Entendiendo la concurrencia de Node.js

El desarrollo web del lado del servidor se caracteriza por procesar grandes volúmenes de solicitudes HTTP de la forma más rápida y eficiente posible. JavaScript es diferente de otros lenguajes y plataformas porque tiene un único hilo de ejecución, lo que significa que las solicitudes HTTP se procesan una a la vez. Sin embargo, detrás de escena, hay mucho más en juego y en este documento se explica por qué el enfoque de JavaScript es inusual, cómo la API de Node.js realiza el trabajo en nombre del código JavaScript y cómo se pueden crear hilos de ejecución adicionales para manejar tareas computacionalmente intensivas. La tabla 1 pone la concurrencia de JavaScript en contexto.

Tabla 1: Poniendo la concurrencia de Node.js en contexto.

Pregunta	Respuesta
¿Qué es?	La concurrencia es la ejecución de múltiples hilos de código. Node.js tiene soporte para la concurrencia, pero oculta los detalles al desarrollador.
¿Por qué es útil?	La concurrencia permite a los servidores lograr un mayor rendimiento al aceptar y procesar múltiples solicitudes HTTP simultáneamente.
¿Cómo se usa?	Node.js tiene un único hilo de ejecución para el código JavaScript, llamado hilo principal, y se basa en eventos para coordinar el trabajo necesario para procesar diferentes hilos de trabajo. La API de Node.js hace un uso extensivo de la ejecución concurrente en sus APIs, pero esto está en gran medida oculto para el desarrollador.
¿Existen limitaciones o inconvenientes?	Se debe tener cuidado de no bloquear el hilo principal; de lo contrario, el rendimiento se verá afectado.
¿Existen alternativas?	No. El modelo de concurrencia es fundamental para Node.js y comprenderlo es esencial para crear aplicaciones web que se escalen de manera económica.

Preparación para esta práctica

Para crear el proyecto para esta práctica, abre un nuevo símbolo del sistema, navega hasta una ubicación conveniente y crea una carpeta llamada webapp. Ejecuta el comando que se muestra en el listado 1 en la carpeta webapp para crear el archivo package.json.

Listado 1: Inicialización del proyecto.

```
npm init -y
```

Ejecuta los comandos que se muestran en el listado 2 en la carpeta webapp para instalar los paquetes que se usarán para compilar archivos TypeScript y monitorear los archivos en busca de cambios.

Listado 2: Instalación de paquetes de herramientas.

```
npm install --save-dev typescript@5.2.2
npm install --save-dev tsc-watch@6.0.4
```

Ejecuta los comandos que se muestran en el listado 3 en la carpeta webapp para agregar los paquetes que configurarán el compilador TypeScript para proyectos Node.js y describirán los tipos utilizados por la API Node.js.

Listado 3: Agregar la configuración del compilador y los paquetes de tipos.

```
npm install --save-dev @tsconfig/node20
npm install --save @types/node@20.6.1
```

Para configurar el compilador TypeScript, crea un archivo llamado tsconfig.json en la carpeta webapp con el contenido que se muestra en el listado 4.

Listado 4: El contenido del archivo tsconfig.json en la carpeta webapp.

```
{
  "extends": "@tsconfig/node20/tsconfig.json",
  "compilerOptions": {
    "rootDir": "src",
    "outDir": "dist",
  }
}
```

Este archivo de configuración extiende el que proporcionan los desarrolladores de TypeScript para trabajar con Node.js. Los archivos TypeScript se crearán en la carpeta src y el JavaScript compilado se escribirá en la carpeta dist.

Abre el archivo package.json y agrega el comando que se muestra en el listado 5 a la sección de script para definir el comando que iniciará las herramientas de compilación.

Listado 5: Agregar un comando de script en el archivo package.json en la carpeta webapp.

```
{
  "name": "webapp",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
```

```
    "start": "tsc-watch --onsuccess \"node dist/server.js\""
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "@tsconfig/node20": "^20.1.4",
    "tsc-watch": "^6.0.4",
    "typescript": "^5.2.2"
  },
  "dependencies": {
    "@types/node": "^20.6.1"
  }
}
```

Crear una aplicación web simple

Con los paquetes y las herramientas de compilación en su lugar, es hora de crear una aplicación web simple. Crea la carpeta webapp/src y agrégale un archivo llamado handler.ts con el contenido que se muestra en el listado 6.

Listado 6: El contenido del archivo handler.ts en la carpeta src.

```
import { IncomingMessage, ServerResponse } from "http";

export const handler = (req: IncomingMessage, res: ServerResponse) => {
  res.end("Hello World");
};
```

Este archivo define el código que procesará las solicitudes HTTP. Describimos las características HTTP que proporciona Node.js en el siguiente documento, pero para este documento, es suficiente saber que la solicitud HTTP está representada por un objeto IncomingMessage y la respuesta se crea utilizando el objeto ServerResponse. El código del listado 6 responde a todas las solicitudes con un simple mensaje Hello World.

A continuación, agrega un archivo llamado server.ts a la carpeta src con el contenido que se muestra en el listado 7.

Listado 7: El contenido del archivo server.ts en la carpeta src.

```
import { createServer } from "http";
import { handler } from "./handler";

const port = 5000;
```

```
const server = createServer(handler);

server.listen(port, function() {
  console.log(`Server listening on port ${port}`);
});
```

Este código crea un servidor HTTP simple que escucha las solicitudes HTTP en el puerto 5000 y las procesa utilizando la función definida en el archivo handler.ts en el listado 6.

Agrega un archivo llamado data.json a la carpeta webapp con el contenido que se muestra en el listado 8. Este archivo se utilizará más adelante en la práctica.

Listado 8: El contenido del archivo data.json en la carpeta webapp.

```
{
  "products": [
    { "id": 1, "name": "Kayak", "category": "Watersports",
      "description": "A boat for one person", "price": 275 },
    { "id": 2, "name": "Lifejacket", "category": "Watersports",
      "description": "Protective and fashionable", "price": 48.95 },
    { "id": 3, "name": "Soccer Ball", "category": "Soccer",
      "description": "FIFA-approved size and weight",
      "price": 19.50 },
    { "id": 4, "name": "Corner Flags", "category": "Soccer",
      "description": "Give your playing field a professional touch",
      "price": 34.95 }
  ]
}
```

Ejecuta los comandos que se muestran en el listado 9 en la carpeta webapp para iniciar el observador que supervisará y compilará los archivos TypeScript y ejecutará el JavaScript que se produzca.

Listado 9: Inicio del proyecto.

```
npm start
```

El archivo server.ts en la carpeta src se compilará para producir un archivo JavaScript puro llamado server.js en la carpeta dist, que producirá el siguiente resultado cuando se ejecute:

```
Server listening on port 5000
```

Abre un navegador web y navega a http://localhost:5000 para enviar una solicitud al servidor HTTP, que producirá la respuesta que se muestra en la figura 1.

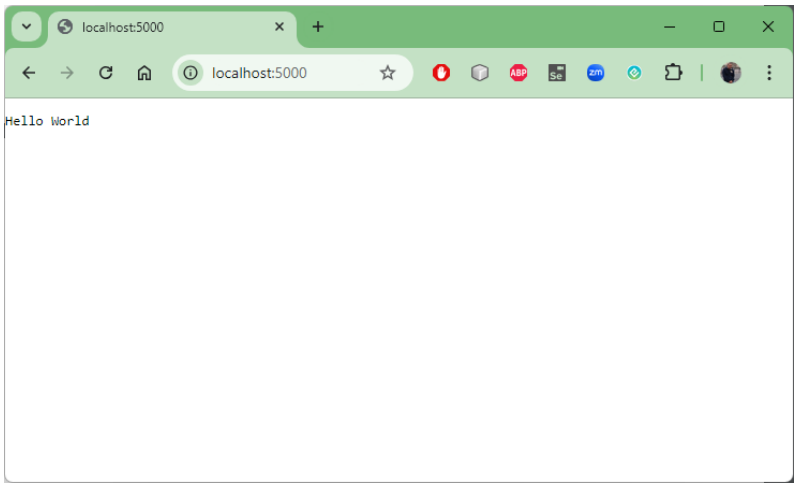

Figura 1: Ejecución de la aplicación de ejemplo.

Comprensión de la ejecución (simplificada) del código del servidor

Se requiere de una advertencia importante: este documento omite algunos detalles, es un poco vago con algunas explicaciones y difumina los límites entre algunos detalles finos.

Los temas tratados en este documento son complejos, con un sinfín de matices y detalles y terminología que significa cosas diferentes en diferentes plataformas. Y por eso, con la mente puesta en la brevedad, nos hemos centrado en lo que es importante para el desarrollo de aplicaciones web en JavaScript, aunque eso signifique pasar por alto algunos temas.

La concurrencia es un tema realmente fascinante y puede ser un área de investigación gratificante. Pero antes de profundizar en los detalles, ten en cuenta que para ser un desarrollador de JavaScript eficaz, solo necesitas una descripción básica de la concurrencia, como la que se presenta en este documento.

Comprensión de la ejecución multiproceso

Las aplicaciones web del lado del servidor deben poder procesar muchas solicitudes HTTP simultáneamente para escalar económicamente de modo que se pueda utilizar una pequeña cantidad de capacidad del servidor para admitir una gran cantidad de clientes.

