

Utilizando flujos en Node.js

Una de las principales tareas requeridas en el desarrollo del lado del servidor es transferir datos, ya sea leyendo datos enviados por un cliente o navegador o escribiendo datos que deben transmitirse o almacenarse de alguna manera. En este documento, presentaremos la API de Node.js para tratar con fuentes y destinos de datos, conocidos como streams (flujos). Explicaremos el concepto detrás de los streams, mostraremos cómo se utilizan para tratar con solicitudes HTTP y explicaremos por qué una fuente común de datos, el sistema de archivos debe usarse con precaución en un proyecto del lado del servidor. La tabla 1 pone los streams en contexto.

Tabla 1: Poniendo los streams en contexto.

Pregunta	Respuesta
¿Qué son?	Node.js usa streams para representar fuentes o destinos de datos, incluidas las solicitudes y respuestas HTTP.
¿Por qué son útiles?	Los streams no exponen los detalles de cómo se producen o consumen los datos, lo que permite que el mismo código procese datos de cualquier fuente.
¿Cómo se usan?	Node.js proporciona streams para tratar con solicitudes HTTP. La API de streams se utiliza para leer datos de la solicitud HTTP y escribir datos en la respuesta HTTP.
¿Existen limitaciones o inconvenientes?	La API de streams puede resultar un poco complicada de utilizar, pero esto se puede mejorar con el uso de paquetes de terceros, que suelen proporcionar métodos más convenientes para realizar tareas comunes.
¿Existen alternativas?	Los streams son parte integral del desarrollo de Node.js. Los paquetes de terceros pueden simplificar el trabajo con streams, pero es útil comprender cómo funcionan los streams cuando surgen problemas.

La tabla 2 resume lo que se tratará en el documento.

Tabla 2: Resumen del documento.

Problema	Solución	Listado
Escribir datos en un flujo	Utilizar los métodos write o end.	4
Establecer encabezados de respuesta	Utilizar el método setHeader.	5-7
Gestionar el almacenamiento en búfer de datos	Utilizar el resultado del método write y gestionar el evento de vaciado.	8-9
Leer datos de un flujo	Gestionar los datos y los eventos end o utilizar un iterador.	10-15
Conectar flujos	Utilizar el método pipe.	16

Transformar datos	Extender la clase Transform y utilizar el modo de objeto de flujo.	17-19
Servir archivos estáticos	Utilizar el middleware estático Express o utilizar los métodos sendFile y download.	20-26
Codificar y decodificar datos	Utilizar el middleware JSON Express y el método de respuesta json.	27-28

Preparación para este documento

En este nuevo documento, seguiremos utilizando el proyecto de aplicación web creado en los dos documentos previos. Para preparar este documento, reemplaza el contenido del archivo `server.ts` en la carpeta `src` con el código que se muestra en el listado 1.

Listado 1: Reemplazo del contenido del archivo `server.Ts` en la carpeta `src`.

```
import { createServer } from "http";
import express, { Express } from "express";
import { basicHandler } from "../handler";

const port = 5000;

const expressApp: Express = express();

expressApp.get("/favicon.ico", (req, resp) => {
  resp.statusCode = 404;
  resp.end();
});
expressApp.get("/*", basicHandler);

const server = createServer(expressApp);

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

El enrutador Express filtra las solicitudes de favicon y pasa todas las demás solicitudes HTTP GET a una función llamada `basicHandler`, que se importa desde el módulo del controlador. Para definir el controlador, reemplaza el contenido del archivo `handler.ts` en la carpeta `src` con el código que se muestra en el listado 2.

Listado 2. El contenido del archivo `handler.ts` en la carpeta `src`.

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

export const basicHandler = (req: IncomingMessage, resp: ServerResponse) => {
```

```
    resp.end("Hello, World");  
};
```

Este controlador utiliza los tipos `IncomingMessage` y `ServerResponse` de Node.js, aunque se utiliza Express para enrutar las solicitudes. Demostraremos las mejoras que ofrece Express en la sección uso de mejoras de terceros, pero comenzamos con las funciones integradas que ofrece Node.js.

Algunos ejemplos de este documento requieren un archivo de imagen. Crea la carpeta `static` y agrégale un archivo de imagen llamado `city.png`. Puedes utilizar cualquier archivo de imagen PNG siempre que lo llames `city.png`, o puedes descargar el panorama de dominio público del horizonte de la ciudad de Nueva York que utilizamos, que se muestra en la figura 1, y se encuentra adjunto al documento de la práctica.



Figura 1: El archivo `city.png` en la carpeta `static`.

Ejecuta el comando que se muestra en el listado 3 en la carpeta `webapp` para iniciar el observador que compila los archivos TypeScript y ejecuta el JavaScript que se produce.

Listado 3: Inicio del proyecto.

```
npm start
```

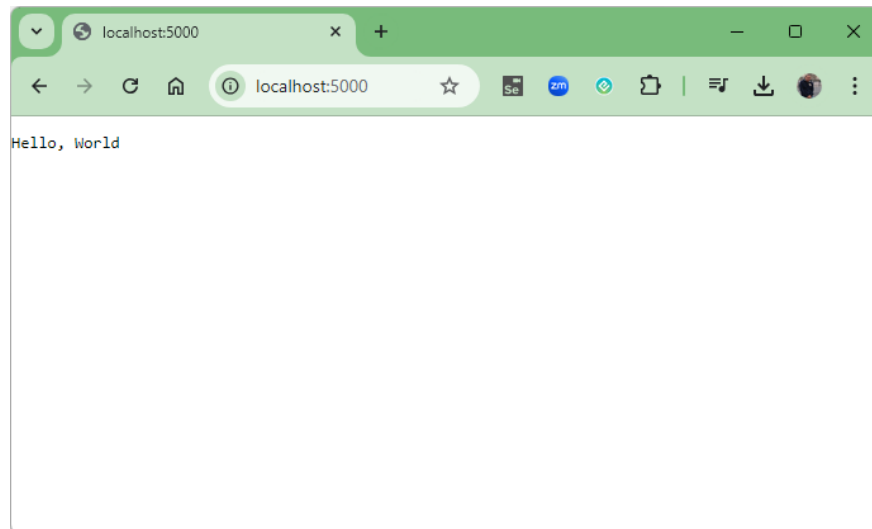


Figura 2: Ejecución del proyecto de ejemplo.

