:

- Reutilización de subcomponentes
- Mejor capacidad de prueba gracias a componentes pequeños y autónomos
- Paralelización durante el desarrollo gracias a la independencia de los componentes
- Intercambiabilidad de componentes individuales a través de interfaces definidas

En el núcleo de Node.js, el diseño modular permite a los desarrolladores modernizar continuamente la plataforma marcando funciones o módulos individuales como obsoletos o

añadiendo nuevos módulos. Las siguientes secciones describen qué módulos componen este núcleo.

## Módulos centrales

Los bloques básicos de construcción de una aplicación son los módulos centrales, como el módulo HTTP o el módulo del sistema de archivos. Puedes encontrar documentación sobre estos módulos en <https://nodejs.org/api/>. Este enlace hace referencia a la documentación de la versión actual de Node.js y cubre todas las interfaces, describe cómo usarlas y te ofrece numerosos ejemplos de código concretos. La documentación se administra utilizando el código fuente de la plataforma. También puedes descargarla directamente y obtener acceso a la descripción de versiones anteriores. Alternativamente, puedes ver estas versiones anteriores utilizando el enlace Ver otra versión en el encabezado de la página de documentación.

Si esta información no es suficiente para ti y la búsqueda en Internet no proporciona el resultado esperado, aún puedes acceder directamente al código fuente del módulo correspondiente. Puedes encontrarlo en el repositorio de GitHub del proyecto Node.js en <https://github.com/nodejs/node> en el directorio lib. Además del código fuente en sí, también puedes acceder a una gran cantidad de pruebas unitarias que describen con más detalle el uso de las respectivas interfaces mediante casos de prueba concretos.

En los primeros días de Node.js, las interfaces cambiaban con mucha frecuencia y también había cambios disruptivos frecuentes, es decir, cambios en las interfaces que causaban problemas para las aplicaciones existentes. Mientras tanto, los desarrolladores intentan evitar este fenómeno tanto como sea posible. Para obtener una lista de los cambios en una versión, debes consultar el registro de cambios del proyecto en <https://github.com/nodejs/node/blob/master/CHANGELOG.md>.

## Estabilidad

El enfoque modular en la estructura de Node.js también respalda la estabilidad del sistema en general y ayuda a evitar errores. Si uno de los módulos contiene un mal funcionamiento, todos los demás módulos pueden seguir funcionando.

Por supuesto, esto no se aplica a los componentes centrales como el motor V8. Pero, en general, la compatibilidad con versiones anteriores y el soporte de la plataforma son muy importantes.

Debido al continuo desarrollo de Node.js, surge la pregunta de qué módulos se pueden usar sin dudarlo, cuáles ya no se deben usar y cuáles es mejor dejar esperar un tiempo antes de entrar en funcionamiento.

Para obtener una mejor visión general de los módulos y su grado de estabilidad, se creó el índice de estabilidad. Puedes encontrar la estabilidad de un módulo en la documentación de la interfaz de programación de aplicaciones (API). Para cada módulo, la tabla de contenidos y el encabezado de una página de documentación están seguidos de una barra de color que indica el valor del índice. Originalmente, este valor constaba de seis niveles, luego se redujo a cuatro niveles, luego a tres y, actualmente, nuevamente a cuatro niveles, que encontrará brevemente descritos en la tabla 1.

Tabla 1: Índice de estabilidad.

Índice	Nombre	Código de color	Descripción
0	Obsoleto	Rojo	Estos módulos ya no deberían utilizarse. Se eliminarán de las versiones futuras. Al utilizar estas funciones, pueden aparecer advertencias. No se garantiza la compatibilidad con versiones anteriores de estos módulos.
1	Experimental	Naranja	Los módulos con este valor de índice se desarrollan activamente, pero no se recomienda su uso en producción. Se esperan cambios. No están sujetos a versiones semánticas; también pueden ocurrir cambios importantes en actualizaciones menores.
2	Estable	Verde	Casi no habrá cambios en estos módulos. La compatibilidad se interrumpe solo en casos excepcionales. Los desarrolladores dan gran importancia a la compatibilidad con el ecosistema npm.
3	Legado	Azul	No se recomienda el uso de estas funciones, ya que están sujetas a control de versiones semántico y seguirán formando parte de la plataforma en el futuro cercano.

Un ejemplo de un módulo obsoleto es el módulo domain. Fue un intento de crear un dominio de error para manejar errores en un área de una aplicación. Este módulo ha demostrado ser inútil durante varios años y ya no será compatible en el futuro. El estado obsoleto también se aplica a numerosos métodos de módulos existentes. Por ejemplo, no deberías utilizar el método exist del módulo fs en el futuro. En la mayoría de los casos, la documentación te dará sugerencias sobre la funcionalidad que puedes utilizar en su lugar.

De las funcionalidades existentes, solo la salida de notación de objetos JavaScript (JSON) de la documentación tiene el estado experimental durante un período de tiempo más largo. Las nuevas características, como el módulo `async_hooks` o el módulo `diagnostics_channel`, se marcan como experimentales hasta que se estabilicen.

La mayoría de los módulos en la plataforma Node.js tienen el estado de índice estable. Esto significa que, si bien la API puede cambiar, los desarrolladores prestan especial atención a la compatibilidad con versiones anteriores. Ejemplos típicos de módulos con estado estable son los módulos `stream`, `http` y `fs`.

Un objetivo importante en el desarrollo de Node.js es mantener la plataforma organizada y manejable. Por esta razón, solo hay una cantidad comparativamente pequeña de módulos básicos.

### Lista de módulos básicos

La tabla 2 proporciona una descripción general de los módulos básicos de la plataforma Node.js.

Tabla 2: Descripción general de los módulos de Node.js.

Módulo	Descripción
<code>assert</code>	Este es el módulo de pruebas de Node.js que se utiliza para cubrir la plataforma con pruebas unitarias.
<code>async_hooks</code>	Este módulo permite y simplifica el seguimiento de recursos asíncronos durante su ciclo de vida.
<code>buffer</code>	Este módulo ayuda a manejar datos binarios para el sistema de archivos o las operaciones de red.
<code>child_process</code>	Este módulo te permite crear y controlar procesos secundarios y comunicarse entre procesos.
<code>cluster</code>	Este módulo es una extensión del módulo <code>child_process</code> para el equilibrio de carga dinámico entre procesos.
<code>console</code>	Este módulo representa un contenedor (wrapper) para la salida estándar.
<code>crypto</code>	Este módulo proporciona cifrado en Node.js.
<code>dgram</code>	Este módulo te permite establecer conexiones de Protocolo de datagramas de usuario (UDP) y comunicarse a través de ellas.
<code>dns</code>	Este módulo resuelve nombres a través del Sistema de nombres de dominio (DNS).
<code>domain</code>	Este módulo se ocupa de las operaciones de agrupamiento y del manejo de los errores que se producen.
<code>events</code>	El módulo de eventos proporciona la infraestructura para registrar y activar eventos.

fs	El módulo fs se utiliza para obtener acceso de lectura y escritura al sistema de archivos del sistema.
http	Este módulo te ayuda a crear clientes y servidores HTTP.
http2	Con este módulo, el protocolo HTTP/2 ahora es compatible de forma nativa con Node.js. HTTP/2 representa un desarrollo posterior de HTTP.
https	Este módulo permite la comunicación cifrada tanto como cliente como servidor HTTP.
inspector	Este módulo representa la interfaz con el inspector V8.
modules	Con este módulo, el sistema de módulos Node.js se implementa como un módulo.
net	Este módulo es la base para los servidores y clientes de red.
os	Este módulo te permite leer información sobre el sistema operativo.
path	Este módulo contiene todas las operaciones importantes para manejar rutas de directorio.
perf_hooks	Este módulo te permite realizar mediciones de tiempo de alta precisión en su aplicación.
process	El módulo de proceso es lo mismo que el módulo os para el sistema operativo para el proceso actual de Node.js.
punycode	Este módulo se ocupa de la conversión conforme a los estándares de caracteres Unicode a caracteres ASCII.
querystring	Este módulo simplifica enormemente el manejo de cadenas de consulta en URL.
readline	Gracias a este módulo, se puede consumir una secuencia línea por línea.
repl	Este módulo representa el ciclo de lectura, evaluación e impresión (REPL), que es la línea de comandos de Node.js.
stream	La API de este módulo proporciona distintos tipos de secuencias de datos.
string_decoder	Este módulo se puede utilizar para convertir objetos de búfer en cadenas.
timers	Este módulo contiene todas las funciones basadas en tiempo.
tls	Este módulo es la interfaz a la infraestructura de llave pública/privada para la comunicación segura.
trace_events	Con este módulo se puede procesar de forma centralizada la información de seguimiento del motor V8, la plataforma Node, así como la información de seguimiento personalizada del usuario.
tty	Este módulo representa la conexión a la terminal, por ejemplo, la entrada y salida estándar.
url	Este módulo se utiliza para generar o analizar URL.
util	Este módulo recopila funciones útiles como consultas de tipo, formato y similares.
v8	Este módulo te permite trabajar directamente con el motor V8.
vm	Este módulo proporciona un entorno para ejecutar código JavaScript.
wasi	Este módulo se puede utilizar para integrar módulos WebAssembly o aplicaciones completas en Node.js. El módulo representa una implementación de la especificación de interfaz del sistema WebAssembly.

worker_threads	Además del módulo child_process, este módulo ayuda a descargar rutinas del proceso principal. El módulo worker_threads se centra en los cálculos que hacen un uso intensivo de la CPU.
zlib	Este módulo se ocupa de la compresión y descompresión de la información.