El enfoque convencional consiste en aprovechar las características de subprocesos múltiples del hardware de servidor moderno mediante la creación de un grupo de subprocesos de procesamiento. Cuando llega una nueva solicitud HTTP, se agrega a una cola donde espera hasta que uno de los subprocesos esté disponible para procesarla. El subproceso procesa la solicitud, envía la respuesta de vuelta al cliente y luego regresa a la cola para la siguiente solicitud.

El hardware de servidor puede ejecutar varios subprocesos simultáneamente, como se ilustra en la figura 2, de modo que se pueda recibir y procesar un gran volumen de solicitudes simultáneamente.

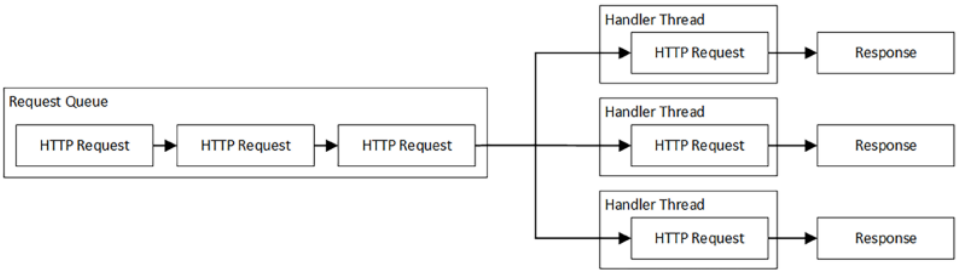

Figura 2: Manejo simultáneo de solicitudes HTTP.

Este enfoque aprovecha al máximo el hardware del servidor, pero requiere que los desarrolladores consideren cómo las solicitudes pueden interferir entre sí. Un problema común es que un subproceso de control modifica los datos mientras otro subproceso los lee, lo que produce un resultado inesperado.

Para evitar este tipo de problemas, la mayoría de los lenguajes de programación incluyen palabras clave que se utilizan para restringir las interacciones entre subprocesos. Los detalles varían, pero palabras clave como lock y asynchronous se utilizan para garantizar que los subprocesos utilicen de forma segura los recursos y los datos compartidos mediante la creación de regiones protegidas de código que solo pueden ser ejecutadas por un subproceso a la vez.

Escribir código que utiliza subprocesos es un equilibrio entre seguridad y rendimiento. Las regiones protegidas de código son cuellos de botella potenciales en el rendimiento y, si las protecciones se aplican de forma demasiado amplia, el rendimiento se ve afectado y la cantidad de solicitudes que se pueden procesar simultáneamente disminuye. Sin embargo, las solicitudes pueden interferir entre sí y producir resultados inesperados si las protecciones se aplican de forma demasiado dispersa.

Comprensión de las operaciones bloqueantes y no bloqueantes

En la mayoría de las aplicaciones del lado del servidor, el subproceso que procesa una solicitud HTTP pasa la mayor parte del tiempo esperando. Esto puede ser esperar a que una base de datos produzca un resultado, esperar el siguiente fragmento de datos de un archivo o esperar el acceso a una región protegida de código.

Cuando un hilo está esperando, se dice que está bloqueado. Un hilo bloqueado no puede realizar ningún otro trabajo hasta que se haya completado la operación que está esperando, tiempo durante el cual se reduce la capacidad del servidor para procesar solicitudes. En

aplicaciones con mucho trabajo, hay un flujo constante de nuevas solicitudes que llegan y tener hilos bloqueados sin hacer nada genera colas de solicitudes esperando ser procesadas y reduce el rendimiento general.

Una solución es utilizar operaciones no bloqueantes, también conocidas como operaciones asíncronas. Estos términos pueden ser confusos. La mejor forma de entenderlos es con un ejemplo del mundo real: una pizzería.

Imagina que, después de tomar un pedido, un empleado del restaurante fue a la cocina, armó su pizza, la puso en el horno, se quedó allí esperando a que se cocinara durante 10 minutos y luego te la sirvió. Este es el enfoque de bloqueo, o sincrónico, para preparar pizza. Los clientes estarán contentos si entran al restaurante cuando haya un empleado disponible para tomar un pedido, porque recibirán su pizza en el menor tiempo posible. Pero nadie más está contento. Los demás clientes de la cola no están contentos porque tienen que esperar en la cola mientras se preparan, cocinan y sirven las pizzas para todos los clientes que están delante de ellos, momento en el que un empleado estará disponible para preparar su pizza. El dueño del restaurante no está contento porque la producción de pizzas es igual al número de empleados, que pasan la mayor parte del tiempo esperando a que se cocine la pizza.

Hay un enfoque más sensato. A un empleado, al que llamaremos Bob, se le asigna la tarea de supervisar el horno. Los demás empleados toman los pedidos, preparan las pizzas y las colocan en el horno como antes, pero en lugar de esperar a que se cocinen, le piden a Bob que les diga cuándo está cocida la pizza.

Mientras Bob observa las pizzas en el horno, los empleados pueden seguir trabajando, tomando el pedido del siguiente cliente de la cola, preparando la siguiente pizza, y así sucesivamente. Bob puede observar muchas pizzas, por lo que el límite en la cantidad de pizzas que se pueden producir es el tamaño del horno y no la cantidad de empleados.

Cocinar una pizza se ha convertido en una operación sin bloqueos para todos, excepto para Bob. No hay forma de evitar esperar a que se encienda el horno, pero el rendimiento del restaurante mejora al hacer que una persona haga todo el trabajo de espera. Todos están contentos.

Bueno, casi. El propietario está contento porque el restaurante produce más pizzas. Los clientes en la cola están contentos porque los empleados pueden comenzar a trabajar en su pizza mientras Bob observa los pedidos anteriores. Pero los pedidos individuales pueden demorar más: Bob puede decirle a otro empleado que una pizza está lista, pero no podrá servirla si está ocupado con otro cliente. El rendimiento general del restaurante mejora, pero los pedidos individuales pueden demorar más en completarse.

El mismo enfoque se puede adoptar con las solicitudes HTTP, como se muestra en la figura 3.

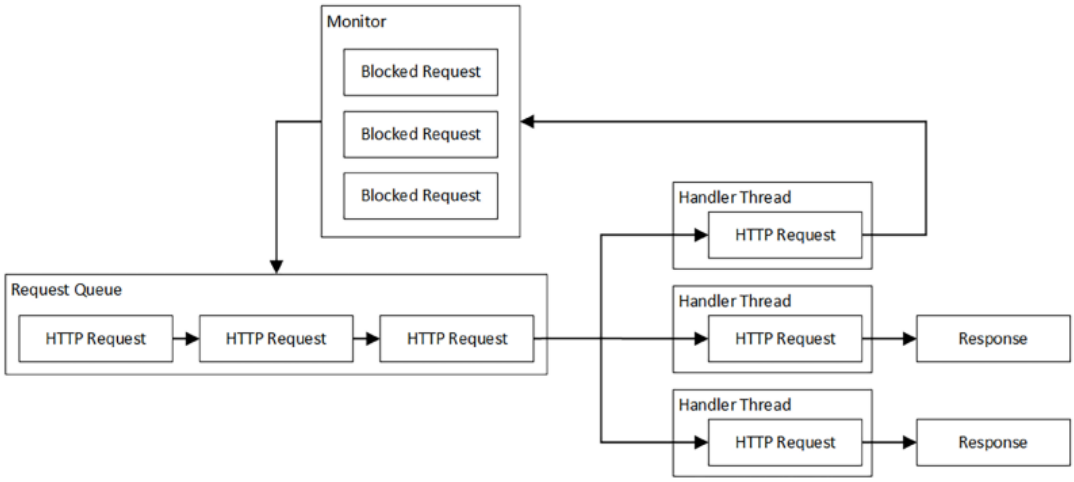

Figura 3: Liberación de los controladores de solicitudes de una operación de bloqueo.

En lugar de esperar a que se complete una operación, los subprocesos controladores dependen de un subproceso de monitorización mientras continúan procesando solicitudes de la cola. Cuando la operación de bloqueo ha finalizado, el subproceso de monitorización vuelve a poner la solicitud en la cola para que un subproceso controlador pueda seguir procesándola.

El proceso de transferencia de una operación para su supervisión suele estar integrado en la API que se utiliza para escribir aplicaciones web, de modo que, al realizar una lectura de un archivo, por ejemplo, se libera automáticamente el subproceso controlador para que pueda realizar otro trabajo y se pueda confiar en que colocará la solicitud en la cola para su procesamiento cuando se complete la operación de lectura del archivo.

Es importante entender que los términos no bloqueante y asíncrono se aplican desde la perspectiva del subproceso controlador. Las operaciones aún tardan en completarse, pero el subproceso controlador puede realizar otro trabajo durante ese período. Todavía hay subprocesos bloqueadores, pero no son los responsables de procesar las solicitudes HTTP, que son los subprocesos que más nos interesan.