Entendiendo los streams

La mejor manera de entender los streams es ignorar los datos y pensar en el agua por un momento. Imagina que estás en una habitación en la que una tubería con un grifo entra por una pared. Tu trabajo es construir un dispositivo que recoja el agua de la tubería. Obviamente, hay algo conectado al otro extremo de la tubería que produce el agua, pero solo puedes ver el grifo, por lo que el diseño de tu dispositivo estará determinado por lo que sabes: tienes que crear algo que se conecte a la tubería y reciba el agua cuando se abra el grifo. Tener una vista tan limitada del sistema con el que estás trabajando puede parecer una restricción, pero la tubería se puede conectar a cualquier fuente de agua y tu dispositivo funciona igual de bien ya sea que el agua provenga de un río o de un embalse; todo es agua que pasa por la tubería a través del grifo y siempre se consume de forma constante.

En el otro extremo de la tubería, el productor de agua tiene una tubería en la que bombea el agua. El productor de agua no puede ver lo que has conectado al otro extremo de la tubería y no sabe cómo vas a consumir el agua. Y no importa, porque todo lo que el productor tiene que hacer es empujar el agua a través de la tubería, independientemente de si el agua se utilizará para hacer funcionar un molino de agua, llenar una piscina o hacer funcionar una ducha. Puedes cambiar el dispositivo conectado a tu tubería y nada cambiaría para el productor, que sigue bombeando agua en la misma tubería de la misma manera.

En el mundo del desarrollo web, **un flujo resuelve el problema de la distribución de datos de la misma manera que la tubería resuelve el problema de la distribución de agua**. Al igual que una tubería, **un flujo tiene dos extremos**.

En un extremo está el productor de datos, también conocido como *escritor*, que coloca una secuencia de valores de datos en el flujo. En el **otro extremo** está el **consumidor de datos, también conocido como *lector*, que recibe la secuencia de valores de datos del flujo.** El **escritor y el lector tienen cada uno su propia API que les permite trabajar con el flujo, como se muestra en la figura 3.**

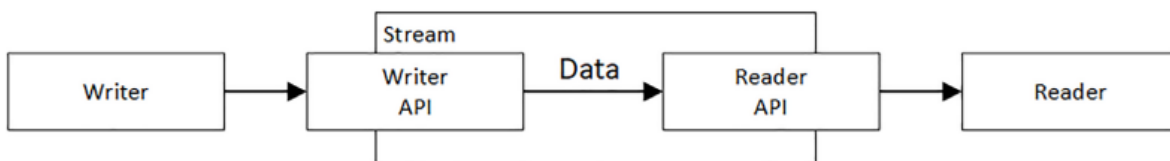


Figura 3: La anatomía de un flujo.

Esta disposición tiene dos características importantes. **La primera es que los datos llegan en el mismo orden en que se escriben, por lo que los flujos suelen describirse como una secuencia de valores de datos.**

La segunda característica es que los valores de datos se pueden escribir en el flujo a lo largo del tiempo, de modo que el escritor no tiene que tener todos los valores de datos listos antes de que se escriba el primer valor. Esto significa que el lector puede recibir y comenzar a procesar datos mientras el escritor aún está preparando o calculando valores posteriores en la secuencia. Esto hace que los flujos sean adecuados para una amplia gama de fuentes de datos y también se integran bien con el modelo de programación de Node.js, como lo demostrarán los ejemplos de este documento.

Uso de flujos de Node.js

El módulo de flujos contiene clases que representan diferentes tipos de flujos, y los dos más importantes se describen en la tabla 3.

Tabla 3: Clases de flujo útiles.

Nombre	Descripción
Writable	Esta clase proporciona la API para escribir datos en un flujo.
Readable	Esta clase proporciona la API para leer datos de un flujo.

En el desarrollo de Node.js, un extremo de un flujo suele estar conectado a algo fuera del entorno de JavaScript, como una conexión de red o el sistema de archivos, y esto permite que los datos se lean y escriban de la misma manera independientemente de dónde vayan o de dónde vengan.

Para el desarrollo web, el uso más importante de los flujos es que se utilizan para representar solicitudes y respuestas HTTP. Las clases `IncomingMessage` y `ServerResponse`, que se utilizan para representar solicitudes y respuestas HTTP, se derivan de las clases `Readable` y `Writable`.

Escritura de datos en un flujo

La clase `Writable` se utiliza para escribir datos en un flujo. Las características más útiles que ofrece la clase `Writable` se describen en la tabla 4 y se explican en las secciones siguientes.

Tabla 4: Características útiles de la clase `Writable`.

Nombre	Descripción
<code>write(data, callback)</code>	Este método escribe datos en el flujo e invoca la función de devolución de llamada opcional cuando se han vaciado los datos. Los datos se pueden expresar como una cadena, un búfer o un arreglo.