Algunos de los módulos presentados aquí sirven como una interfaz para el sistema operativo, como los módulos net o fs. Otros implementan protocolos o definen estructuras de uso común, como los módulos http o el módulo events. Estos últimos son utilizados a menudo por Node.js para construir otros módulos básicos. Partes del módulo fs, por ejemplo, utilizan el módulo stream para mapear flujos de datos; este módulo a su vez se basa en el módulo events.

### Carga de módulos básicos

Los módulos básicos de Node.js se cargan a través del sistema de módulos de la plataforma, con algunas excepciones.

Actualmente, la plataforma se encuentra en un estado de transición, ya que el sistema de módulos CommonJS utilizado anteriormente se está cambiando a los módulos ECMAScript.

Por supuesto, un cambio tan drástico lleva tiempo, por lo que la transición al nuevo sistema modular ya ha llevado años. En el transcurso de este curso, utilizaremos principalmente el sistema de módulos ECMAScript.

Sin embargo, en esta sección, conocerás ambos sistemas de módulos y sus respectivas características.

### Carga de módulos principales a través del sistema de módulos CommonJS

El sistema de módulos original de Node.js utiliza la función require para cargar estructuras. El listado 1 muestra un ejemplo de carga del módulo os para mostrar el tiempo de actividad del sistema.

Listado 1: Carga de módulos principales.

```
// Loading the entire module
const os = require('os');
console.log(os.uptime());

// Loading the module and extracting certain functions by means of
// destructuring
const { uptime } = require('os');
console.log(uptime());
```



Cada uno de los módulos principales exporta un objeto a través del cual puedes utilizar la funcionalidad del módulo respectivo. Normalmente, asignas el valor de retorno de la función require a una variable que tiene el mismo nombre que el módulo y luego llamas a las respectivas funciones requeridas.

En el ejemplo de código, puedes ver tanto esta variante como cómo puedes extraer directamente las interfaces requeridas con una declaración de desestructuración.

---

## Desestructuración

Si deseas almacenar propiedades individuales de objetos o elementos de arreglos en varias variables, son necesarias varias operaciones, como puedes ver en el listado 2.

Listado 2: Asignación de propiedades de objetos a variables.

```
const person = {  
  name: 'Lisa',  
  age: 32  
};  
const name = person.name;  
const age = person.age;
```

Tanto para objetos como para arreglos, puedes combinar dichas operaciones mediante la desestructuración. Esta función forma parte de la versión estándar desde ECMAScript 2015 y ahora está disponible en todos los navegadores modernos. Puedes escribir el código del listado 2 de una manera más compacta mediante la desestructuración, como se muestra en el listado 3.

Listado 3: Declaración de desestructuración.

```
const person = {  
  name: 'Lisa',  
  age: 32  
};  
  
const { name, age } = person;  
  
console.log('Name: ', name); // Output: Name: Lisa  
console.log('Age: ', age); // Output: Age: 32  
  
const person2 = ['John', 17];  
const [name2, age2] = person2;  
  
console.log('Name: ', name2); // Output: Name: John  
console.log('Age: ', age2); // Output: Age: 17
```

El ejemplo de código también muestra que puedes utilizar corchetes en lugar de llaves para arreglos y, de este modo, asignar los elementos individuales a variables.

---

---

También puedes aplicar la desestructuración a niveles más profundos y combinar la desestructuración para objetos y arreglos como desees. Sin embargo, debes tener cuidado de no exagerar, ya que puede volverse muy confuso fácilmente.

---

## **Carga de módulos principales mediante el sistema de módulos ECMAScript**

El sistema de módulos oficial de JavaScript adopta un enfoque ligeramente diferente al sistema de módulos ECMAScript utilizado anteriormente. Con respecto a la carga de estructuras, el cambio más importante es que la palabra clave `import` reemplaza la función `require`.

Los módulos principales de Node.js tienen una exportación predeterminada que es muy similar al comportamiento del antiguo sistema de módulos. Además, cada módulo exporta su interfaz pública como exportaciones con nombre, que puedes comparar con la declaración de desestructuración que utilizamos anteriormente. El listado 4 muestra cómo puedes utilizar la exportación predeterminada de los módulos principales.

Listado 4: Inclusión de módulos principales con una importación predeterminada.

```
import os from 'os';  
console.log(os.uptime());
```

El listado 4.5 muestra la segunda variante; es decir, el uso de exportaciones con nombre.

Listado 5: Inclusión de módulos principales con una importación con nombre.

```
import { uptime } from 'os';  
console.log(uptime());
```

Depende de ti cuál de las dos variantes utilizas en tu aplicación, pero asegúrate de implementar tu código fuente de la manera más coherente posible y utilizar las mismas variantes en todo momento, ya que esto aumenta significativamente la legibilidad del código fuente.

---

## **Sistema de módulos ECMAScript**

Actualmente, el sistema de módulos CommonJS sigue siendo el estándar para Node.js. Aunque esto cambiará en el futuro, tenlo en cuenta si quieres trabajar con el sistema de módulos ECMAScript. Hay tres formas de habilitar el sistema de módulos ECMAScript:

### **Extensión de archivo .mjs**

Ya has aprendido sobre esta variante. Si un nombre de archivo termina con `.mjs`, Node.js se asegura automáticamente de que el sistema de módulos ECMAScript esté activado.

---

---

### **Campo de tipo (type) en el archivo package.json**

Alternativamente, puedes configurar el campo de tipo en el archivo package.json con el valor del módulo (module). Si lo haces, tus archivos pueden mantener la extensión .js habitual.

### **Bandera —input-type**

La tercera variante es iniciar tu proceso Node.js con la opción de línea de comandos —input-type=module.

Nuevamente, si decides hacer eso, puedes seguir usando la extensión de archivo .js.

---

### **Archivo package.json**

Puedes utilizar el campo type en el archivo package.json para habilitar el sistema de módulos ECMAScript. Pero este archivo puede hacer mucho más, como aprenderás a lo largo de este curso. En este punto, lo único importante que debes saber es que este es el archivo de descripción central de tu aplicación. En él, almacenas información, como el nombre o el número de versión de tu aplicación. Además, el archivo contiene una lista de todas las dependencias que requiere tu aplicación.

Puedes hacer que npm cree un archivo package.json en la línea de comandos ejecutando el comando npm init.

Luego se te harán una serie de preguntas sobre tu proyecto. Además, puedes utilizar la opción -y para que se respondan estas preguntas con las respuestas predeterminadas.

---

Además de los módulos principales que debes cargar explícitamente, existen algunas estructuras disponibles globalmente que puedes usar en tu aplicación. También existen algunas diferencias entre el sistema de módulos CommonJS y el sistema de módulos ECMAScript, que describiremos con más detalle en la siguiente sección.

### **Objetos globales**

Los desarrolladores de Node.js han optado por hacer que los objetos y funciones individuales estén disponibles globalmente y dentro del alcance de un módulo, respectivamente, por varias razones. Para algunos, es simplemente necesario que estén allí porque sin ellos, no podrías desarrollar tu aplicación. Esto incluye, por ejemplo, el sistema modular.

Otros objetos y funciones también están disponibles globalmente en el navegador y se han adoptado en Node.js, como el objeto de consola y las funciones de temporización. La tercera categoría proporciona funcionalidades de uso común, como la constante \_\_filename.

La tabla 3 proporciona una descripción general de los objetos globales en Node.js.

Tabla 3: Objetos globales en Node.js.

Nombre de la variable	Nombre de la variable	Nombre de la variable	Nombre de la variable
__dirname	__filename	module	performance
Buffer	clearImmediate()	process	queueMicrotask()
clearInterval()	clearTimeout()	require()	setImmediate()
console	Event	setInterval()	setTimeout()
EventTarget	exports	TextDecoder	TextEncoder
global	MessageChannel	URL	URLSearchParams
MessageEvent	MessagePort	WebAssembly	

Además de estos objetos específicos de Node.js, se incluyen muchas otras clases y objetos en el estándar de JavaScript, por ejemplo, las clases de cadena y arreglos.

En las siguientes secciones, presentaremos brevemente los objetos globales individuales y sus posibles usos.

### Nombre de archivo y directorio

Dos variables que son muy útiles en el desarrollo de aplicaciones son `__filename` y `__dirname`. Como ya sugieren los nombres de estas variables, ambas contienen información sobre la ubicación del archivo, que almacena el código fuente del script que se está ejecutando actualmente. La cadena en `__filename` es la ruta absoluta y el nombre de archivo del script. `__dirname` es solo la ruta.