Comprender la ejecución del código JavaScript

Los orígenes de JavaScript como lenguaje basado en navegadores han dado forma a la forma en que se escribe y ejecuta el código JavaScript. JavaScript se utilizó originalmente para proporcionar interacción del usuario con elementos HTML. Cada tipo de elemento define eventos que describen las diferentes formas en que el usuario puede interactuar con ese elemento. Un elemento de botón, por ejemplo, tiene eventos para cuando el usuario hace clic en el botón, mueve el apuntador sobre el botón, etc.

El programador escribe funciones de JavaScript, conocidas como devoluciones de llamada (*callbacks*), y utiliza la API del navegador para asociar esas funciones con eventos específicos en los elementos. Cuando el navegador detecta un evento, agrega la devolución de llamada a una cola para que pueda ser ejecutada por el entorno de ejecución de JavaScript.

El entorno de ejecución de JavaScript tiene un solo hilo, llamado hilo principal (*main thread*), que es responsable de ejecutar las devoluciones de llamada. El hilo principal se ejecuta en un ciclo, tomando devoluciones de llamada de la cola y ejecutándolas, lo que se conoce como ciclo de eventos de JavaScript. El ciclo de eventos es la forma en que el código nativo del navegador, que está escrito para un sistema operativo específico, interactúa con el código JavaScript, que se ejecuta en cualquier entorno de ejecución compatible.

Nota

El ciclo de eventos es más complicado, pero la idea de una cola de devoluciones de llamadas es lo suficientemente cercana para un desarrollo web JavaScript eficaz. Vale la pena explorar los detalles si, como cuando investigo más detalles, encuentras este tipo de cosas interesantes. Un buen lugar para comenzar es: https://nodejs.org/en/learn/asynchronous-work/event-loop-timers-and-nexttick.

Los eventos a menudo ocurren en grupos, como cuando el apuntador se mueve a través de varios elementos, por lo que la cola puede contener múltiples devoluciones de llamadas esperando ser ejecutadas, como se muestra en la figura 4.

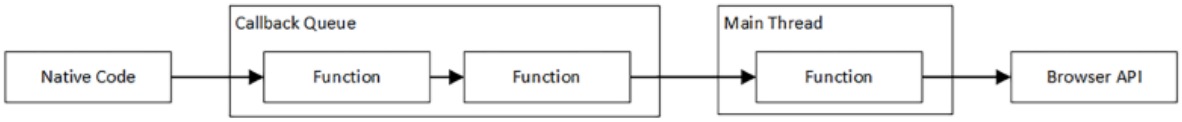

Figura 4: La cola de devoluciones de llamadas.

El uso de un solo hilo significa que cualquier operación en una devolución de llamada que tarde tiempo en completarse hace que la aplicación se congele mientras las devoluciones de llamadas se ponen en cola esperando ser procesadas. Para ayudar a solucionar este problema, muchas funciones de la API del navegador no son bloqueantes y utilizan el patrón de devolución de llamada para entregar sus resultados.

A lo largo de los años, se han agregado funciones al lenguaje JavaScript y a las API del navegador, pero el ciclo de eventos y las funciones de devolución de llamada se utilizan para ejecutar JavaScript. La API que proporciona el navegador para las solicitudes HTTP, por ejemplo, define una serie de eventos que describen el ciclo de vida de la solicitud, y estos eventos se manejan con funciones de devolución de llamada, como se muestra en la figura 5.

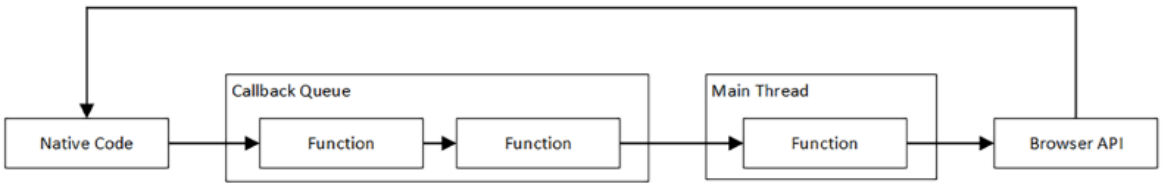

Figura 5: Los resultados de la API del navegador se procesan con funciones de devolución de llamada de JavaScript.

En segundo plano, el navegador utiliza subprocesos nativos para realizar la solicitud HTTP y esperar la respuesta, que luego se pasa al entorno de ejecución de JavaScript mediante una devolución de llamada.

El entorno de ejecución de JavaScript solo ejecuta una devolución de llamada, por lo que el lenguaje JavaScript no necesita palabras clave como lock y asynchronous. El código JavaScript interactúa con el navegador a través de una API que oculta los detalles de implementación y recibe resultados de manera consistente.

Comprensión de la ejecución del código de Node.js

Node.js conserva el subproceso principal y el ciclo de eventos, lo que significa que el código del lado del servidor se ejecuta de la misma manera que el código de JavaScript del lado del cliente. Para los servidores HTTP, el subproceso principal es el único controlador de solicitudes y las devoluciones de llamada se utilizan para manejar las conexiones HTTP entrantes. La aplicación de ejemplo demuestra el uso de una devolución de llamada para manejar una solicitud HTTP:

```
... const server = createServer(handler); ...
```

La función de devolución de llamada que se pasa a la función createServer se invocará cuando Node.js reciba una conexión HTTP. La función define parámetros que representan la solicitud que se ha recibido y la respuesta que se devolverá al cliente:

```
..
export const handler = (req: IncomingMessage, res: ServerResponse) => {
    res.end("Hello World");
};
...
```

Describimos la API que Node.js proporciona para HTTP en el siguiente documento, pero la función de devolución de llamada utiliza sus parámetros para preparar la respuesta que se enviará al cliente. Los detalles de cómo Node.js recibe solicitudes HTTP y devuelve respuestas HTTP están ocultos en el código nativo, como se muestra en la figura 6.

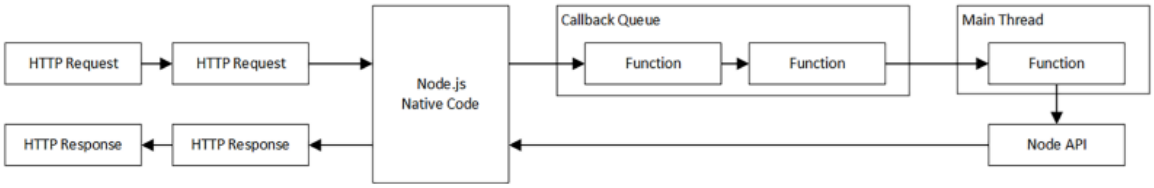

Figura 6: Manejo de solicitudes HTTP en Node.js.

Node.js puede tener solo un único hilo de controlador, pero el rendimiento puede ser excelente porque el hardware de servidor moderno es increíblemente rápido. Aun así, un solo hilo no aprovecha al máximo el hardware multinúcleo y multiprocesador en el que se implementan la mayoría de las aplicaciones.

Para escalar, se inician varias instancias de Node.js. Las solicitudes HTTP son recibidas por un balanceador de carga (o un controlador de ingreso o un nodo principal, según cómo se implemente la aplicación, como se describe más adelante) y distribuidas a las instancias de Node.js, como se muestra en la figura 7.

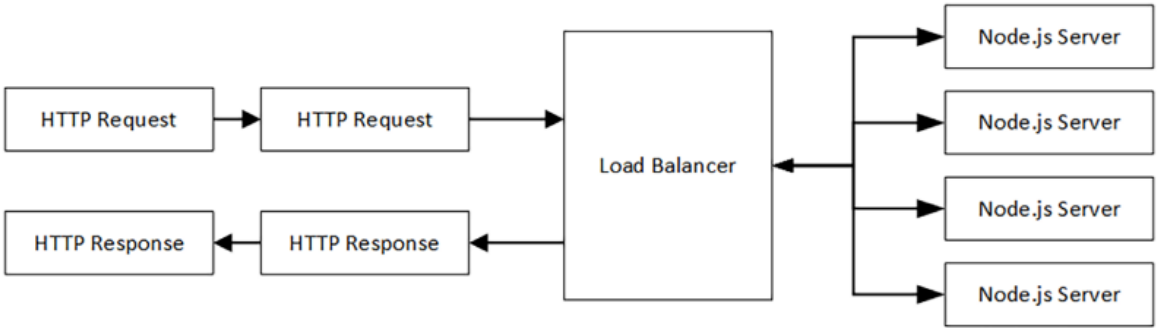

Figura 7: Escalado con múltiples instancias de Node.js.

Las instancias individuales de Node.js aún tienen un único hilo de JavaScript, pero en conjunto pueden procesar un mayor volumen de solicitudes.

Una consecuencia importante de aplicar el modelo de ejecución de JavaScript a las solicitudes HTTP es que bloquear el hilo principal impide que esa instancia de Node.js procese todas las solicitudes, lo que crea el mismo tipo de bloqueo que puede surgir en JavaScript del lado del cliente.

Node.js ayuda a los programadores a evitar bloquear el hilo principal de dos maneras: una API que realiza muchas tareas de forma asíncrona, conocida como grupo de trabajadores (*worker pool*), y soporte para iniciar hilos adicionales para ejecutar código JavaScript bloqueador, conocidos como hilos de trabajo (*worker threads*). Ambas características se describen en las secciones siguientes.