	<p>UInt8. Para los valores de cadena, se puede especificar una codificación opcional.</p> <p>El método devuelve un valor booleano que indica si el flujo puede aceptar más datos sin exceder el tamaño de su búfer, como se describe en la sección cómo evitar el almacenamiento excesivo de datos en búfer.</p>
end(data, callback)	Este método le indica a Node.js que no se enviarán más datos. Los argumentos son un fragmento final opcional de datos para escribir y una función de devolución de llamada opcional que se invocará cuando se terminen de procesar los datos.
destroy(error)	Este método destruye el flujo inmediatamente, sin esperar a que se procesen los datos pendientes.
closed	Esta propiedad devuelve verdadero si se ha cerrado el flujo.
destroyed	Esta propiedad devuelve verdadero si se ha llamado al método destroy .
writable	Esta propiedad devuelve verdadero si se puede escribir en el flujo, lo que significa que el flujo no ha finalizado, no ha encontrado un error ni se ha destruido.
writableEnded	Esta propiedad devuelve verdadero si se ha llamado al método end .
writableHighWaterMark	Esta propiedad devuelve el tamaño del búfer de datos en bytes. El método write devolverá falso cuando la cantidad de datos almacenados en búfer supere esta cantidad.
errored	Esta propiedad devuelve verdadero si el flujo ha encontrado un error.

La clase **Writable** también emite eventos, los más útiles de los cuales se describen en la tabla 5.

Tabla 5: Eventos útiles que se pueden escribir.

Nombre	Descripción
close	Este evento se emite cuando se cierra el flujo.
drain	Este evento se emite cuando el flujo puede aceptar datos sin almacenarlos en búfer.
error	Este evento se emite cuando se produce un error.
finish	Este evento se emite cuando se llama al método de finalización y se han procesado todos los datos del flujo.

El enfoque básico para usar un flujo que se puede escribir es llamar al método **write** hasta que todos los datos se hayan enviado al flujo y luego llamar al método **end**, como se muestra en el listado 4.

Listado 4: Escritura de datos en el archivo **handler.ts** en la carpeta **src**.

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

export const basicHandler = (req: IncomingMessage, resp: ServerResponse) => {

  for (let i = 0; i < 10; i++) {
```

```
    resp.write(`Message: ${i}\n`);  
  }  
  
  resp.end("End");  
};
```

Guarda los cambios, permite que Node.js se reinicie y luego solicite `http://localhost:5000`. El controlador escribirá sus datos en el flujo de respuesta, lo que producirá el resultado que se muestra en la figura 4.

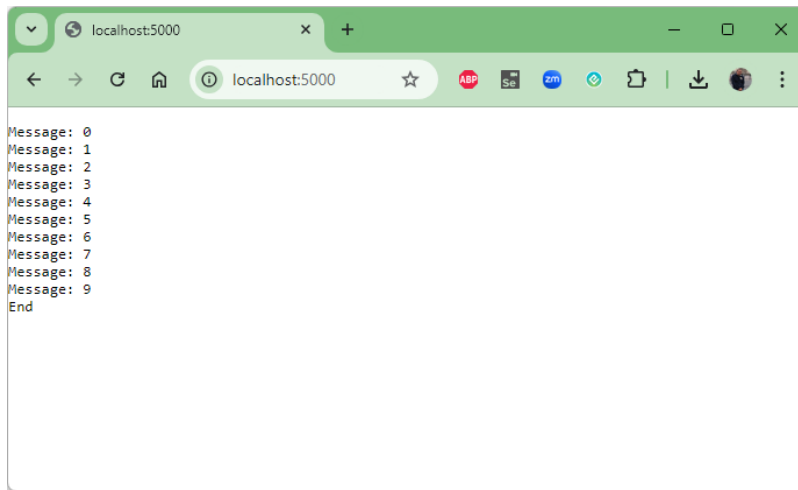


Figura 4: Escritura de datos en un flujo de respuesta HTTP.

Es fácil pensar en el punto final del flujo como una tubería directa al destinatario final de los datos, que es el navegador web en este caso, pero esa rara vez es el caso. El punto final de la mayoría de los flujos es la parte de la API de Node.js que interactúa con el sistema operativo, en este caso, el código que se ocupa de la pila de red del sistema operativo para enviar y recibir datos. Esta relación indirecta conduce a consideraciones importantes, como se describe en las siguientes secciones.

Comprensión de las mejoras de los flujos

Algunos flujos se mejoran para facilitar el desarrollo, lo que significa que los datos que escribes en el flujo no siempre serán los datos que se reciben en el otro extremo. En el caso de las respuestas HTTP, por ejemplo, la API HTTP de Node.js ayuda al desarrollo al garantizar que todas las respuestas cumplan con los requisitos básicos del protocolo HTTP, incluso cuando el programador no utiliza explícitamente las funciones proporcionadas para configurar el código de estado y los encabezados. Para ver el contenido que el ejemplo del

listado 4 escribe en el flujo, abre un nuevo símbolo del sistema y ejecute el comando de Linux que se muestra en el listado 5.

Listado 5: Cómo realizar una solicitud HTTP (Linux).

```
curl --include http://localhost:5000
```

Si eres un usuario de Windows, utiliza PowerShell para ejecutar el comando que se muestra en el listado 6.

Listado 6: Realizar una solicitud HTTP (Windows).