Obtienes el nombre del archivo o directorio donde se usa la variable, incluso si está incluida en otro lugar a través del sistema de módulos.

Cuando se usa en el REPL, ten en cuenta que ni `__filename` ni `__dirname` están definidos, ya que no hay ningún script en este caso. Acceder a estas variables dará como resultado un `ReferenceError`.

La misma restricción se aplica al sistema de módulos ECMAScript. Si intentas acceder a estas dos variables, el sistema devolverá un `ReferenceError`. Sin embargo, existe una manera de obtener la información del nombre del archivo y directorio actual, como se muestra en el listado 6.

Listado 6: Simulación de `__filename` y `__dirname` para el sistema de módulos ECMAScript.

```
import { dirname } from 'path';
import { fileURLToPath } from 'url';

const __filename = fileURLToPath(import.meta.url);
const __dirname = dirname(__filename);
```

```
console.log(__filename);  
console.log(__dirname);
```

Puedes utilizar la propiedad `import.meta.url` para mostrar el nombre de archivo absoluto del script actual. Sin embargo, aquí Node.js agrega el protocolo al principio, en este caso, la cadena `file://`. Para obtener la ruta absoluta pura, debes utilizar la función `fileURLToPath` del módulo `url`. Puedes obtener el nombre del directorio al pasar el nombre de archivo absoluto a la función `dirname` del módulo `path`.

## Buffer

La clase `buffer` también está disponible globalmente porque se trabaja con este tipo de objeto muy a menudo en Node.js. Los objetos `buffer` se utilizan siempre que se trata de manejo de datos binarios. El listado 7 contiene un ejemplo que muestra cómo puedes entrar en contacto con objetos `buffer`, a veces de manera involuntaria.

Listado 7: Objetos `buffer` al leer archivos.

```
import { readFile } from 'fs';  
  
readFile('input.txt', (err, data) => {  
  console.log(data); // Output: <Buffer 48 61 6c 6c 6f 20 57 65 6c 74>  
  console.log(data.toString()); // Output: Hello world  
});
```

En este ejemplo, se lee el contenido del archivo `input.txt` y se escribe en la salida estándar. Sin embargo, como no se ha especificado ninguna codificación de caracteres aquí, Node.js vuelve al manejo predeterminado, lo que significa que considera los datos como un búfer. Los métodos más importantes de los objetos de búfer son el método `toString` para convertir a una cadena y el método `write` para escribir información en el búfer. Por lo tanto, si deseas enviar el contenido del objeto de búfer a la consola, puedes utilizar el método `data.toString`, como se muestra en el ejemplo.

## Funciones de temporización (Timing)

Las funciones de temporización `setTimeout`, `setInterval` y `setImmediate` no forman parte del estándar del lenguaje ECMAScript. En el navegador, forman parte del objeto de ventana. Debido a que estas funciones son esenciales para implementar funciones basadas en el tiempo, se han trasladado a Node.js y están disponibles globalmente. `setTimeout` te permite ejecutar una función de devolución de llamada después de que haya transcurrido la cantidad especificada de milisegundos. La función `setInterval` ejecuta la función de devolución de llamada especificada periódicamente a intervalos, y `setImmediate` ejecuta la función de

devolución de llamada al final del ciclo del loop de eventos actual. El valor de retorno de estas tres funciones se puede utilizar junto con la función `clear` respectiva, por ejemplo, `clearTimeout`, para abortar la operación correspondiente.

En el listado 8, primero crea una variable de contador y luego inicia un intervalo que genera una salida cada segundo.

Sin ninguna otra condición de terminación, el script continuaría ejecutándose hasta que finalmente lo termine manualmente en la consola. Para evitar este efecto, debes definir una condición de terminación que garantice que el procesamiento finalice después de tres ejecuciones.

Listado 8: Uso de la función “`setInterval`”.

```
let counter = 1;
const interval = setInterval(() => {
  console.log(`${counter} iteration`);
  if (counter++ > 2) {
    clearInterval(interval);
  }
}, 1000);
```

## consola

Otra funcionalidad que ya conoces del JavaScript del lado del cliente (el objeto `console`) también está disponible como un objeto global y se utiliza principalmente para la salida en la línea de comandos durante el desarrollo. Además del conocido método `console.log`, existen muchos otros métodos útiles en el objeto `console`. La tabla 4 proporciona una lista de los métodos más importantes.

Tabla 4: Métodos más importantes del objeto “`console`”.

Método	Descripción
<code>debug</code> , <code>info</code> , <code>warn</code> , <code>error</code>	Estos métodos representan diferentes prioridades de registro.
<code>group</code> , <code>groupEnd</code>	Estos métodos se utilizan para agrupar visualmente las salidas de la consola.
<code>time</code> , <code>timeEnd</code>	Estos métodos permiten una medición de tiempo sencilla. Con <code>time</code> , puedes establecer un marcador, mientras que <code>timeEnd</code> finaliza la medición de tiempo y muestra el resultado.
<code>trace</code>	Este método muestra el seguimiento de la pila actual.

## API de eventos globales

Las dos clases, `EventTarget` y `Event`, son implementaciones de la API de eventos del lado del cliente. En el sentido más amplio, puedes comparar `EventTarget` con, por ejemplo, un elemento de botón en el navegador en el que un usuario ha hecho clic. La mayor diferencia es que la estructura de elementos en el navegador es jerárquica, lo que no es el caso en Node.js. Esto significa que los eventos no pasan a través de un árbol de elementos, sino que solo se activan en `EventTarget`. El listado 9 muestra un ejemplo de cómo se pueden utilizar las dos clases de eventos.

Listado 9: Uso de la API de eventos.

```
const target = new EventTarget();

target.addEventListener('customEvent', (event) => {
  console.log(`${event.type} was triggered`);
  // Output: customEvent was triggered
});

const event = new Event('customEvent');
target.dispatchEvent(event);
```

En el ejemplo, primero se crea una instancia de `EventTarget` en la que se registra un controlador de eventos para el tipo `customEvent` a través del método `addEventListener`. Puedes elegir estos tipos de eventos tú mismo. Luego, creas un objeto de evento del tipo `customEvent` y lo activas a través del método `dispatchEvent` de `EventTarget`.

Más adelante, aprenderás sobre otra forma de controlar eventos en Node.js a través de la clase `EventEmitter` del módulo de eventos. Esta clase constituye la base de numerosos módulos internos y externos en Node.js.

## Sistema de módulos CommonJS

También puedes acceder a las funciones y objetos específicos de Node.js del sistema de módulos desde cualquier lugar dentro de tu módulo. La función `require` te permite cargar dependencias externas, mientras que las exportaciones o el objeto de módulo se utilizan para hacer que la información esté disponible fuera de tu módulo.

También puedes utilizar este objeto para controlar otros aspectos del sistema de módulos. Presentaremos el sistema de módulos de Node.js con más detalle más adelante. Las versiones más recientes de Node.js también proporcionan al sistema de módulos ECMAScript las palabras clave `import` y `export`.

## **global: Ámbito global**

En JavaScript del lado del cliente, tienes el ámbito global para las variables además de los ámbitos de bloque, función y cierre. Node.js no contiene nada parecido. Si defines una variable en un archivo, solo puedes acceder a ella dentro de este archivo, por lo que la variable solo es válida dentro del módulo. Si agregas una nueva propiedad al objeto global, estará disponible como una propiedad global en toda tu aplicación, es decir, incluso más allá de los límites de archivo. El listado 10 ilustra cómo se puede utilizar el objeto global.

Listado 10: Uso del objeto “global”.

```
function createGlobal() {  
    global.myName = 'Peter';  
}  
createGlobal();  
console.log(myName);
```

En el listado 10, primero se crea una función en la que se establece la variable global. Luego se llama a la función y se genera el contenido de la variable global. A mayor escala, la configuración y el acceso a variables globales también funcionan a través de los límites de los archivos.

---

### **Evita usar variables globales!**

En este punto, recomendamos enfáticamente no usar variables globales, ya que pueden generar conflictos de nombres con variables locales y, en general, hacen que tu aplicación sea más difícil de leer y más propensa a errores.

---

## **Sistema de mensajes para comunicarse con procesos de trabajo**

Puedes utilizar tres clases, `MessageChannel`, `MessageEvent` y `MessagePort`, para la comunicación entre procesos de trabajo. La clase `MessagePort` representa el final de un canal de comunicación bidireccional, que a su vez está representado por la clase `MessageChannel`. La implementación se basa en la implementación `MessagePort` del navegador.

Como puedes ver en el listado 11, la clase `MessageChannel` crea un nuevo objeto con las dos propiedades `port1` y `port2`, cada una de las cuales es una instancia de la clase `MessagePort` y representa los dos puntos finales del canal de comunicación. Ahora puedes utilizar estos objetos para enviar mensajes entre ambos puertos.