Uso de la API de Node.js

Node.js reemplaza la API proporcionada por el navegador por una que admite tareas comunes del lado del servidor, como procesar solicitudes HTTP y leer archivos. Detrás de escena, Node.js utiliza subprocesos nativos, conocidos como el grupo de trabajadores, para realizar operaciones de forma asíncrona.

Para demostrarlo, el listado 10 utiliza la API de Node.js para leer el contenido de un archivo.

Listado 10: Uso de la API de Node.js en el archivo handler.ts en la carpeta src.

```
import { IncomingMessage, ServerResponse } from "http";
import { readFile } from "fs";

export const handler = (req: IncomingMessage, res: ServerResponse) => {
  readFile("data.json", (err: Error | null, data: Buffer) => {
    if (err == null) {
      res.end(data, () => console.log("File sent"));
    } else {
      console.log(`Error: ${err.message}`);
      res.statusCode = 500;
      res.end();
    }
  });
};
```

Como sugiere su nombre, la función readFile lee el contenido de un archivo. Utiliza un navegador web para solicitar http://localhost:5000 y verás el resultado que se muestra en la figura 8.

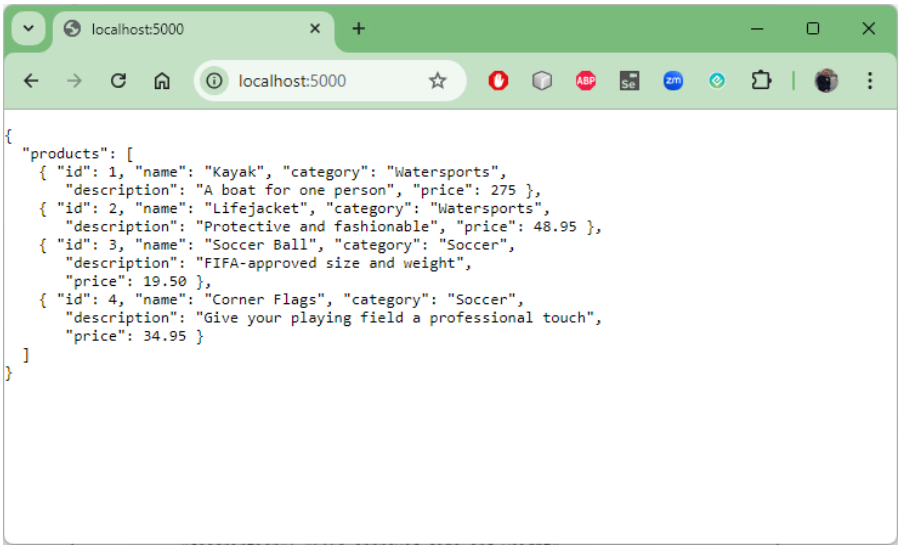

Figura 8: Envío del contenido de un archivo al cliente.

La operación de lectura es asíncrona y se implementa utilizando un subproceso nativo. El contenido del archivo se pasa a una función de devolución de llamada, que lo envía al cliente HTTP.

Hay tres devoluciones de llamada en el código. La primera devolución de llamada es la que se pasa a la función createServer, que se invoca cuando se recibe una solicitud HTTP:

```
...
const server = createServer(handler);
...
```

La segunda devolución de llamada es la que se pasa a la función readFile, que se invoca cuando se ha leído el contenido del archivo o si se produce un error:

```
...
export const handler = (req: IncomingMessage, res: ServerResponse) => {
    readFile("data.json", (err: Error | null, data: Buffer) => {
        if (err == null) {
            res.end(data, () => console.log("File sent"));
        } else {
            console.log(`Error: ${err.message}`);
            res.statusCode = 500;
            res.end();
        }
    });
};
...
```

Usamos anotaciones de tipo para ayudar a describir la forma en que se presentan los resultados de la lectura del archivo.

El tipo del primer argumento de la devolución de llamada es Error | null y se utiliza para indicar el resultado.

Si el primer argumento es null, entonces la operación se ha completado correctamente y el contenido del archivo estará disponible en el segundo argumento, cuyo tipo es Buffer. (Los buffers son la forma en que Node.js representa los arreglos de bytes).

Si el primer argumento no es null, entonces el objeto Error proporcionará detalles del problema que impidió que se leyera el archivo.

Nota

Es posible que veas dos mensajes escritos en el símbolo del sistema cuando envíes una solicitud HTTP desde un navegador. Los navegadores suelen solicitar el archivo favicon.ico para obtener un icono que se pueda mostrar en el encabezado de la pestaña, y esta es la razón por la que a veces verás que File sent aparece dos veces en la salida.

La tercera devolución de llamada se invoca cuando los datos leídos del archivo se han enviado al cliente:

```
...
export const handler = (req: IncomingMessage, res: ServerResponse) => {
    readFile("data.json", (err: Error | null, data: Buffer) => {
        if (err == null) {
            res.end(data, () => console.log("File sent"));
        } else {
            console.log(`Error: ${err.message}`);
            res.statusCode = 500;
            res.end();
        }
    });
};
...
```

Dividir el proceso de producción de una respuesta HTTP con devoluciones de llamada significa que el hilo principal de JavaScript no tiene que esperar a que el sistema de archivos lea el contenido del archivo, y esto permite que se procesen las solicitudes de otros clientes, como se ilustra en la figura 9.

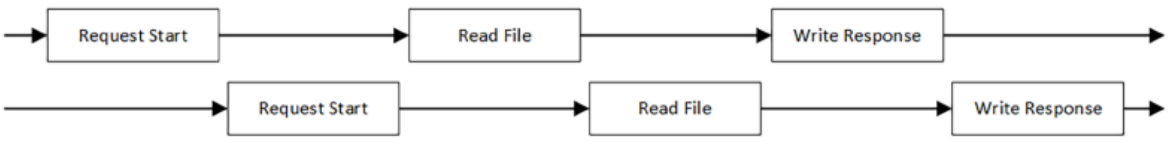

Figura 9: Desglose del manejo de solicitudes con múltiples devoluciones de llamada.

Manejo de eventos

Los eventos se utilizan para proporcionar notificaciones de que el estado de la aplicación ha cambiado y brindan la oportunidad de ejecutar una función de devolución de llamada para manejar ese cambio. Los eventos se utilizan en toda la API de Node.js, aunque a menudo hay funciones de conveniencia que ocultan los detalles. El listado 11 revisa el código que escucha las solicitudes HTTP para usar eventos directamente.

Listado 11: Manejo de eventos en el archivo server.ts en la carpeta src.

```
import { createServer } from "http";
import { handler } from "./handler";

const port = 5000;

const server = createServer();

server.on("request", handler)

server.listen(port);

server.on("listening", () => {
  console.log(`(Event) Server listening on port ${port}`);
});
```

Muchos de los objetos creados con la API de Node.js extienden la clase EventEmitter, que denota una fuente de eventos. La clase EventEmitter define los métodos descritos en la tabla 3 para recibir eventos.

Tabla 3: Métodos eventemitter útiles.

Nombre	Descripción
on(event, callback)	Este método registra una devolución de llamada que se invocará cada vez que se emita el evento especificado.
off(event, callback)	Este método deja de invocar la devolución de llamada cuando se emite el evento específico.
once(event, callback)	Este método registra una devolución de llamada que se invocará la próxima vez que se emita el evento especificado, pero no después.

Las clases que extienden EventEmitter definen eventos y especifican cuándo se emitirán. La clase Server devuelta por el método createServer extiende EventEmitter y define dos eventos que se utilizan en el listado 11: los eventos de solicitud y de escucha.

El código en el listado 7 y el listado 11 tiene el mismo efecto y la única diferencia es que la función createServer registra su argumento de función como una devolución de llamada para el evento de solicitud en segundo plano, mientras que el método listen registra su argumento de función como una devolución de llamada para el evento de escucha.

Es importante entender que los eventos son una parte integral de la API de Node.js y que pueden usarse directamente, con los métodos descritos en la tabla 3, o indirectamente a través de otras funciones.

Trabajar con promesas

Las promesas son una alternativa a las devoluciones de llamadas y algunas partes de la API de Node.js proporcionan funciones que utilizan tanto devoluciones de llamadas como promesas.

Una promesa tiene el mismo propósito que una devolución de llamada, que es definir el código que se ejecutará cuando se complete una operación asíncrona. La diferencia es que el código escrito con promesas a menudo puede ser más simple que el código equivalente que utiliza devoluciones de llamadas.

Una parte de la API donde Node.js proporciona promesas y devoluciones de llamadas es para trabajar con archivos, como se muestra en el listado 12.

Listado 12: Uso de una promesa en el archivo handler.ts en la carpeta src.

```
import { IncomingMessage, ServerResponse } from "http";
//import { readFile } from "fs";
import { readFile } from "fs/promises";

export const handler = (req: IncomingMessage, res: ServerResponse) => {
  const p: Promise<Buffer> = readFile("data.json");
  p.then((data: Buffer) => res.end(data, () => console.log("File sent")));
  p.catch((err: Error) => {
    console.log(`Error: ${err.message}`);
    res.statusCode = 500;
    res.end();
  });
};
```

No es así como se utilizan habitualmente las promesas, por lo que el código parece más complejo que los ejemplos anteriores. Pero este código enfatiza la forma en que funcionan las promesas. Esta es la declaración que crea la promesa:

```
...
const p: Promise<Buffer> = readFile("data.json");
...
```

La función readFile tiene el mismo nombre que la función utilizada para las devoluciones de llamadas, pero está definida en el módulo fs/promises. El resultado devuelto por la función readFile es Promise<Buffer>, que es una promesa que producirá un objeto Buffer cuando se complete su operación asíncrona.