```
(Invoke-WebRequest http://localhost:5000).RawContent
```

Estos comandos facilitan la visualización de la respuesta completa enviada por Node.js. El código del Listado 4 utiliza solo los métodos `write` y `end`, pero la respuesta HTTP será así:

```
HTTP/1.1 200 OK
Connection: keep-alive
Keep-Alive: timeout=5
Transfer-Encoding: chunked
Date: Tue, 10 Sep 2024 23:59:23 GMT
X-Powered-By: Express
```

```
Message: 0
Message: 1
Message: 2
Message: 3
Message: 4
Message: 5
Message: 6
Message: 7
Message: 8
Message: 9
End
```

La API HTTP de Node.js se asegura de que la respuesta sea HTTP legal agregando un número de versión HTTP, un código de estado y un mensaje, y un conjunto mínimo de encabezados. Esta es una característica útil y ayuda a ilustrar el hecho de que no puedes asumir que los datos que escribes en una secuencia serán los datos que llegan al otro extremo.

La clase `ServerResponse` demuestra otro tipo de mejora de la secuencia, que son métodos o propiedades que escriben contenido en la secuencia por usted, como se muestra en el listado 7.

Listado 7: Uso de un método de mejora de la secuencia en el archivo handler.ts en la carpeta src.

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

export const basicHandler = (req: IncomingMessage, resp: ServerResponse) => {

  resp.setHeader("Content-Type", "text/plain");

  for (let i = 0; i < 10; i++) {
    resp.write(`Message: ${i}\n`);
  }

  resp.end("End");
};
```

Detrás de escena, la clase `ServerResponse` fusiona los argumentos pasados al método `setHeader` con el contenido predeterminado utilizado para las respuestas. La clase `ServerResponse` se deriva de `Writable` e implementa los métodos y propiedades descritos en la tabla 4, pero las mejoras facilitan la escritura de contenido en el flujo que es específico para las solicitudes HTTP, como la configuración de un encabezado en la respuesta. Si ejecutas los comandos que se muestran en el listado 6 o el listado 7 nuevamente, verás el efecto de llamar al método `setHeader`:

```
HTTP/1.1 200 OK
Connection: keep-alive
Keep-Alive: timeout=5
Transfer-Encoding: chunked
Content-Type: text/plain
Date: Wed, 11 Sep 2024 00:02:33 GMT
X-Powered-By: Express
```

```
Message: 0
Message: 1
Message: 2
Message: 3
Message: 4
Message: 5
Message: 6
Message: 7
Message: 8
Message: 9
End
```

Cómo evitar el almacenamiento excesivo de datos en búfer

Los flujos escribibles (writable) se crean con un búfer en el que se almacenan los datos antes de procesarlos. El búfer es una forma de mejorar el rendimiento, al permitir que el productor de datos escriba datos en el flujo en ráfagas más rápido de lo que el punto final del flujo puede procesarlos.

Cada vez que el flujo procesa un fragmento de datos, se dice que ha vaciado (*flushed*) los datos. Cuando se han procesado todos los datos en el búfer del flujo, se dice que el búfer del flujo se ha vaciado. La cantidad de datos que se pueden almacenar en el búfer se conoce como marca de agua alta (*high-water mark*).

Un flujo escribible siempre aceptará datos, incluso si tiene que aumentar el tamaño de su búfer, pero esto no es deseable porque aumenta la demanda de memoria que puede requerirse durante un período prolongado mientras el flujo vacía los datos que contiene.

El enfoque ideal es escribir datos en un flujo hasta que su búfer esté lleno y luego esperar hasta que se vacíen esos datos antes de escribir más datos. Para ayudar a lograr este objetivo, el método `write` devuelve un valor booleano que indica si el flujo puede recibir más datos sin expandir su búfer más allá de su marca de agua alta objetivo.

El listado 8 usa el valor devuelto por el método `write` para indicar cuándo el búfer del flujo ha alcanzado su capacidad.

Listado 8: Verificación de la capacidad del flujo en el archivo `handler.ts` en la carpeta `src`.

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

export const basicHandler = (req: IncomingMessage, resp: ServerResponse) => {

  resp.setHeader("Content-Type", "text/plain");

  for (let i = 0; i < 10_000; i++) {
    if (resp.write(`Message: ${i}\n`)) {
      console.log("Stream buffer is at capacity");
    }
  }

  resp.end("End");
};
```

Es posible que debas aumentar el valor máximo utilizado por el ciclo `for`, pero para mi PC de desarrollo, escribir rápidamente 10 000 mensajes en el flujo alcanzará de manera confiable

los límites del flujo. Utiliza un navegador para solicitar `http://localhost:5000` y verás mensajes como estos producidos por la consola de Node.js:

```
Stream buffer is at capacity
Stream buffer is at capacity
Stream buffer is at capacity
Stream buffer is at capacity
Stream buffer is at capacity
Stream buffer is at capacity
Stream buffer is at capacity
Stream buffer is at capacity
Stream buffer is at capacity
Stream buffer is at capacity
Stream buffer is at capacity
Stream buffer is at capacity
Stream buffer is at capacity
Stream buffer is at capacity
Stream buffer is at capacity
Stream buffer is at capacity
Stream buffer is at capacity
Stream buffer is at capacity
Stream buffer is at capacity
Stream buffer is at capacity
Stream buffer is at capacity
Stream buffer is at capacity
Stream buffer is at capacity
Stream buffer is at capacity
```

Los flujos de escritura emiten el evento de vaciado cuando sus búferes se han vaciado, momento en el que se pueden escribir más datos. En el listado 9, los datos se escriben en el flujo de respuesta HTTP hasta que el método de escritura devuelve falso y luego deja de escribir hasta que se recibe el evento de vaciado. (Si deseas saber cuándo se vacía un fragmento individual de datos, puedes pasar una función de devolución de llamada al método de escritura del flujo).

Listado 9: Cómo evitar el almacenamiento excesivo de datos en búfer en el archivo `handler.ts` en la carpeta `src`.

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

export const basicHandler = (req: IncomingMessage, resp: ServerResponse) => {

  resp.setHeader("Content-Type", "text/plain");

  let i = 0;
  let canWrite = true;

  const writeData = () => {
    console.log("Started writing data");
    do {
      canWrite = resp.write(`Message: ${i++}\n`);
    } while (i < 10_000 && canWrite);
  }
```

```

    console.log("Buffer is at capacity");
    if (i < 10_000) {
      resp.once("drain", () => {
        console.log("Buffer has been drained");
        writeData();
      });
    } else {
      resp.end("End");
    }
  }

  writeData();
};