Listado 11: Uso de la clase `MessageChannel`.

```
const { port1, port2 } = new MessageChannel();
```



```
port1.on('message', (message) => {  
  console.log(message);  
});  
  
port2.postMessage({ data: 'Hello world' });
```

## performance

El objeto `performance` es una referencia al módulo `perf_hooks` que puedes utilizar para medir el rendimiento de tu aplicación. Para obtener más información sobre esta interfaz y otros aspectos de rendimiento, consulte el manual de referencia de Node.js.

## proceso

El objeto de proceso (*process*) global no es un módulo Node.js completo; su código fuente no se encuentra en el directorio `lib` de la plataforma Node.js. El objeto de proceso es tu interfaz con el proceso Node.js actual que ejecuta tu aplicación. Por ejemplo, puedes acceder a la salida estándar, la entrada estándar y la salida de error estándar a través de `process.stdout`, `process.stdin` y `process.stderr`.

También puedes utilizar el objeto de proceso para finalizar tu aplicación de manera controlada. Por ejemplo, con una llamada al método de salida, al que opcionalmente le pasas un código de estado, puedes finalizar el proceso actual, como se muestra en el listado 12.

Listado 12: Valor de retorno de aplicaciones Node.js.

```
$ node  
> process.exit(42);  
$ echo $?  
42
```

Con `echo $?`, puedes consultar el valor de retorno de la última aplicación en el shell de Unix. Un valor distinto de 0 significa que se produjo un error durante la ejecución. Dependiendo de cómo elijas este valor, puedes usarlo para codificar más información.

El método `abort` es un poco más drástico que `process.exit`. Aquí, el apagado ya no se produce de manera controlada. Además de estos dos métodos, también puedes utilizar el método `kill`, que acepta un ID de proceso como primer argumento y una señal que se enviará como segundo argumento opcional. Normalmente, se trata de `SIGTERM`, que lleva a la terminación del proceso.

El listado 13 muestra cómo puedes utilizar el método `kill`.

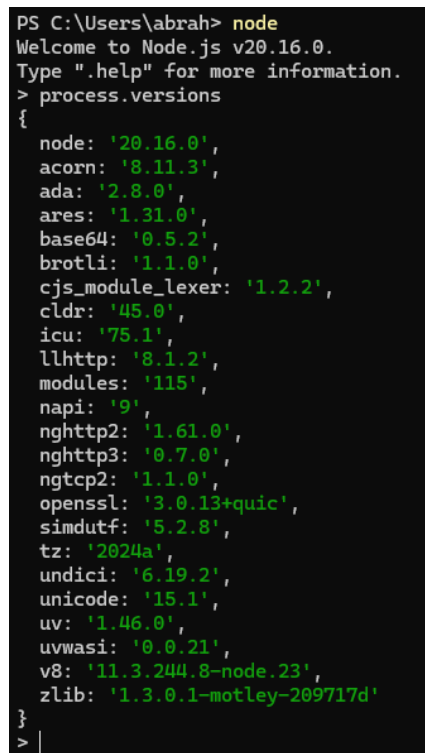
## Listado 13: Finalización de procesos con “kill”.

```
$ node
> process.kill(process.pid, 'SIGKILL');
Killed: 9
$ echo $?
137
```

Puedes utilizar la propiedad `argv` del objeto `process` para acceder a la llamada de línea de comandos de tu aplicación. Suponiendo que llamas a tu aplicación a través de la línea de comandos `node server.js --env=debug`, `argv` contiene un arreglo de tres elementos. El primer elemento es el comando `node` en sí, el segundo es la ruta absoluta y el nombre del archivo JavaScript, y el tercero es la opción `--env=debug` especificada.

La propiedad `env` proporciona las variables de entorno del proceso actual de Node.js. Parte de la información es, por ejemplo, el nombre del usuario, la variable `PATH` y el directorio de inicio del usuario.

Puedes utilizar el objeto de proceso para leer las versiones de la plataforma Node.js. `process.version` devuelve la versión de Node.js en sí. `process.versions` también contiene la información de la versión de bibliotecas adicionales, como `libuv`, `zlib`, `openssl` o el motor V8. En la figura 3, puedes ver la salida de estas versiones.



```
PS C:\Users\abrah> node
Welcome to Node.js v20.16.0.
Type ".help" for more information.
> process.versions
{
  node: '20.16.0',
  acorn: '8.11.3',
  ada: '2.8.0',
  ares: '1.31.0',
  base64: '0.5.2',
  brotli: '1.1.0',
  cjs_module_lexer: '1.2.2',
  cldr: '45.0',
  icu: '75.1',
  llhttp: '8.1.2',
  modules: '115',
  napi: '9',
  nghttp2: '1.61.0',
  nghttp3: '0.7.0',
  ngtcp2: '1.1.0',
  openssl: '3.0.13+quic',
  simdutf: '5.2.8',
  tz: '2024a',
  undici: '6.19.2',
  unicode: '15.1',
  uv: '1.46.0',
  uvwasi: '0.0.21',
  v8: '11.3.244.8-node.23',
  zlib: '1.3.0.1-motley-209717d'
}
```

Figura 4.3 Salida de las versiones de Node.js.

Con el objeto del proceso, también puedes manipular activamente el proceso actual. Por ejemplo, puedes cambiar el usuario bajo el cual se ejecuta el proceso actual a través de `process.setuid`. Sin embargo, para hacer eso, necesitas el permiso para cambiar el usuario; de lo contrario, recibirás un mensaje de error. Puedes obtener el ID de usuario configurado actualmente utilizando el método `process.getuid`. De manera similar al ID de usuario, también puedes utilizar los dos métodos, `process.setgid` y `process.getgid`, con el ID de grupo.

El objeto exportado a través del módulo de proceso también representa un `EventEmitter`. Esto significa que, en determinadas situaciones, el módulo activa eventos a los que puedes suscribirte. La tabla 5 contiene una descripción general de los eventos con una breve descripción de cada uno.

Tabla 5: Eventos del módulo “process”.

Evento	Descripción
<code>beforeExit</code>	Este evento se activa cuando el ciclo de eventos ya no tiene más trabajo, lo que provoca que la aplicación finalice.
<code>Disconnect</code>	Este evento se activa cuando se interrumpe el canal de comunicación entre un proceso padre y su proceso hijo.
<code>Exit</code>	Este evento se activa cuando el proceso está a punto de finalizar.
<code>Message</code>	Puedes crear procesos hijos a través del módulo <code>child_process</code> . Puedes comunicarte entre los procesos individuales a través de mensajes. Tan pronto como un proceso recibe un mensaje, se activa el evento de mensaje
<code>multipleResolves</code>	Este evento se activa cuando una promesa se ha resuelto o rechazado más de una vez, o cuando se ha resuelto nuevamente después de un rechazo o se ha rechazado nuevamente después de una resolución. La activación de este evento es un indicador de posibles fuentes de error en la aplicación.
<code>rejectionHandled</code>	Este evento se activa si se rechaza una promesa y una función de control la intercepta.
<code>uncaughtException</code>	Si se lanza una excepción en tu aplicación, generalmente hace que el proceso finalice. Al controlar este evento, puedes manipular el comportamiento predeterminado.
<code>unhandledRejection</code>	Este evento actúa de manera similar al evento <code>rejectHandled</code> , excepto que se activa para promesas que no tienen controladores de rechazo.
<code>Warning</code>	Este evento se activa cuando se lanza una advertencia en tu aplicación.
<code>Worker</code>	Este evento se activa después de que se haya creado un proceso de trabajo. El argumento que obtiene es una referencia al proceso de trabajo creado en la función de control de eventos.

Cuando se trabaja con asincronicidad en el contexto de promesas, uno de estos eventos es particularmente útil. El evento `unhandledRejection` se activa cuando no se maneja un error de promesa en tu aplicación. El listado 14 contiene un ejemplo de código para esto.

## Listado 14: Rechazo de promesa no manejado.

```

process.on('unhandledRejection', (error) => {
  console.error('unhandledRejection'); // Output: unhandledRejection
  console.error(error); // Output: Whoops, an Error occurred
});

function withPromise() {
  return Promise.reject('Whoops, an Error occurred');
}

withPromise().then(() => {
  console.log('Promise resolved');
});

```

**queueMicrotask**

La asincronicidad es omnipresente en JavaScript y, por lo tanto, también en Node.js. En este punto, queremos darte una perspectiva sobre el tema y el contexto de la secuencia de ejecución. Existen diferentes tipos de asincronicidad en Node.js. Estos se refieren principalmente al orden de ejecución. Uno de los componentes principales de Node.js es el ciclo de eventos, que es responsable de registrar y ejecutar controladores de eventos.

Las ejecuciones individuales del ciclo de eventos se dividen en diferentes fases, como se muestra en el listado 15.

## Listado 15: Operaciones asíncronas en Node.js.

```

setTimeout(() => {
  console.log('setTimeout');
}, 0);

Promise.resolve().then(() => {
  console.log('Promise');
});

queueMicrotask(() => {
  console.log('queueMicrotask');
});

process.nextTick(() => {
  console.log('nextTick');
});

// Output:
// nextTick
// Promise
// queueMicrotask
// setTimeout

```

En el listado 15, puedes ver que la salida no corresponde al orden de registro de las respectivas funciones de devolución de llamada. El método `process.nextTick` es la unidad más pequeña que puedes utilizar. Para ser precisos, no es parte del ciclo de eventos, pero se ejecuta entre las diferentes fases del ciclo de eventos.