Entender cuándo son útiles los métodos síncronos

Además de las devoluciones de llamadas y las promesas, algunas partes de la API de Node.js también ofrecen funciones síncronas que bloquean el hilo principal hasta que se completen. Un ejemplo es la función readFileSync, que realiza la misma tarea que readFile, pero bloquea la ejecución hasta que se haya leído el contenido del archivo.

En la mayoría de los casos, debes usar las funciones sin bloqueo que proporciona Node.js para maximizar la cantidad de solicitudes que Node.js puede manejar, pero hay dos situaciones en las que las operaciones de bloqueo tienen más sentido. La primera situación surge cuando sabes con certeza que las operaciones se completarán tan rápido que es más rápido que configurar una promesa o una devolución de llamada. Existe un costo de recursos y tiempo asociado con la realización de una operación asíncrona y esto a veces se puede evitar.
Esta situación no surge a menudo y debes considerar cuidadosamente el posible impacto en el rendimiento.

La segunda situación es más común y es cuando sabes que el siguiente bloque de código que ejecutará el hilo principal será el resultado de la operación que estás a punto de realizar. Puedes ver un ejemplo de esto más adelante, donde leeremos archivos de configuración de forma síncrona antes de que Node.js comience a escuchar solicitudes HTTP.

Las promesas se *resuelven* o se *rechazan*. Una promesa que se completa con éxito y produce su resultado se resuelve. El método then se utiliza para registrar la función que se invocará si se resuelve la promesa, lo que significa que el archivo se ha leído correctamente, de esta manera:

```
...
p.then((data: Buffer) => res.end(data, () => console.log("File sent")));
...
```

Una promesa rechazada es aquella en la que se ha producido un error. El método catch se utiliza para registrar una función que maneja el error producido por una promesa rechazada, de esta manera:

```
...
p.catch((err: Error) => {
    console.log(`Error: ${err.message}`);
    res.statusCode = 500;
    res.end();
});
...
```

Observa que el uso de una promesa no cambia los tipos de datos utilizados para describir los resultados: se utiliza un Buffer para describir los datos leídos del archivo y se utiliza un Error para describir los errores.

El uso de los métodos then y catch separa los resultados exitosos de los errores, a diferencia de la API de devolución de llamada, que presenta ambos y requiere que la función de devolución de llamada determine lo que sucedió.

Los métodos then y catch se pueden encadenar, lo que es una pequeña mejora en la simplificación del código, como se muestra en el listado 13, y es una forma más típica de utilizar promesas.

Listado 13: Encadenamiento de métodos de promesa en el archivo handler.ts en la carpeta src.

```
import { IncomingMessage, ServerResponse } from "http";
import { readFile } from "fs/promises";

export const handler = (req: IncomingMessage, res: ServerResponse) => {
  readFile("data.json")
    .then((data: Buffer) => res.end(data, () => console.log("File sent")))
    .catch((err: Error) => {
      console.log(`Error: ${err.message}`);
      res.statusCode = 500;
      res.end();
  });
};
```

Esto es un poco más ordenado, pero la mejora real viene con el uso de las palabras clave async y await, que permiten realizar operaciones asíncronas utilizando una sintaxis que no requiere funciones anidadas o métodos encadenados, como se muestra en el listado 14.

Listado 14: Uso de las palabras clave async y await en el archivo handler.ts en la carpeta src.

```
import { IncomingMessage, ServerResponse } from "http";
import { readFile } from "fs/promises";

export const handler = async (req: IncomingMessage, res: ServerResponse) => {
  const data: Buffer = await readFile("data.json");
  res.end(data, () => console.log("File sent"));
};
```

El uso de las palabras clave async y await simplifica el código al eliminar la necesidad del método then y su función. La palabra clave async se aplica a la función que se utiliza para manejar las solicitudes:

```
...
export const handler = async (req: IncomingMessage, res: ServerResponse) => {
...
```

La palabra clave await se aplica a las instrucciones que devuelven promesas, como esta:

```
...
const data: Buffer = await readFile("data.json");
...
```

Estas palabras clave no cambian el comportamiento de la función readFile, que sigue leyendo un archivo de forma asíncrona y sigue devolviendo un Promise<Buffer>, pero el entorno de ejecución de JavaScript toma el resultado producido de forma asíncrona por la promesa, un objeto Buffer en este caso, lo asigna a una constante denominada data y luego ejecuta las instrucciones que siguen. El resultado es el mismo (y la forma en que se obtiene el resultado también es la misma), pero la sintaxis es más simple y más fácil de leer.

Esta no es la versión final del código. Para admitir el manejo de errores, el método catch utilizado en los objetos Promise se reemplaza con un bloque try/catch cuando se utiliza la palabra clave await, como se muestra en el listado 15.

Listado 15: Adición del manejo de errores en el archivo handler.ts en la carpeta src.

```
import { IncomingMessage, ServerResponse } from "http";
import { readFile } from "fs/promises";

export const handler = async (req: IncomingMessage, res: ServerResponse) => {
  try {
     const data: Buffer = await readFile("data.json");
     res.end(data, () => console.log("File sent"));
  } catch (err: any) {
     console.log(`Error: ${err?.message ?? err}`);
     res.statusCode = 500;
     res.end();
  }
};
```

El tipo del valor proporcionado a la excepción catch es cualquiera, no Error, porque JavaScript no restringe los tipos que se pueden usar para representar errores.

Sugerencia

Una ventaja de las devoluciones de llamadas sobre las promesas es que las devoluciones de llamadas se pueden invocar más de una vez para la misma operación, lo que permite que se proporcione una serie de actualizaciones mientras se realiza un trabajo asíncrono. Las promesas están destinadas a producir un único resultado sin actualizaciones intermedias. Puedes ver un ejemplo de esta diferencia al final del documento.

Envolver (wrapping) devoluciones de llamadas y desenvolver (unwrapping) promesas

No todas las partes de la API de Node.js admiten promesas y devoluciones de llamadas, y eso puede provocar que ambos enfoques se mezclen en el mismo código. Puedes ver este problema en el ejemplo, donde la función readFile devuelve una promesa, pero el método end, que envía datos al cliente y finaliza la respuesta HTTP, utiliza una devolución de llamada:

```
...
const data: Buffer = await readFile("data.json");
res.end(data, () => console.log("File sent"));
...
```

Las API de promesa y devolución de llamada se pueden mezclar sin problemas, pero el resultado puede ser un código extraño. Para ayudar a garantizar la coherencia, la API de Node.js incluye dos funciones útiles en el módulo util, que se describen en la tabla 4.

Tabla 4: Las funciones para envolver devoluciones de llamadas y desenvolver promesas.

Nombre	Descripción
promisify	Esta función crea una Promise a partir de una función que acepta una devolución de llamada convencional. La convención es que los argumentos que se pasan a la devolución de llamada son un objeto de error y el resultado de la operación. Hay soporte para otras disposiciones de argumentos que utilizan un símbolo personalizado; consulta: https://nodejs.org/docs/latest/api/util.html#utilpromisifycustom para obtener más detalles.
callbackify	Esta función acepta un objeto Promise y devuelve una función que aceptará una devolución de llamada convencional.

La idea detrás de estas funciones es buena, pero tienen limitaciones, especialmente cuando se intenta crear promesas a partir de devoluciones de llamadas para que se pueda utilizar la palabra clave await. La mayor restricción es que la función promisify no funciona sin problemas en los métodos de clase a menos que se tenga cuidado de lidiar con la forma en

que JavaScript maneja la palabra clave this. También hay un problema específico de TypeScript, donde el compilador no identifica correctamente los tipos involucrados.

Agrega un archivo llamado promises.ts a la carpeta src con el contenido que se muestra en el listado 16.

Listado 16: El contenido del archivo promises.ts en la carpeta src.

```
import { ServerResponse } from "http";
import { promisify } from "util";

export const endPromise = promisify(ServerResponse.prototype.end) as
 ( data: any) => Promise<void>;
```

El primer paso es usar promisify para crear una función que devuelva una promesa, lo que hacemos al pasar la función ServerResponse.prototype.end a promisify. Usamos la palabra clave *as* para anular el tipo inferido por el compilador de TypeScript con una descripción de los parámetros del método y el resultado:

```
…
export const endPromise = promisify(ServerResponse.prototype.end) as
  ( data: any) => Promise<void>;
…
```

El listado 17 importa la función definida en el listado 16 y usa la promesa que produce.