```

La función `writeData` ingresa en un ciclo `do...while` que escribe datos en el flujo hasta que el método de escritura devuelve falso. El método `once` se utiliza para registrar un controlador que se invocará una vez cuando se emita el evento de vaciado y que invoca la función `writeData` para reanudar la escritura.

Una vez que se han escrito todos los datos, se llama al método `end` para finalizar el flujo.

Cómo evitar el error de finalización anticipada

Un error común (y que cometemos con regularidad) es poner la llamada al método de finalización fuera de las funciones de devolución de llamada que escriben los datos, de esta manera:

```

...
const writeData = () => {
  console.log("Started writing data");
  do {
    canWrite = resp.write(`Message: ${i++}\n`);
  } while (i < 10_000 && canWrite);
  console.log("Buffer is at capacity");
  if (i < 10_000) {
    resp.once("drain", () => {
      console.log("Buffer has been drained");
      writeData();
    });
  }
}

writeData();
resp.end("End");
...

```

El resultado puede variar, pero suele ser un error porque la devolución de llamada invocará el método de escritura después de que se haya cerrado el flujo, o no se escribirán todos los datos en el flujo porque no se emitirá el evento de vaciado. Para evitar este error, asegúrate de que el método de finalización se invoque dentro de la función de devolución de llamada una vez que se hayan escrito los datos.

Utiliza un navegador para solicitar `http://localhost:5000` y verás mensajes de la consola de Node.js que muestran que la escritura se detiene cuando el búfer alcanza su capacidad máxima y se reanuda una vez que se agota:

```
...
Started writing data
Buffer is at capacity
Buffer has been drained
Started writing data
Buffer is at capacity
Buffer has been drained
Started writing data
Buffer is at capacity
...
```

Lectura de datos de un flujo

La fuente de datos más importante en una aplicación web proviene de los cuerpos de las solicitudes HTTP. El proyecto de ejemplo necesita un poco de preparación para que el código del lado del cliente pueda realizar una solicitud HTTP con un cuerpo. **Agrega un archivo llamado `index.html` a la carpeta estática** con el contenido que se muestra en el listado 10.

Listado 10: El contenido del archivo `index.html` en la carpeta `static`.

```
<!DOCTYPE html>
<html>
  <head>
    <script>
      document.addEventListener('DOMContentLoaded', function() {
        document.getElementById("btn")
          .addEventListener("click", sendReq);
      });
      sendReq = async () => {
        let payload = "";
        for (let i = 0; i < 10_000; i++) {
          payload += `Payload Message: ${i}\n`;
        }
        const response = await fetch("/read", {
          method: "POST", body: payload
        })
        document.getElementById("msg").textContent
          = response.statusText;
        document.getElementById("body").textContent
          = await response.text();
      }
    </script>
  </head>
  <body>
```

```

    <button id="btn">Send Request</button>
    <div id="msg"></div>
    <div id="body"></div>
  </body>
</html>

```

Este es un documento HTML simple que contiene algo de código JavaScript. Haremos mejoras más adelante en el documento, incluida la separación del contenido JavaScript y HTML en archivos separados, pero esto es suficiente para comenzar. El código JavaScript en el listado 10 usa la API Fetch del navegador para enviar una solicitud HTTP POST con un cuerpo que contiene 1000 líneas de texto. El listado 11 actualiza el manejador de solicitudes existente para que responda con el contenido del archivo HTML.

Listado 11: Actualización de los manejadores en el archivo handler.ts en la carpeta src.

```

import { IncomingMessage, ServerResponse } from "http";
import { readFileSync } from "fs";

export const basicHandler = (req: IncomingMessage, resp: ServerResponse) => {
  resp.write(readFileSync("static/index.html"));
  resp.end();
};

```

Utilizamos la función readFileSync para realizar una lectura de bloqueo del archivo index.html, que es simple pero no es la mejor manera de leer archivos, como explicaremos más adelante.

Para crear un nuevo manejador que se utilizará para leer los datos enviados por el navegador, agrega un archivo llamado readHandler.ts a la carpeta src con el contenido que se muestra en el listado 12. Por el momento, este manejador es un marcador de posición que finaliza la respuesta sin producir ningún contenido.

Listado 12: El contenido del archivo readHandler.ts en la carpeta src.

```

import { IncomingMessage, ServerResponse } from "http";

export const readHandler = (req: IncomingMessage, resp: ServerResponse) => {
  // TODO - read request body
  resp.end();
}

```

El listado 13 completa la preparación agregando una ruta que coincide con las solicitudes POST y las envía al nuevo manejador.