Node.js primero ejecuta las micro tareas, que incluyen promesas resueltas directamente, así como funciones de devolución de llamada que registra con la función `queueMicrotask`. Luego se ejecutan las tareas asíncronas, que puedes registrar, por ejemplo, con las funciones de temporización `setImmediate`, `setTimeout` o `setInterval`.

## TextEncoder y TextDecoder

Las clases `TextEncoder` y `TextDecoder` están destinadas a trabajar con `TypedArrays` en JavaScript. Estas son estructuras de datos que funcionan con datos binarios sin procesar. Las instancias de la clase `buffer` de Node.js también son instancias de `Uint8Array` y `TypedArray`. Existen incompatibilidades menores entre las clases `buffer` y `TypedArray`, pero en general ambas se comportan de una manera muy similar y se utilizan para intercambiar datos dentro de una aplicación. En el listado 16, puedes ver cómo puedes convertir una cadena en un `Uint8Array` y luego decodificarla nuevamente.

Listado 16: Uso de `TextEncoder` y `TextDecoder`.

```
const textEncoder = new TextEncoder();
const encodedString = textEncoder.encode('Hello World');
console.log(encodedString); // Output: Uint8Array(11) [72,101,108,108,
// 111,32,87,111,114,108,100]

const textDecoder = new TextDecoder();
const decodedString = textDecoder.decode(encodedString);
console.log(decodedString); // Output: Hello World
```

## URL y URLSearchParams

En Node.js, a menudo se trabaja con URL, por lo que la plataforma te proporciona la clase `URL` y `URLSearchParams` de manera global (consulta el listado 17), para que no tengas que cargarlos por separado.

Listado 17: Trabajo con URL en Node.js.

```
const url = new URL('/dist/latest-v16.x/docs/api/',
  'https://nodejs.org');

console.log(url.href); // Output: https://nodejs.org/dist/
// latest-v16.x/docs/api/
```

```
const searchParams = new URLSearchParams();
searchParams.set('name', 'john');
searchParams.set('age', 42);
console.log(searchParams.toString()); // Output: name=john&age=42
```

La clase URL representa los diversos aspectos de una URL, como el protocolo, el nombre de host o la ruta. Puedes utilizar la clase URLSearchParams de forma implícita junto con una instancia de URL y la propiedad searchParams, o de forma explícita como una instancia independiente.

## WebAssembly

El objeto WebAssembly controla el acceso a las interfaces WebAssembly del motor V8. Con esta interfaz, puedes trabajar con WebAssembly dentro de tu aplicación Node.js y, por ejemplo, subcontratar componentes críticos para el rendimiento.

## Sistemas de módulos de JavaScript

Ya conociste los sistemas de módulos de Node.js en el enfoque cuando analizamos los módulos principales de Node.js. En las siguientes secciones, analizaremos más de cerca los sistemas de módulos y aprenderás a dividir tu aplicación en varios archivos en los sistemas de módulos CommonJS y ECMAScript, así como también cómo funcionan juntos los sistemas de módulos.

### CommonJS

En Node.js, un módulo es un ámbito auto encapsulado para variables. Esto significa que puedes definir variables en el nivel superior de un archivo y que solo son válidas en el módulo. Esto evita la definición accidental de variables globales. Sin embargo, esto también significa que, sin el sistema de módulos, no tienes forma de usar la funcionalidad de tu módulo en tu aplicación. Un módulo nunca se representa a sí mismo; en cambio, es un componente de una aplicación más grande. Con el objeto de exportaciones o module.exports, puedes definir la interfaz de tu módulo, que luego puedes usar en otros módulos. El listado 18 contiene un ejemplo de una exportación de una función simple.

Listado 18: Exportación de una función (add.js).

```
module.exports = function(a, b) {
  return a + b;
};
```

El sistema de módulos CommonJS permite cargar módulos representados por archivos mediante la función `require`. Esta función devuelve un objeto que representa la interfaz pública del archivo. Puedes utilizar la funcionalidad del módulo a través de este objeto. El listado 19 muestra cómo puedes incluir la función exportada anteriormente.

Listado 19: Uso de módulos (`index.js`).

```
const add = require('./add');
const result = add(1, 2);
console.log('result: ', result);
```

En el listado 19, se supone que has guardado el código fuente del listado 18 en un archivo llamado `add.js`. Puedes elegir los nombres de tus módulos, así como su ubicación. Solo asegúrate de que la especificación de la ruta para la importación sea correcta. Al importar, pasa la ruta al archivo que deseas incluir a la función `require`. En este contexto, puedes elegir entre rutas absolutas y relativas. Una ruta absoluta comienza desde la raíz del sistema de archivos; una ruta relativa comienza desde el archivo actual. Especificar la extensión del archivo durante la importación es opcional.

Al cargar módulos mediante el método `require`, el cargador de módulos no solo busca el archivo especificado, sino también el nombre del archivo. Si no se especifica ninguna extensión de archivo, el cargador de módulos agrega las extensiones `.js`, `.json` y `.node`.

## Módulos ECMAScript

El principio básico de los módulos ECMAScript es similar al de los módulos CommonJS. La mayor diferencia entre ambos sistemas de módulos se puede encontrar en la sintaxis. A partir de la versión 8.5 de Node.js, los módulos ECMAScript forman parte de la plataforma. Mientras tanto, han alcanzado el índice de estabilidad 2, es decir, estables, por lo que puedes usarlos sin dudarlo. Como ya sabes, el sistema de módulos CommonJS sigue siendo actualmente el predeterminado en Node.js, por lo que debes habilitar explícitamente el sistema de módulos ECMAScript. Puedes lograr esto utilizando una de las siguientes medidas:

- Extensión de archivo `.mjs`
- Campo `type` en el `package.json` con el valor del módulo
- Opción `—input-type` en la línea de comandos con el valor del módulo

El alcance de las variables permanece limitado al nivel del módulo, como es el caso de los módulos CommonJS. Puedes utilizar la palabra clave `export` para especificar que un objeto, una clase o una función se pueden utilizar desde fuera del módulo.

El listado 20 contiene el código fuente de un módulo que exporta una función.

Listado 20: Exportación de una función (add.mjs).

```
export function add(a, b) {
  return a + b;
}
```

Puedes importar esta función utilizando la palabra clave import.

Para que funcione el código del listado 21, debes guardar la función add del listado 20 en un archivo llamado add.mjs.

Listado 21: Importación de una función (index.mjs).

```
import { add } from './add.mjs';

const result = add(1, 2);
console.log('result: ', result);
```

Al igual que con los módulos CommonJS, el nombre y la ubicación del archivo son arbitrarios, y puedes utilizar rutas absolutas y relativas. Sin embargo, deberías preferir las rutas relativas para tu aplicación en ambos casos, ya que esto proporciona una mejor reutilización de tu aplicación en otros sistemas.

El sistema de módulos ECMAScript ofrece un grado de flexibilidad ligeramente superior al de CommonJS. La tabla 6 resume las diferentes variantes de exportación.

Tabla 6: Exportaciones en ECMAScript.

Export statement	Type
export { var1, var2 as alias }	Named
export let var1, var2;	Named
export default <expression>	Default
export * from 'filename'	Default
export { var1, var2 as alias } from 'filename'	Named

Como puedes ver en la tabla 6, hay dos tipos de exportaciones. Las exportaciones con nombre (named) te permiten exportar variables, funciones o clases. Con una exportación predeterminada (default), exportas cualquier expresión. Solo puede haber una exportación predeterminada por módulo, pero puede haber cualquier cantidad de exportaciones con nombre. Para las exportaciones con nombre, la palabra clave as también te permite cambiar el nombre del objeto exportado.

Un módulo puede tener una exportación predeterminada y una o más exportaciones con nombre; esto significa que también puedes combinar las dos. Las importaciones brindan un



tipo de flexibilidad similar a las exportaciones. La tabla 7 contiene una descripción general de las diferentes variantes.

Tabla 4.7 Importaciones en ECMAScript.

Import statement	Type
<code>import * as moduleContent from 'filename';</code>	Named
<code>import {var1, var2 as alias} from 'filename';</code>	Named
<code>import 'filename';</code>	Named
<code>import defaultVar from 'filename'</code>	Default
<code>import defaultVar as aliasVar from 'filename';</code>	Default
<code>import * as defaultVar from 'filename';</code>	Default

Con respecto a las importaciones, se hace una distinción general entre una importación con nombre y una importación predeterminada. Las primeras tres filas de la tabla se refieren a importaciones con nombre y las últimas tres filas se refieren a importaciones predeterminadas. Si un módulo contiene una exportación predeterminada y una importación con nombre, puedes combinar la segunda y cuarta filas de la tabla, como se muestra en el listado 22.