Listado 17: Uso de una promesa en el archivo handler.ts en la carpeta src.

```
import { IncomingMessage, ServerResponse } from "http";
import { readFile } from "fs/promises";
import { endPromise } from "./promises";

export const handler = async (req: IncomingMessage, res: ServerResponse) => {
  try {
    const data: Buffer = await readFile("data.json");
    await endPromise.bind(res)(data);
    console.log("File sent");
  } catch (err: any) {
    console.log(`Error: ${err?.message ?? err}`);
    res.statusCode = 500;
    res.end();
  }
};
```

Tenemos que usar el método bind cuando usamos la palabra clave await en la función que promisify crea, de esta manera:

```
...
await endPromise.bind(res)(data);
...
```

El método bind asocia el objeto ServerResponse para el cual se invoca la función.
El resultado es una nueva función, que se invoca al pasar los datos que se enviarán al cliente:

```
...
await endPromise.bind(res)(data);
...
```

El resultado es que se puede usar la palabra clave await en lugar de la devolución de llamada, aunque es un proceso un poco complicado.

Ejecución de código personalizado

Todo el código JavaScript es ejecutado por el hilo principal, lo que significa que cualquier operación que no use la API sin bloqueo proporcionada por Node.js bloqueará el hilo. Por el bien de la coherencia, agrega la declaración que se muestra en el listado 18 al archivo promises.ts para envolver el método de escritura definido por la clase ServerResponse en una promesa.

Listado 18: Agregar una función en el archivo promises.ts en la carpeta src.

```
import { ServerResponse } from "http";
import { promisify } from "util";

export const endPromise = promisify(ServerResponse.prototype.end) as
  ( data: any) => Promise<void>;

 export const writePromise = promisify(ServerResponse.prototype.write) as
  (data: any) => Promise<void>;
```

El listado 19 filtra las solicitudes del archivo favicon.ico, lo cual estaba bien en ejemplos anteriores, pero agregará solicitudes no deseadas en esta sección.

Listado 19: Filtrado de solicitudes en el archivo server.ts en la carpeta src.

```
import { createServer } from "http";
import { handler } from "./handler";
```

```
const port = 5000;

const server = createServer();

server.on("request", (req, res) => {
  if (req.url?.endsWith("favicon.ico")) {
    res.statusCode = 404;
    res.end();
  } else {
   handler(req, res)
  }
});

server.listen(port);

server.on("listening", () => {
  console.log(`(Event) Server listening on port ${port}`);
});
```

El listado 20 demuestra el problema del bloqueo de subprocesos al introducir una operación que consume mucho tiempo y que se implementa completamente en JavaScript.

Listado 20: Una operación de bloqueo en el archivo handler.ts en la carpeta src.

```
import { IncomingMessage, ServerResponse } from "http";
//import { readFile } from "fs/promises";
import { endPromise, writePromise } from "./promises";

const total = 2_000_000_000;
const iterations = 5;
let shared_counter = 0;

export const handler = async (req: IncomingMessage, res: ServerResponse) => {
  const request = shared_counter++;
  for (let iter = 0; iter < iterations; iter++) {
    for (let count = 0; count < total; count++) {
      count++;
    }
    const msg = `Request: ${request}, Iteration: ${(iter)}`;
    console.log(msg);
    await writePromise.bind(res)(msg + "\n");
  }
  await endPromise.bind(res)("Done");
};
```

Dos ciclos for incrementan repetidamente un valor numérico y, dado que esta operación está escrita completamente en JavaScript, el subproceso principal se bloquea hasta que ambos ciclos se hayan completado. Para ver el efecto del subproceso bloqueado, abre dos pestañas del navegador y solicita http://localhost:5000 en ambas.

Debes iniciar la solicitud en la segunda pestaña antes de que finalice la primera, y es posible que debas ajustar el valor total para tener tiempo. El valor total del listado 20 tarda tres o cuatro segundos en completarse en mi sistema, lo que es tiempo suficiente para iniciar solicitudes en ambas pestañas del navegador.

Cómo evitar el problema de la memoria caché del navegador

Algunos navegadores, incluido Chrome, no realizan solicitudes simultáneas para la misma URL. Esto significa que la solicitud de la segunda pestaña del navegador no se iniciará hasta que se haya recibido la respuesta de la solicitud de la primera pestaña, lo que puede hacer que parezca que las solicitudes siempre están bloqueadas.

Los navegadores hacen esto para ver si el resultado de la primera solicitud se puede agregar a su caché y usar para solicitudes posteriores. Esto no suele ser un problema, pero puede ser confuso, especialmente para funciones como las que se analizan en este documento.

Puedes evitar este problema deshabilitando la memoria caché del navegador (Chrome tiene una casilla de verificación Deshabilitar caché en la pestaña Red en la ventana de herramientas para desarrolladores F12, por ejemplo) o solicitando URL diferentes, como http://localhost:5000?id=1 y http://localhost:5000?id=2.

Verás que ambas pestañas del navegador obtienen resultados, como se muestra en la figura 10. Cada solicitud se identifica incrementando el valor shared_counter, lo que facilita la correlación de la salida que se muestra en el navegador con los mensajes de la consola de Node.js.

Examina la salida de la consola de Node.js y verás que todas las iteraciones de la primera solicitud se completaron antes de que se iniciara el trabajo de la segunda solicitud:

```
(Event) Server listening on port 5000
Request: 0, Iteration: 0
Request: 0, Iteration: 1
Request: 0, Iteration: 2
Request: 0, Iteration: 3
Request: 0, Iteration: 4
Request: 1, Iteration: 0
Request: 1, Iteration: 1
Request: 1, Iteration: 2
Request: 1, Iteration: 3
Request: 1, Iteration: 4
```

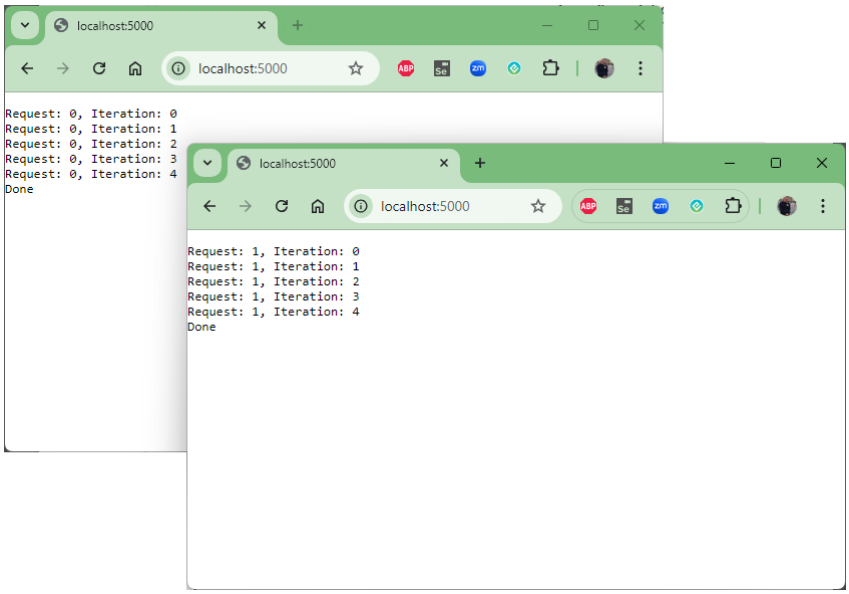

Figura 4.10: Bloqueo del hilo principal.

Este es un ejemplo típico, aunque exagerado, de bloqueo del hilo de JavaScript, de modo que las solicitudes se ponen en cola esperando su turno para ser procesadas y el rendimiento general de las solicitudes disminuye.

Cesión del control del hilo principal

Una forma de abordar el bloqueo es dividir el trabajo en fragmentos más pequeños que se intercalan con otras solicitudes. El trabajo aún se realiza en su totalidad con el hilo principal, pero el bloqueo se produce en una serie de períodos más cortos, lo que significa que el acceso al hilo principal es más equitativo.

La tabla 5 describe las funciones que están disponibles para indicarle a Node.js que invoque una función en el futuro. (Como antes, estamos simplificando las cosas aquí para evitar entrar en detalles de bajo nivel del ciclo de eventos de Node.js).

Tabla 5: Las funciones de scheduling.

Nombre	Descripción
setImmediate	Esta función le indica a Node.js que agregue una función a la cola de devolución de llamadas.
setTimeout	Esta función le indica a Node.js que agregue una función a la cola de devolución de llamadas que no debe invocarse durante al menos una cantidad específica de milisegundos.

Estas son funciones globales, lo que significa que se pueden usar sin una importación de módulo.

El listado 21 usa la función setImmediate para que la operación de conteo se divida en bloques de trabajo más pequeños.