Listado 13: Cómo agregar una ruta en el archivo server.ts de la carpeta src.

```
import { createServer } from "http";
import express, { Express } from "express";
import { basicHandler } from "../handler";
import { readHandler } from "../readHandler";

const port = 5000;

const expressApp: Express = express();

expressApp.get("/favicon.ico", (req, resp) => {
  resp.statusCode = 404;
  resp.end();
});
expressApp.get("*", basicHandler);
expressApp.post("/read", readHandler);

const server = createServer(expressApp);

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

Utiliza un navegador para solicitar `http://localhost:5000` y verás el botón definido por el documento HTML. Haz clic en el botón y el navegador enviará una solicitud HTTP POST y mostrará el mensaje de estado de la respuesta que recibe, como se muestra en la figura 5. El contenido presentado por el navegador no tiene ningún estilo, pero esto es suficiente por ahora.

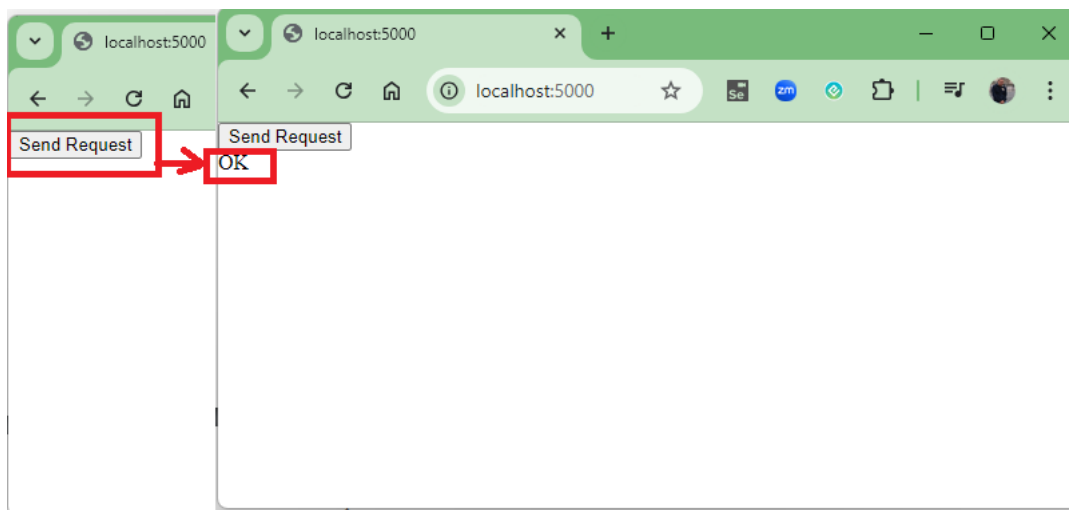


Figura 5: Cómo enviar una solicitud HTTP POST.

Entendiendo la clase Readable

La clase Readable se utiliza para leer datos de un flujo. La tabla 6 describe las características Readable más útiles.

Tabla 6: Características Readable útiles.

Nombre	Descripción
<code>pause()</code>	Al llamar a este método, se le indica al flujo que deje de emitir temporalmente el evento de datos.
<code>resume()</code>	Al llamar a este método, se le indica al flujo que reanude la emisión del evento de datos.
<code>isPaused()</code>	Este método devuelve verdadero si los eventos de datos del flujo se han pausado.
<code>pipe(writable)</code>	Este método se utiliza para transferir los datos del flujo a un Writable.
<code>destroy(error)</code>	Este método destruye el flujo inmediatamente, sin esperar a que se procesen los datos pendientes.
<code>closed</code>	Esta propiedad devuelve verdadero si el flujo se ha cerrado.
<code>destroyed</code>	Esta propiedad devuelve verdadero si se ha llamado al método <code>destroy</code> .
<code>errored</code>	Esta propiedad devuelve verdadero si el flujo ha encontrado un error.

La clase Readable también emite eventos, los más útiles son los que se describen en la tabla 7.

Tabla 6.7: Eventos Readable útiles.

Nombre	Descripción
<code>data</code>	Este evento se emite cuando el flujo está en modo de flujo y proporciona acceso a los datos del flujo. Consulta la sección Lectura de datos con eventos para obtener más detalles.
<code>end</code>	Este evento se emite cuando ya no hay más datos para leer del flujo.
<code>close</code>	Este evento se emite cuando se cierra el flujo.
<code>pause</code>	Este evento se emite cuando se pausa la lectura de datos al llamar al método <code>pause</code> .
<code>resume</code>	Este evento se emite cuando se reinicia la lectura de datos al llamar al método <code>resume</code> .
<code>error</code>	Este evento se activa si hay un error al leer los datos del flujo.

Lectura de datos con eventos

Los datos se pueden leer del flujo mediante eventos, como se muestra en el listado 14, donde se utiliza una función de devolución de llamada (callback) para procesar los datos a medida que están disponibles.

Listado 14: Lectura de datos en el archivo `readHandler.ts` en la carpeta `src`.

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



```
export const readHandler = (req: IncomingMessage, resp: ServerResponse) => {
  req.setEncoding("utf-8");

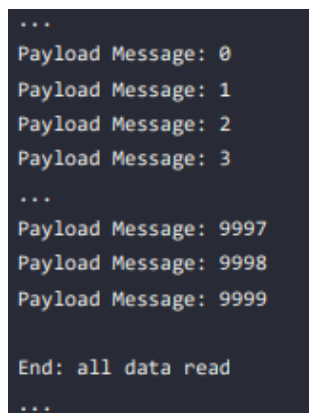
  req.on("data", (data: string) => {
    console.log(data);
  });

  req.on("end", () => {
    console.log("End: all data read");
    resp.end();
  });
}
```

El evento `data` se emite cuando los datos están disponibles para ser leídos del flujo y están disponibles para su procesamiento por la función de devolución de llamada utilizada para manejar el evento. Los datos se pasan a la función de devolución de llamada como un `Buffer`, que representa un arreglo de bytes sin signo, a menos que se haya utilizado el método `setEncoding` para especificar la codificación de caracteres, en cuyo caso los datos se expresan como una cadena.

Este ejemplo establece la codificación de caracteres en UTF-8 para que la función de devolución de llamada para el evento de datos reciba valores de cadena, que luego se escriben utilizando el método `console.log`.

El evento final se emite cuando se han leído todos los datos de la secuencia. Para evitar una variación del problema de finalización anticipada que describimos anteriormente, llamamos al método final de la respuesta solo cuando se emite el método final de la secuencia legible. Utiliza un navegador para solicitar `http://localhost:5000` y haz clic en el botón `Enviar solicitud` (Send Request); verás una secuencia de mensajes de consola de Node.js a medida que se leen los datos de la secuencia:



```
...
Payload Message: 0
Payload Message: 1
Payload Message: 2
Payload Message: 3
...
Payload Message: 9997
Payload Message: 9998
Payload Message: 9999
End: all data read
...
```

El hilo principal de JavaScript garantiza que los eventos de datos se procesen de manera secuencial, pero la idea básica es que los datos se lean y procesen lo más rápido posible, de modo que el evento de datos se emita lo antes posible una vez que los datos estén disponibles para leerse.

Lectura de datos con un iterador

Las instancias de la clase Readable se pueden utilizar como una fuente de datos en un ciclo for, que puede proporcionar una forma más familiar de leer datos de un flujo, como se muestra en el listado 15.

Listado 15: Lectura de datos en un bucle en el archivo readHandler.ts en la carpeta src.

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

export const readHandler = async (req: IncomingMessage, resp: ServerResponse) => {
  req.setEncoding("utf-8");

  for await (const data of req) {
    console.log(data);
  }
  console.log("End: all data read");
  resp.end();
}
```

Las palabras clave async y await se deben utilizar como se muestra en el ejemplo, pero el resultado es que el ciclo for lee datos del flujo hasta que se consumen todos. Este ejemplo produce el mismo resultado que el listado 14.

Conducción (piping) de datos a un flujo escribible

El método pipe se utiliza para conectar un flujo Readable a un flujo Writable, lo que garantiza que todos los datos se lean del Readable y se escriban en el Writable sin más intervención, como se muestra en el listado 16.

Listado 16: Conducción de datos al archivo readHandler.ts en la carpeta src.

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

export const readHandler = async (req: IncomingMessage, resp: ServerResponse) => {
  req.pipe(resp);
}
```

Esta es la forma más sencilla de transferir datos entre flujos, y el método final se llama automáticamente en el flujo de escritura una vez que se han transferido todos los datos. Utiliza un navegador para solicitar `http://localhost:5000` y haz clic en el botón Enviar solicitud. Los datos que se envían en la solicitud HTTP se canalizan a la respuesta HTTP y se muestran en la ventana del navegador, como se muestra en la figura 6.

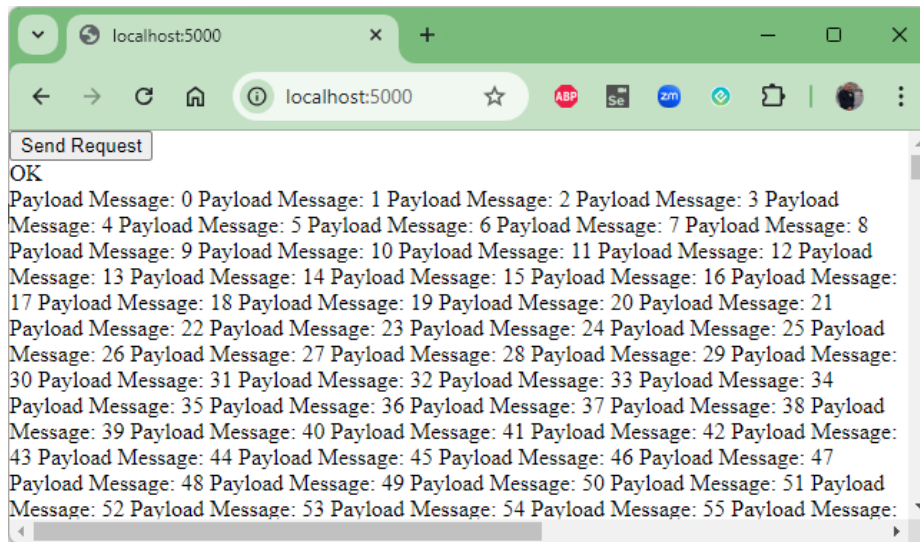


Figura 6: Canalización de datos.

Transformación de datos

La clase Transform se utiliza para crear objetos, conocidos como *transformadores*, que reciben datos de un flujo de lectura, los procesan de alguna manera y luego los pasan. Los transformadores se aplican a los flujos con el método de canalización, como se muestra en el listado 17.

Listado 17: Creación de un transformador en el archivo `readHandler.ts` en la carpeta `src`.

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

export const readHandler = async (req: IncomingMessage, resp: ServerResponse) => {
  req.pipe(createLowerTransform()).pipe(resp);
}

const createLowerTransform = () => new Transform({
  transform(data, encoding, callback) {
    callback(null, data.toString().toLowerCase());
  }
});
```

El argumento del constructor Transform es un objeto cuyo valor de propiedad transform es una función que se invocará cuando haya datos para procesar. La función recibe tres argumentos: un fragmento de datos para procesar, que puede ser de cualquier tipo de datos, un tipo de codificación de cadena y una función de devolución de llamada que se utiliza para pasar los datos transformados.

En este ejemplo, los datos que se reciben se convierten en una cadena en la que se llama al método toLowerCase. El resultado se pasa a la función de devolución de llamada, cuyos argumentos son un objeto que representa cualquier error que haya ocurrido y los datos transformados.