Listado 22: Importaciones predeterminadas y con nombre (index.mjs).

```
import divide, {add} from './module.mjs';
console.log(divide(4, 2));
console.log(add(2,2));
```

Puedes encontrar las exportaciones correspondientes en el listado 23.

Listado 23: Módulo con exportaciones predeterminadas y con nombre (module.mjs).

```
export function add(a, b) {
  return a + b;
}
export default function(a, b) {
  return a / b;
}
```

Puedes utilizar módulos ECMAScript y CommonJS en tu aplicación. Sin embargo, no puedes cargar módulos ECMAScript utilizando la función `require`. En ese caso, deberás utilizar la función `import`. Sin embargo, el sistema de módulos ECMAScript te permite cargar módulos CommonJS. Si importas un módulo de este tipo, se comporta como si fuera un módulo ECMAScript con una exportación predeterminada. Esto se aplica tanto a sus propios módulos como a los módulos npm.

## Creación y uso de módulos propios

Si estás trabajando en una aplicación Node.js, debes hacerla lo más modular posible; es decir, dividir el código fuente en varios archivos. Una de las razones más importantes para este tipo de estructuración es la claridad. En un archivo de varios miles de líneas de código fuente, encontrar secciones de código específicas se vuelve cada vez más difícil. También hace que sea casi imposible reutilizar bloques de código en diferentes aplicaciones.

Pero la reutilización dentro de una aplicación también es mucho más conveniente con módulos autónomos, cada uno en su propio archivo. Una vez que tu aplicación se vuelve más extensa y compleja, y comienzan a surgir áreas con funcionalidad autónoma con las que se puede comunicar a través de interfaces uniformes, es el momento adecuado para externalizar este código fuente e integrarlo como un módulo. Una señal segura de que comienzas a modularizar es cuando comienzas a copiar bloques de código completos. En este caso, debes intercambiar el bloque de código y generalizarlo para que pueda usarse en varios lugares.

### Módulos en Node.js: CommonJS

Para tu aplicación Node.js, debe seguir este principio: una estructura, un archivo. Esto significa que todas las unidades lógicas de tu aplicación se encuentran en sus propios archivos. La ventaja aquí es que cada archivo trata solo de un tema y tiene una única interfaz definida con el resto de la aplicación, por lo que las diferentes partes de tu aplicación se pueden localizar de forma rápida y sencilla.

---

#### Acoplamiento flexible, cohesión estrecha

Este tipo de modularización utiliza el principio de acoplamiento flexible, lo que significa que las estructuras (es decir, funciones, objetos o clases) de una aplicación solo están conectadas de forma flexible a través de interfaces. Esto, a su vez, tiene la ventaja de que puedes reemplazar módulos individuales sin tener que reconstruir todo el sistema.

Los componentes de los módulos individuales deben estar estrechamente relacionados entre sí, lo que se conoce como cohesión estrecha. Esto significa que solo la lógica que realmente pertenece al tema del módulo se coloca en un módulo. Todo lo demás se intercambia a un módulo separado y se aborda a través de interfaces.

Como resultado, tienes una aplicación que consta de numerosos módulos manejables, lo que en última instancia conduce a una mejor capacidad de mantenimiento, mayor paralelización de tareas, mejor capacidad de prueba del código fuente y mejor capacidad de respuesta a los requisitos cambiantes.

---

Normalmente, cuando creas tu aplicación, agrupas tus módulos según temas. Cuando externalizas bloques de código que se usan varias veces, el truco es encontrar el equilibrio adecuado entre generalización y esfuerzo.

Como guía, debes tener en cuenta que siempre debes resolver principalmente tu problema actual y solo generalizar lo que sea necesario para tu aplicación.

Si escribes código que es demasiado genérico, automáticamente se vuelve más complejo, más difícil de leer y más costoso de implementar.

En la práctica, tus módulos generalmente exportan clases que instancias en otros módulos o funciones y objetos con los que puedes interactuar directamente. Un buen ejemplo lo proporcionan los módulos que Node.js pone a tu disposición.

### **Módulos Node.js personalizados**

En el siguiente ejemplo, implementarás una función para contar la frecuencia de las palabras en una oración. Esta función se exporta a través del sistema de módulos. La primera variante muestra la implementación en el sistema de módulos CommonJS y la segunda la muestra en el sistema de módulos ECMAScript.

En primer lugar, en el listado 24, se almacenan un punto y una coma en una expresión regular para ignorar estos caracteres durante el procesamiento. El carácter de espacio como separador de palabras también se almacena en una constante. La función `wordCount` acepta una cadena en la que primero se reemplazan los puntos y las comas con una cadena vacía. La cadena preparada de esta manera se convierte a minúsculas para ignorar la distinción entre mayúsculas y minúsculas.

Luego, se utiliza el método `split` para convertir la cadena en un arreglo de palabras. El recuento real de palabras se lleva a cabo en el método `reduce`. Se utiliza un objeto vacío como valor inicial. Las palabras individuales se utilizan como llaves en el objeto. Para cada palabra, se aumenta un valor existente en uno o se establece el valor inicial en uno si la palabra aún no existe como llave en el objeto.

Listado 24: Implementación de la función “`wordCount`” como un módulo CommonJS (`word-count.js`).

```
const ignore = /[.,]/g;
const separator = ' ';
module.exports = function wordCount(sentence) {
  return sentence
    .replace(ignore, "")
    .toLowerCase()
```

```
.split(separator)
.reduce((prev, current) => {
  prev[current] = prev[current] + 1 || 1;
  return prev;
}, {});
}
```

La función se pone a disposición como una interfaz del módulo mediante el objeto `module.exports`, por lo que puedes incluirla en tu aplicación mediante la función `require`.

El código fuente del listado 25 muestra cómo se puede utilizar el módulo que acabas de crear. Utiliza la declaración `wc = require('./word-count')` para incluir el módulo en la aplicación actual. Esto supone que has guardado el código fuente del listado 24 en un archivo llamado `word-count.js`. Todas las funciones y objetos que has asignado como propiedades al objeto `module.exports` en el módulo están disponibles.

Listado 25: Uso del módulo “wordCount” (`index.js`).

```
const wc = require('./word-count');
const sentence = 'Where there is much light, there is also much
shadow.';
const wordCount = wc(sentence);
console.log(sentence);
for (let i in wordCount) {
  console.log(wordCount[i] + ' x ' + i);
}
```

---

## Nombrar archivos y estructuras de JavaScript

A diferencia de JavaScript, no todos los sistemas de archivos distinguen entre mayúsculas y minúsculas. Para evitar problemas en este contexto, solo debes utilizar letras minúsculas al nombrar tus archivos. Si un nombre de archivo o directorio consta de varias palabras, sepáralas con un guion. Por otro lado, la designación `CamelCase` ahora es mucho más común. Aquí, las palabras individuales en el nombre del archivo se introducen con una letra mayúscula.

Ten en cuenta que no puedes utilizar guiones en los nombres de variables en JavaScript. Esto es posible a través de la notación de arreglo al acceder a las propiedades de los objetos, pero aún así debes evitarlo. Los objetos y funciones de JavaScript comienzan con una letra minúscula. De lo contrario, puedes utilizar la notación `CamelCase`. Solo los nombres de clase comienzan con una letra mayúscula.

---

Como has visto en el listado 25, solo se publican los contenidos e interfaces de un módulo que están disponibles a través del objeto de exportaciones. Todas las demás variables,

funciones o clases definidas dentro de un módulo solo están disponibles en ese módulo. Por lo tanto, en este caso, no tienes que preocuparte por mantener limpio el alcance global de tu aplicación.

## Módulos en Node.js: ECMAScript

Hasta ahora, hemos utilizado la extensión de archivo .mjs para los módulos ECMAScript en todos los ejemplos. En este ejemplo, utilizamos el campo type en el archivo package.json para activar el sistema de módulos ECMAScript. Para ello, primero debes ejecutar el comando `npm init -y` en el directorio donde quieres crear tu archivo inicial. Este comando crea un package.json estándar. Luego debes insertar el campo type, como se muestra en el listado 26.

Listado 26: “package.json” con el campo “type”.

```
{
  "name": "node-book",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "type": "module",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Con esta preparación, puedes transferir el ejemplo anterior al sistema de módulos ECMAScript con solo unos pocos ajustes. En el listado 27, puedes encontrar el código fuente personalizado del módulo wordCount. Al realizar ajustes en el archivo package.json, también puedes mantener la extensión de archivo .js.

Listado 27: Módulo “wordCount” como módulo ECMAScript (word-count.js).

```
const ignore = /[.,]/g;
const separator = ' ';
export function wordCount(sentence) {
  return sentence
    .replace(ignore, '')
    .toLowerCase()
    .split(separator)
    .reduce((prev, current) => {
```

```

    prev[current] = prev[current] + 1 || 1;
    return prev;
  }, {});
}

```

El cambio aquí es que en lugar de asignarlo a `module.exports`, utilizas la palabra clave `export`. En este caso, creas una exportación con nombre que puedes usar en tu aplicación. La personalización en el listado 25 también está limitada a solo una línea, como puedes ver en el listado 28.

Listado 28: Integración del módulo “wordCount”.