Listado 21: Uso de la función setImmediate en el archivo handler.ts en la carpeta src.

```
import { IncomingMessage, ServerResponse } from "http";
import { endPromise, writePromise } from "./promises";

const total = 2_000_000_000;
const iterations = 5;
let shared_counter = 0;

export const handler = async (req: IncomingMessage, res: ServerResponse) => {
  const request = shared_counter++;

  const iterate = async (iter: number = 0) => {
    for (let count = 0; count < total; count++) {
      count++;
    }

    const msg = `Request: ${request}, Iteration: ${(iter)}`;
    console.log(msg);
    await writePromise.bind(res)(msg + "\n");
    if (iter == iterations -1) {
       await endPromise.bind(res)("Done");
    } else {
       setImmediate(() => iterate(++iter));
    }
  }
  iterate();
};
```

La función iterate realiza un bloque de conteo y luego usa la función setImmediate para diferir el siguiente bloque. Usa dos pestañas del navegador para solicitar http://localhost:5000 (o http://localhost:5000?id=1 y http://localhost:5000?id=2 si no has deshabilitado la memoria caché del navegador) y verás que los mensajes de consola generados por Node.js muestran que el trabajo realizado para las dos solicitudes se ha intercalado:

Puede ver una secuencia diferente de iteraciones, pero el punto importante es que el trabajo para las solicitudes HTTP se divide y se intercala.

```
...
Request: 0, Iteration: 0
Request: 0, Iteration: 1
Request: 1, Iteration: 0
Request: 0, Iteration: 2
Request: 1, Iteration: 1
Request: 0, Iteration: 3
Request: 1, Iteration: 2
Request: 0, Iteration: 4
Request: 1, Iteration: 3
Request: 1, Iteration: 4
...
```

Cómo evitar la trampa de las promesas de JavaScript puro

Un error común es intentar envolver código JavaScript bloqueado en una promesa, como esta:

```
...
await new Promise<void>(resolve => {
    // executor - perform one unit of blocking work
    resolve();
}).then(() => {
    // follow on - set up next unit of work
});
...
```

Este enfoque presenta dos trampas para el desarrollador incauto. La primera es que el executor, que es la función que realiza el trabajo, se realiza de forma síncrona. Esto puede parecer extraño, pero recuerda que todo el código JavaScript se ejecuta de forma síncrona y la expectativa es que el executor se utilizará para invocar métodos de API asíncronos que producirán resultados en el futuro y se agregarán a la cola de devolución de llamadas para su procesamiento final.

La segunda trampa es que la función de seguimiento (*follow-on*), que se pasa al método then, se ejecuta tan pronto como el executor finaliza, antes de que el hilo principal regrese a la cola de devolución de llamadas para obtener otra función para ejecutar, con el efecto de que no hay intercalación de trabajo.
Las promesas son una forma útil de consumir una API que utiliza hilos nativos para realizar trabajo asíncrono, pero no ayudan cuando se ejecuta código JavaScript puro.

Uso de subprocesos de trabajo

La principal limitación del ejemplo anterior es que sigue habiendo un solo subproceso principal, y todavía tiene que hacer todo el trabajo, independientemente de lo equitativa que sea la realización del trabajo.

Node.js admite subprocesos de trabajo (*worker threads*), que son subprocesos adicionales para ejecutar código JavaScript, aunque con restricciones. JavaScript no tiene las características para coordinar subprocesos que se encuentran en otros lenguajes, como C# o Java, e intentar agregarlos sería difícil.

En cambio, los subprocesos de trabajo se ejecutan en instancias separadas del motor Node.js, ejecutando código de forma aislada del subproceso principal. La comunicación entre el subproceso principal y los subprocesos de trabajo se realiza mediante eventos, como se muestra en la figura 11, que se adapta perfectamente al ciclo de eventos de JavaScript, de modo que los resultados producidos por los subprocesos de trabajo se procesan mediante funciones de devolución de llamada, al igual que cualquier otro código JavaScript.

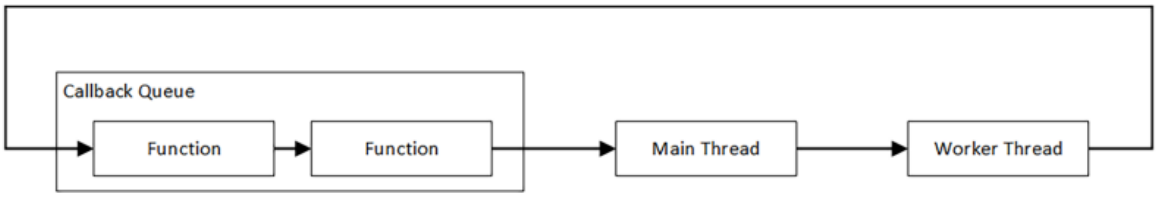

Figura 11: El subproceso principal y los subprocesos de trabajo.

Los subprocesos de trabajo no son la solución a todos los problemas porque crearlos y administrarlos implica una sobrecarga, pero proporcionan una forma efectiva de ejecutar código JavaScript sin bloquear el subproceso principal.

Comprender los subprocesos de trabajo frente al grupo de trabajadores

Existe una superposición de terminología que puede causar confusión porque Node.js utiliza dos términos similares: subprocesos de trabajo (*worker threads*) y el grupo de trabajadores (*worker pool*). Los subprocesos de trabajo son el tema de esta parte del documento y son iniciados por el programador para ejecutar código JavaScript sin bloquear el subproceso principal. El grupo de trabajadores es el conjunto de subprocesos que Node.js utiliza para implementar las características asíncronas de su API, como las funciones utilizadas en este documento para leer archivos y escribir respuestas HTTP. No interactúas directamente con el grupo de trabajadores, que Node.js administra automáticamente.

Para aumentar la confusión, los subprocesos de trabajo a menudo se agrupan en un grupo por razones de rendimiento, lo que permite reutilizar subprocesos de trabajo individuales en lugar de usarlos una vez y luego descartarlos. Explicaremos esto más adelante cómo se hace esto.

Escritura del código de trabajo

El código que ejecutan los subprocesos de trabajo se define por separado del resto de la aplicación JavaScript. Agrega un archivo llamado count_worker.ts a la carpeta src con el contenido que se muestra en el listado 22.

Listado 22: El contenido del archivo count_worker.ts en la carpeta src.

```
import { workerData, parentPort } from "worker_threads";

console.log(`Worker thread ${workerData.request} started`);

for (let iter = 0; iter < workerData.iterations; iter++) {
  for (let count = 0; count < workerData.total; count++) {
    count++;
  }
  parentPort?.postMessage(iter);
}

console.log(`Worker thread ${workerData.request} finished`);
```

Las características de los subprocesos de trabajo se definen en el módulo worker_theads, y dos de esas características se utilizan en el listado 22. La primera, workerData, es un objeto o valor que se utiliza para pasar datos de configuración del subproceso principal al trabajador. En este caso, el trabajador recibe tres valores a través de workerData, que especifican el ID de solicitud, la cantidad de iteraciones y el valor objetivo para cada bloque de trabajo de conteo:

```
…
console.log(`Worker thread ${workerData.request} started`);

for (let iter = 0; iter < workerData.iterations; iter++) {
  for (let count = 0; count < workerData.total; count++) {
…
```

La otra característica es parentPort, que se utiliza para emitir eventos que serán recibidos por el subproceso principal, como este:

```
…
parentPort?.postMessage(iter);
…
```

El método postMessage emite un evento de mensaje y se encarga de transferir el valor del argumento desde el entorno de ejecución de JavaScript del subproceso de trabajo al

subproceso principal. El valor parentPort puede ser nulo, por lo que se requiere el operador ? al llamar al método postMessage.

Creación de un subproceso de trabajo

El siguiente paso es actualizar el código de manejo de solicitudes para que cree un subproceso de trabajo utilizando el código definido en la sección anterior, como se muestra en el listado 23.

Listado 23: Uso de un subproceso de trabajo en el archivo handler.ts en la carpeta src.

```
import { IncomingMessage, ServerResponse } from "http";
import { endPromise, writePromise } from "./promises";
import { Worker } from "worker_threads";

const total = 2_000_000_000;
const iterations = 5;
let shared_counter = 0;

export const handler = async (req: IncomingMessage, res: ServerResponse) => {
  const request = shared_counter++;

  const worker = new Worker(__dirname + "/count_worker.js", {
    workerData: {
      iterations,
      total,
      request
    }
  });

  worker.on("message", async (iter: number) => {
    const msg = `Request: ${request}, Iteration: ${(iter)}`;
    console.log(msg);
    await writePromise.bind(res)(msg + "\n");
  });

  worker.on("exit", async (code: number) => {
    if (code == 0) {
       await endPromise.bind(res)("Done");
    } else {
       res.statusCode = 500;
       await res.end();
    }
  });
```

```
  worker.on("error", async (err) => {
    console.log(err)
    res.statusCode = 500;
    await res.end();
  });
};
```

Los subprocesos de trabajo se crean mediante la instanciación de la clase Worker, que se define en el módulo worker_threads. Los argumentos del constructor son el archivo de código JavaScript que se ejecutará y un objeto de configuración:

```
...
const worker = new Worker(__dirname + "/count_worker.js", {
    workerData: {
        iterations,
        total,
        request
    }
});
...
```

Node.js proporciona dos valores globales que brindan información de ruta sobre el módulo actual y son útiles para especificar rutas de archivos, que se describen en la tabla 6 para una referencia rápida. Para especificar el archivo de código creado en el listado 22, combinamos el valor __dirname con el nombre del archivo JavaScript compilado (no el archivo TypeScript, que Node.js no puede ejecutar directamente).

Tabla 6: Los valores globales para el módulo actual.

Nombre	Descripción
__filename	Este valor contiene el nombre del archivo del módulo actual. Recuerda que este será el nombre del archivo JavaScript y no el archivo TypeScript.
__dirname	Este valor contiene el nombre del directorio que contiene el módulo actual. Recuerda que este será el directorio que contiene el archivo JavaScript compilado y no el archivo TypeScript.

El objeto de configuración que se pasa al constructor Worker admite ajustes de configuración para administrar la forma en que se ejecuta un subproceso de trabajo, pero la única opción requerida para este ejemplo es workerData, que permite definir los valores de datos utilizados por el subproceso de trabajo.

Los subprocesos de trabajo se comunican con el subproceso principal mediante la emisión de eventos, que son manejados por funciones registradas por el método *on*, de la siguiente manera:

```
...
worker.on("message", async (iter: number) => {
    const msg = `Request: ${request}, Iteration: ${(iter)}`;
    console.log(msg);
    await writePromise.bind(res)(msg + "\n");
});
...
```

Sugerencia

Consulta https://nodejs.org/docs/latest/api/worker_threads.html#newworkerfilename-options para conocer las otras opciones de configuración del subproceso de trabajo, aunque las otras rara vez se requieren.

El primer argumento del método on es una cadena que especifica el nombre del evento que será manejado. Este manejador es para el evento de mensaje, que se emite cuando el subproceso de trabajo usa el método parentPort.postMessage. En este ejemplo, el evento de mensaje indica que el subproceso de trabajo ha completado una de sus iteraciones de conteo. Hay otros dos eventos manejados en este ejemplo. El evento de salida es activado por Node.js cuando el subproceso de trabajo termina, y el evento proporciona un código de salida que indica si el subproceso de trabajo terminó normalmente o fue finalizado con un error.

También hay un evento de error, que se envía si el código JavaScript ejecutado por el subproceso de trabajo arroja una excepción no detectada. Utiliza dos pestañas del navegador para solicitar http://localhost:5000 (o http://localhost:5000?id=1 y http://localhost:5000?id=2 si no has deshabilitado la memoria caché del navegador) y verás mensajes de la consola de Node.js que muestran los cálculos realizados para las solicitudes superpuestas, como este:

```
...
Worker thread 0 started
Request: 0, Iteration: 0
Request: 0, Iteration: 1
Worker thread 1 started
Request: 0, Iteration: 2
Request: 1, Iteration: 0
Request: 0, Iteration: 3
Request: 1, Iteration: 1
Request: 0, Iteration: 4
Worker thread 0 finished
Request: 1, Iteration: 2
Request: 1, Iteration: 3
Request: 1, Iteration: 4
Worker thread 1 finished
...
```

La diferencia importante con los ejemplos anteriores es que el trabajo para las solicitudes se realiza en paralelo, en lugar de que todo el trabajo se realice en un solo hilo.

Empaquetado de hilos de trabajo en una devolución de llamada

El código del listado 23 se puede encapsular para que sea coherente con la API de Node.js, mediante una devolución de llamada. Para la devolución de llamada, agrega un archivo llamado counter_cb.ts a la carpeta src con el contenido que se muestra en el listado 24.

Listado 24: El contenido del archivo counter_cb.ts en la carpeta src.

```
import { Worker } from "worker_threads";

export const Count = (request: number, iterations: number, total: number,
    callback: (err: Error | null, update: number | boolean) => void) => {
  const worker = new Worker(__dirname + "/count_worker.js", {
    workerData: {
      iterations,
      total,
      request
    }
  });

  worker.on("message", async (iter: number) => {
    callback(null, iter);
  });

  worker.on("exit", async (code: number) => {
    callback(code === 0 ? null : new Error(), true);
  });

  worker.on("error", async (err) => {
    callback(err, true);
  });
}
```

La función Count acepta argumentos que describen el trabajo a realizar y una función de devolución de llamada que se invocará cuando haya un error, cuando se complete una iteración y cuando se haya realizado todo el trabajo. El listado 25 actualiza el código de manejo de solicitudes para utilizar la función Count.

Listado 25: Uso de una función de devolución de llamada en el archivo handler.ts en la carpeta src.

```
import { IncomingMessage, ServerResponse } from "http";
```

```
import { endPromise, writePromise } from "./promises";
//import { Worker } from "worker_threads";
import { Count } from "./counter_cb";

const total = 2_000_000_000;
const iterations = 5;
let shared_counter = 0;

export const handler = async (req: IncomingMessage, res: ServerResponse) => {
  const request = shared_counter++;

  Count(request, iterations, total, async (err, update) => {
    if (err !== null) {
      console.log(err)
      res.statusCode = 500;
      await res.end();
    } else if (update !== true) {
      const msg = `Request: ${request}, Iteration: ${(update)}`;
      console.log(msg);
      await writePromise.bind(res)(msg + "\n");
    } else {
      await endPromise.bind(res)("Done");
    }
  });
};
```

Este ejemplo produce los mismos resultados que el ejemplo anterior, pero es más coherente con la mayoría de la API de Node.js, cuyas partes clave se describen en los siguientes documentos.

Empaquetado de subprocesos de trabajo en una promesa

Los subprocesos de trabajo también se pueden envolver en una promesa, aunque las promesas no son adecuadas para recibir actualizaciones provisionales como lo son las devoluciones de llamada, por lo que el uso de una promesa solo producirá un resultado cuando se haya completado todo el trabajo o cuando haya un problema. Agrega un archivo llamado count_promise.ts a la carpeta src con el contenido que se muestra en el listado 26.

Nota

Es posible producir actualizaciones provisionales con promesas, pero requiere generar una serie de promesas que deben usarse con la palabra clave await en un ciclo. El resultado es un código desordenado que no se comporta de la manera en que las promesas suelen

funcionar y es mejor evitarlo. Usa una devolución de llamada si necesita actualizaciones provisionales de un hilo de trabajo.

Listado 26: El contenido del archivo count_promise.ts en la carpeta src.

```
import { Worker } from "worker_threads";

export const Count = (request: number,
    iterations: number, total: number) : Promise<void> => {

  return new Promise<void>((resolve, reject) => {
    const worker = new Worker(__dirname + "/count_worker.js", {
      workerData: {
        iterations, total, request
      }
    });

    worker.on("message", (iter) => {
      const msg = `Request: ${request}, Iteration: ${(iter)}`;
      console.log(msg);
    });

       worker.on("exit", (code) => {
      if (code !== 0) {
        reject();
      } else {
        resolve();
      }
    });

    worker.on("error", reject);
  });
}
```

La función Count devuelve un Promise<void> cuyo executor inicia un hilo de trabajo y configura controladores para los eventos que emite. Las funciones que manejan los eventos de salida y error resuelven o rechazan la promesa, lo que indicará que la promesa está completa o lanzará una excepción.

La función de controlador para el evento de mensaje escribe mensajes de consola para mostrar el progreso, pero no afecta el resultado de la promesa. El listado 27 revisa el controlador de solicitudes para usar la versión basada en promesas de la función Count.

Listado 27: Uso de una promesa en el archivo handler.ts en la carpeta src.

```
import { IncomingMessage, ServerResponse } from "http";
import { endPromise, writePromise } from "./promises";
//import { Count } from "./counter_cb";
import { Count } from "./count_promise";

const total = 2_000_000_000;
const iterations = 5;
let shared_counter = 0;

export const handler = async (req: IncomingMessage, res: ServerResponse) => {
  const request = shared_counter++;

  try {
    await Count(request, iterations, total);
    const msg = `Request: ${request}, Iterations: ${(iterations)}`;
    await writePromise.bind(res)(msg + "\n");
    await endPromise.bind(res)("Done");
  } catch (err: any) {
    console.log(err);
    res.statusCode = 500;
    res.end();
  }
};
```

Esto es similar a los ejemplos anteriores, excepto que la respuesta enviada al cliente no incluye ningún mensaje generado al final de cada bloque de trabajo, como se muestra en la figura 12.

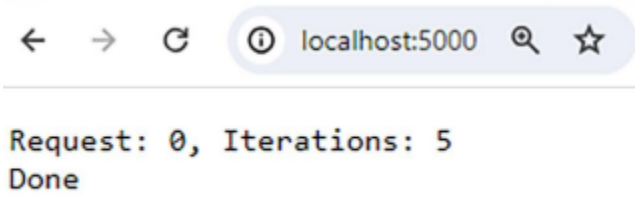

Figura 12: El resultado del hilo de trabajo envuelto en promesa.

Resumen

En este documento, describimos la forma en que se ejecuta el código JavaScript y explicamos el efecto que esto tiene en el procesamiento de solicitudes HTTP y por qué este enfoque es diferente de otras plataformas. Explicamos que el código JavaScript se ejecuta en un solo

hilo principal y demostramos las características que proporciona Node.js para descargar el trabajo en otros hilos.

- El código JavaScript se ejecuta en un único hilo, conocido como hilo principal.
- La API de Node.js utiliza hilos nativos para realizar muchas operaciones y evitar bloquear el hilo principal.
- La API de Node.js utiliza principalmente devoluciones de llamadas, pero también admite algunas promesas.
- Node.js proporciona funciones para convertir devoluciones de llamadas y promesas.
- Node.js admite hilos de trabajo para ejecutar código JavaScript sin bloquear el hilo principal.

En el próximo documento, describiremos las características que ofrece Node.js para trabajar con solicitudes HTTP.