```

import { wordCount as wc } from './word-count.js';
const sentence = 'Where there is much light, there is also much
shadow.';
const wordCount = wc(sentence);
console.log(sentence);
for (let i in wordCount) {
  console.log(wordCount[i] + ' x ' + i);
}

```

En el listado 28, importas la función `wordCount` desde el archivo `word-count.js` utilizando la palabra clave `import`. Debido a que este nombre ya se usa en el código fuente, la función se renombra a `wc` utilizando la palabra clave `as`. El resto del código fuente permanece sin cambios.

---

### Extensión de nombre de archivo requerida

Como has visto en los ejemplos de este documento, especificar la extensión del nombre de archivo es obligatorio al cargar módulos en el sistema de módulos ECMAScript. Esto se aplica tanto a las especificaciones de ruta absoluta como relativa. Si omites la extensión, Node.js devuelve un error correspondiente.

---

### Exportación de diferentes tipos de datos

Al utilizar el sistema de módulos Node.js, no estás limitado a utilizar únicamente funciones, ya que puedes exportar cualquier tipo de datos. Por lo tanto, es posible exportar clases u objetos o simplemente proporcionar valores simples como una cadena o un número. El listado 29 contiene un ejemplo en el que se exporta un objeto con varias funciones.

Listado 29: Exportación de objetos.

```

function add(a, b) {
  return a + b;
}

```

```
function subtract(a, b) {  
  return a - b;  
}  
module.exports = { add, subtract };
```

El listado 30 muestra la misma funcionalidad, pero en la sintaxis del módulo ECMAScript.

Listado 30: Exportación de objetos en módulos ECMAScript.

```
function add(a, b) {  
  return a + b;  
}  
function subtract(a, b) {  
  return a - b;  
}  
export { add, subtract };
```

En los ejemplos que se muestran en el listado 29 y el listado 30, puedes ver que la implementación y la exportación son procesos separados. Esta es una práctica recomendada para que el código fuente sea más claro. Siempre debes colocar las exportaciones al final del archivo para que todos los desarrolladores que trabajen con tu código sepan dónde encontrar la definición de la interfaz del módulo. Otra característica única de este ejemplo es que utiliza la notación abreviada introducida con ECMAScript 2015 para las propiedades de los objetos, donde el nombre de la propiedad y la variable que contiene el valor tienen la misma designación.

Hay casos en los que es necesario manipular el comportamiento de un módulo desde el exterior. Puedes hacerlo exportando una función que recibe argumentos desde el exterior y genera un valor de retorno basado en ellos.

El código fuente del listado 31 aclara el ejemplo.

Listado 31: Manipulación de módulos.

```
export default function(DEBUG) {  
  return {  
    options: {  
      outputStyle: DEBUG ? 'expanded' : 'compressed',  
      sourceMap: DEBUG,  
      sourceMapEmbed: true,  
    },  
    files: {  
      'style/style.css': 'style/style.scss',  
    },  
  };  
};
```

En el listado 31, transfieres un valor booleano que afecta al objeto devuelto. Esta es la configuración intercambiada de partes de una configuración de grunt para el preprocesador CSS, Sass.

## **El módulo modules**

El módulo modules representa el cargador de módulos CommonJS. Este módulo no debe utilizarse junto con módulos ECMAScript.

Los dos componentes más importantes del módulo modules son el objeto de exportaciones o `module.exports`, que puedes utilizar para publicar módulos, y la función `require`, que puedes utilizar para incluir el módulo. Sin embargo, además de estas dos características principales, este módulo te ofrece una serie de otras funcionalidades que pueden resultar muy útiles en el desarrollo y el funcionamiento de los módulos. Para este propósito, la plataforma Node.js te proporciona el objeto módulo de forma global. Si utilizas el sistema de módulos ECMAScript, este no es el caso y la variable módulo no está definida. Puedes utilizar este objeto para consultar información diversa sobre el módulo actual.

En primer lugar, se deben mencionar aquí las propiedades `module.id` y `module.filename`. Estas dos propiedades normalmente especifican el nombre de archivo absoluto del módulo. Si deseas utilizar esta información como base, debes utilizar la segunda variante, es decir, `module.filename`. La propiedad `module.id` contiene el valor `.` cuando accedes a ella fuera de un módulo, como en el archivo principal de tu aplicación. También puedes utilizar estas dos propiedades del objeto módulo en el REPL de Node.js. En este caso, la propiedad `id` tiene el valor `repl` y la propiedad `filename` tiene el valor `null`.

Puedes utilizar la propiedad `load` del objeto módulo para averiguar si el módulo actual ya se ha cargado. En este caso, el valor es `true`. Si el módulo actual aún no se ha cargado o todavía está en proceso de carga, el valor de esta propiedad es `false`.

Cuando se inicia una aplicación Node.js, la propiedad `require.main` se completa automáticamente con el objeto módulo del archivo inicial. Por lo tanto, puedes acceder al nombre de archivo del archivo que sirve como punto de entrada a tu aplicación en cada módulo a través de `require.main.filename`.

Además de esta información, el módulo modules también contiene datos sobre las relaciones entre los módulos, es decir, qué módulo fue cargado por qué otro módulo o qué módulos son cargados por un módulo en particular. La propiedad `parent` del objeto módulo proporciona información sobre el módulo que cargó el módulo actual. Nuevamente, esta información viene en forma de un objeto módulo, lo que significa que tienes acceso a las propiedades id



o filename del módulo padre, entre otras. La propiedad `children` contiene un arreglo de módulos que el módulo actual carga a través de `require`.

Sin embargo, esto solo se aplica a tus propios módulos. Los módulos principales no se enumeran en el arreglo `children`. Los objetos en esta estructura de datos tienen la misma estructura que el objeto padre.

## Cargador de módulos

La inclusión de módulos se realiza mediante la función `require` o mediante la declaración `import`. Para ello, debes confirmar el nombre del archivo que contiene el código fuente del módulo. Puedes especificar el nombre del archivo como un nombre absoluto y relativo al archivo actual.

Suponiendo una estructura de directorio como la del listado 32, es decir, los dos archivos JavaScript `wordcount.js` e `index.js`, donde `index.js` es el punto de entrada a tu aplicación, tu aplicación se inicia mediante el comando `node index.js`. Luego puedes incluir el módulo `wordcount` mediante una ruta relativa usando `require('./word-count')` o `import wordCount from './word-count'`, o mediante la ruta absoluta, es decir, usando `require('/srv/node/word-count/word-count')` o `import wordCount from '/srv/node/word-count/word-count'`. Como ya se mencionó, debes utilizar la variante relativa, que te permite ejecutar la aplicación en otros sistemas o ponerla a disposición de otras personas

Listado 32: Listado de directorios de la aplicación “wordCount”.

```
$ ls /srv/node/word-count
word-count.js index.js package.json
```

## Resolución de módulos del administrador de paquetes de Node

Al buscar módulos npm, el cargador de módulos CommonJS y el cargador de módulos ECMAScript se comportan de la misma manera. Puedes cargar módulos npm sin anteponer `/` o `./` al módulo que deseas cargar. En este caso, el cargador de módulos busca un módulo principal de Node.js o busca un paquete con el nombre correspondiente en el subdirectorio `node_modules` del directorio actual. Si no se encuentra el paquete allí, se intenta resolver el nombre del paquete en un nivel superior. En la estructura de directorios del listado 32 esto significa que la búsqueda se realizará primero en el directorio `/srv/node/word-count/node_modules`, luego en `/srv/node/node_modules`, y así sucesivamente hasta que el cargador de módulos haya llegado al directorio raíz. Si el cargador de módulos no puede encontrar el paquete en esta estructura de directorios, busca en los directorios especificados en la variable de entorno `NODE_PATH`. También busca en los directorios `<home>/.node_modules`, `<home>/.node_modules` y `<install-prefix>/lib/node`. Esta forma de

buscar módulos te permite instalar diferentes versiones de paquetes npm en tu sistema y controlar qué versión debe cargar el cargador de módulos mediante la ubicación de cada paquete. Esto te permite evitar conflictos de versiones causados por otra aplicación o biblioteca que requiere una versión diferente de un paquete.

Dependiendo de dónde almacenes los paquetes, puedes influir en la velocidad del proceso de carga. La mejor práctica ha sido instalar los paquetes en el directorio raíz de la aplicación, si es posible. Los paquetes ubicados en el directorio local `node_modules` se cargan más rápido, lo que evita un largo proceso de búsqueda.

La siguiente lista resume brevemente en qué directorios se buscan los paquetes en el ejemplo:

- `/srv/node/word-count/node_modules`
- `/srv/node/node_modules`
- `/srv/node_modules`
- `/node_modules`
- `/home/<username>/.node_modules`
- `/home/<username>/.node_modules`
- `/usr/local/lib/node_modules`

En este sistema, la variable de entorno `NODE_PATH` está vacía, por lo que esto no agrega una ruta adicional.

## Importación de directorios

Con el cargador de módulos CommonJS de Node.js, existe otra forma de incluir módulos en tu aplicación.

En lugar de especificar el nombre de un módulo, también puedes especificar un directorio completo. La razón es que las bibliotecas a menudo se almacenan en sus propias estructuras de directorio. Para esos módulos, existe una convención de nomenclatura definida sobre cómo nombrar los archivos para que el cargador de módulos pueda encontrar el código fuente e incluirlo de manera correcta. Si, en lugar de un nombre de archivo, pasas un directorio al cargador de módulos, intenta encontrar un `package.json` en el primer paso. Puedes usar este archivo para señalar el punto de entrada de un módulo dentro de un directorio. Para demostrar este caso, debemos ajustar ligeramente el ejemplo del contador de palabras. Primero, debes crear un directorio llamado `word-count`. Este directorio debería contener tu módulo más adelante. En este directorio, nuevamente debes crear un subdirectorio llamado `lib`, y luego debes copiar el archivo `word-count.js` del listado 24 en él. A continuación, debes crear un archivo llamado `package.json` en el directorio `word-count`. El contenido de este archivo corresponde al código fuente del listado 33.

Listado 33: “package.json” para el Módulo “wordCount”.

```
{
  "name": "word-count",
  "main": "./lib/word-count.js",
}
```

El último ajuste debe implementarse en index.js del listado 25, es decir, en el punto de entrada de la aplicación. Aquí, solo debes asegurarte de que el módulo wordCount esté incluido mediante el comando `const wc = require('./word-count')`. Si especificas un directorio cuando deseas cargar un módulo y ese directorio no tiene un archivo package.json con un campo principal válido, el cargador de módulos busca alternativamente los archivos index.js e index.node como puntos de entrada al módulo.

## Caché de módulos

Dentro de tu aplicación, puedes cargar un módulo no solo una vez al comienzo de un archivo de código fuente, o incluso solo en un solo archivo, sino también tantas veces como desees, en diferentes lugares y, por lo tanto, varias veces. Para evitar tener que leer estos módulos desde el sistema de archivos varias veces, tanto los cargadores CommonJS como ECMAScript realizan una optimización almacenando el módulo en la memoria caché. Esto significa que el módulo solo necesita cargarse y ejecutarse la primera vez. Las llamadas de requerimiento o importación restantes con el nombre del módulo se proporcionan luego desde la caché. Como resultado, esta característica garantiza que el código fuente solo se ejecute en la primera llamada de requerimiento o importación. Si deseas obtener un efecto secundario en cada llamada, debes forzar esto a través de otras funciones. En el listado 34 y el listado 35, puedes ver cómo se puede recrear este comportamiento.

Listado 34: “my-module.js”.

```
console.log('myModule called');
```

Listado 35: “index.js”.

```
require('./my-module'); // Output: myModule called
require('./my-module.js'); // no output
```

Cuando ejecutas el código fuente a través del comando `node index.js`, obtienes la salida myModule llamada solo una vez porque la segunda llamada require se proporciona directamente desde la memoria caché y el archivo my module.js no se ejecuta una segunda vez. El mismo resultado ocurre si personalizas el archivo index.js como se muestra en el listado 36 y usa el sistema de módulos ECMAScript.

Listado 36: Múltiples importaciones del mismo módulo en el sistema de módulos ECMAScript.

```
import './my-module.js'; // Output: myModule called
import './my-module.js'; // no output
```

Al igual que el sistema de módulos CommonJS, el sistema de módulos ECMAScript tiene una memoria caché local. Sin embargo, ambas memorias caché son independientes entre sí. Otra diferencia es que puedes manipular la memoria caché del módulo CommonJS mucho más fácilmente.

Hay varias formas de lograr un efecto múltiple al cargar un módulo. Por ejemplo, puedes cargar el módulo, borrar la memoria caché del módulo y cargar el módulo nuevamente. Sin embargo, la mejor variante es prescindir de cualquier efecto secundario al cargar un módulo y encapsularlo en una función. De esta manera, solo tienes que cargar el módulo una vez y puedes llamar a la función devuelta tantas veces como desees. El listado 37 y el listado 38 muestran cómo se ve esto para el sistema de módulos CommonJS, y el listado 39 y el listado 40 muestran lo mismo para el sistema de módulos ECMAScript.

Listado 37: Función exportada en CommonJS (my-module.js).

```
module.exports = function () {
  console.log('myModule called');
};
```

Listado 38: Múltiples llamadas de la función exportada en CommonJS (index.js).

```
const myModule = require('./my-module');

myModule(); // Output: myModule called
myModule(); // Output: myModule called
```

Listado 39: Función exportada en ECMAScript (my module.js).

```
export default function () {
  console.log('myModule called');
}
```

Listado 40: Múltiples llamadas de la función exportada en ECMAScript (index.js).

```
import myModule from './my-module.js';

myModule(); // Output: myModule called
myModule(); // Output: myModule called
```

## **require Functionality**

La estructura del cargador de módulos CommonJS de Node.js es relativamente simple. Sin embargo, puede resultar difícil realizar un seguimiento de las aplicaciones con muchos módulos. También es un desafío manejar diferentes versiones de ciertos módulos en un sistema. El mecanismo de almacenamiento en caché del cargador de módulos también puede convertirse en un problema en determinadas circunstancias. Además de las funcionalidades del sistema de módulos Node.js presentadas hasta ahora, existen otras características que pueden facilitarte el trabajo con módulos.

El método `resolve` del objeto `require` te permite averiguar en qué archivo se encuentra un determinado módulo. Para ello, solo tienes que proporcionar al método la información sobre qué módulo deseas cargar. El esquema de nombres es similar al que se utiliza en las operaciones de carga habituales. El método `resolve` también realiza las mismas operaciones de búsqueda que el método `require` real, la única diferencia es que el módulo no se carga, sino que se devuelve la ruta absoluta del archivo en el que se encuentra el módulo como una cadena. Para los módulos principales, `resolve` devuelve solo el nombre del módulo. Además, esta funcionalidad no solo está disponible en las aplicaciones, sino también en el REPL de Node.js.

Ya has aprendido sobre la caché de módulos de Node.js. Puedes leer y editar esta caché a través de la propiedad `cache` del objeto `require`. El objeto de caché es un objeto JavaScript normal en el que se almacena la información. Las llaves consisten en los nombres de archivo absolutos de los módulos respectivos, donde en cada caso el valor es el objeto de módulo del módulo. Por lo tanto, puedes utilizar el objeto de caché para comprobar si un módulo ya se ha cargado dentro de tu aplicación. Sin embargo, hay un aspecto importante que debes tener en cuenta: no hay módulos principales enumerados en la caché. En el ejemplo de caché de módulo en el listado 34 y el listado 35, viste que un módulo se evalúa solo una vez y se proporciona desde la caché en la segunda llamada. El objeto de caché te permite cambiar este comportamiento.

Si eliminas la entrada de caché para el módulo después de la primera llamada `require`, como se muestra en el listado 41, el módulo se eliminará completamente de la caché, lo que hará que se vuelva a cargar y evaluar la próxima vez que se llame, y cualquier efecto secundario que se pretenda que tenga el módulo se producirá nuevamente. Puedes manipular no solo la caché, sino también la forma en que Node.js carga los módulos que incluyes a través de `require`. Para este propósito, el módulo `require` tiene la propiedad `extensions`. El valor es un objeto donde las llaves consisten en varias extensiones de archivo, y los valores asociados consisten en las funciones utilizadas para cargar los archivos con las extensiones correspondientes. Ahora, si deseas agregar soporte para otra extensión de archivo, todo lo que necesitas hacer es extender el objeto de `extensions` con una llave que tenga el nombre

de la extensión de archivo y luego agregar una función que sea responsable de cargar los archivos. La forma más fácil de hacer esto es simplemente usar una de las funciones existentes. En un tema posterior, implementarás una aplicación un poco más extensa donde podrás ver los diversos aspectos de Node.js en acción.

Listado 41: “app.js”.

```
require('./my-module'); // Output: myModule called
delete require.cache[require.resolve('./my-module')];
require('./my-module'); // Output: myModule called
```

## Resumen

Este documento se han presentado los componentes principales de Node.js y se ha resaltado las diferencias entre los distintos niveles de módulos. Has visto qué módulos están incluidos en la plataforma y cómo puedes utilizarlos.

Un componente importante de Node.js es el sistema de módulos, que te permite dividir tu aplicación en secciones más pequeñas e independientes que interactúan entre sí a través de interfaces definidas. Node.js está experimentando una renovación importante en este momento en términos del sistema de módulos.

Durante muchos años, los desarrolladores han basado sus esfuerzos en el sistema de módulos CommonJS, mientras que ahora Node.js se está moviendo gradualmente al sistema de módulos ECMAScript. Mientras tanto, el nuevo sistema de módulos es tan estable que puedes usarlo en una aplicación de producción sin ningún problema.

Sin embargo, el sistema de módulos CommonJS todavía representa el valor predeterminado, por lo que debes habilitar el nuevo sistema de módulos a través de una extensión de archivo especial, una entrada en el archivo package.json o un indicador de línea de comandos.