El transformador se aplica con el método pipe y, en este caso, se encadena de modo que los datos leídos de la solicitud HTTP se transformen y luego se escriban en la respuesta HTTP. Ten en cuenta que se debe crear un nuevo objeto Transform para cada solicitud, de esta manera:

```
...  
req.pipe(createLowerTransform()).pipe(resp);  
...
```

Usa un navegador para solicitar `http://localhost:5000` y haz clic en el botón Enviar solicitud. El contenido que muestra el navegador, que proviene del cuerpo de la respuesta HTTP, está todo en minúsculas, como se muestra en la figura 7.

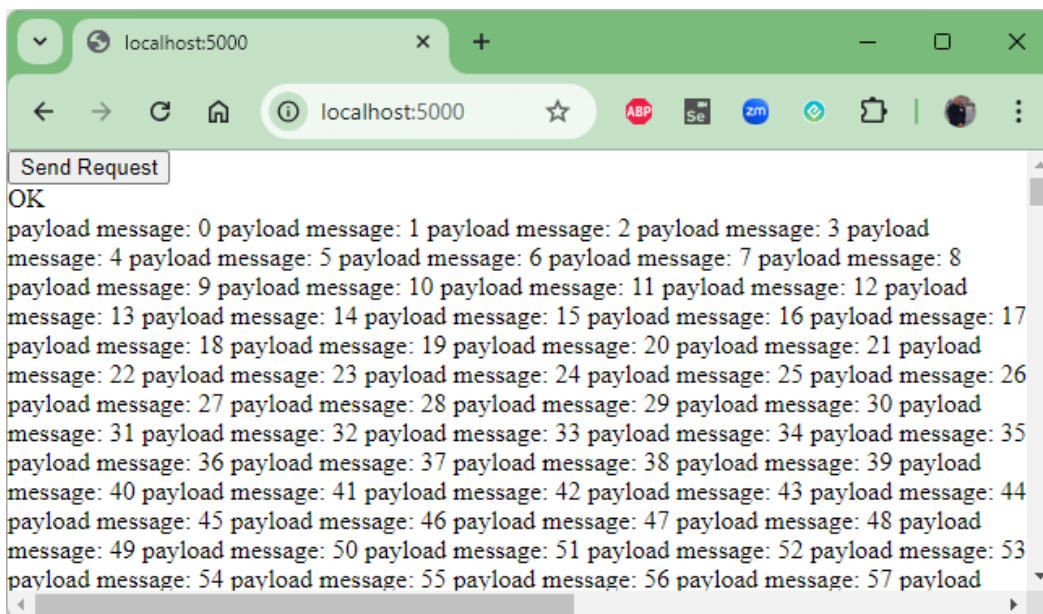


Figura 7: Uso de un transformador simple.

Uso del modo objeto

Los flujos creados por la API de Node.js, como los que se usan para solicitudes HTTP o archivos, funcionan solo con cadenas y arreglos de bytes. Esto no siempre es conveniente, por lo que algunos flujos, incluidos los transformadores, pueden usar el modo objeto, que permite leer o escribir objetos. Para prepararse para este ejemplo, el listado 18 actualiza el código JavaScript contenido dentro del archivo HTML estático para enviar una solicitud que contiene un arreglo de objetos con formato JSON.

Listado 18: Envío de un cuerpo de solicitud JSON en el archivo index.html en la carpeta static.

```
<!DOCTYPE html>
<html>
  <head>
    <script>
      document.addEventListener('DOMContentLoaded', function() {
        document.getElementById("btn").addEventListener("click", sendReq);
      });
      sendReq = async () => {
        let payload = [];
        for (let i = 0; i < 5; i++) {
          payload.push({ id: i, message: `Payload Message: ${i}\n` });
        }
        const response = await fetch("/read", {
          method: "POST", body: JSON.stringify(payload),
          headers: {
            "Content-Type": "application/json"
          }
        });
        document.getElementById("msg").textContent = response.statusText;
        document.getElementById("body").textContent = await response.text();
      }
    </script>
  </head>
  <body>
    <button id="btn">Send Request</button>
    <div id="msg"></div>
    <div id="body"></div>
  </body>
</html>
```

Los datos enviados por el cliente aún se pueden leer como una cadena o un arreglo de bytes, pero se puede usar una transformación para convertir la carga útil de la solicitud en un objeto

JavaScript o convertir un objeto JavaScript en una cadena o arreglo de bytes, conocido como modo objeto. Se utilizan dos ajustes de configuración del constructor Transform para indicarle a Node.js cómo se comportará un transformador, como se describe en la tabla 8.

Tabla 8: Configuración del constructor de transformación.

Nombre	Descripción
readableObjectMode	Cuando se configura como verdadero, el transformador consumirá datos de cadena/byte y producirá un objeto.
writableObjectMode	Cuando se configura como verdadero, el transformador consumirá un objeto y producirá datos de cadena/byte.

El listado 19 muestra un transformador que establece el parámetro readableObjectMode en verdadero, lo que significa que leerá datos de cadena de la carga útil de la solicitud HTTP, pero producirá un objeto JavaScript cuando se lean sus datos.

Listado 19: Análisis de JSON en el archivo readHandler.ts en la carpeta src.

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

export const readHandler = async (req: IncomingMessage, resp: ServerResponse) => {
  if (req.headers["content-type"] === "application/json") {
    req.pipe(createFromJsonTransform()).on("data", (payload) => {
      if (payload instanceof Array) {
        resp.write(`Received an array with ${payload.length} items`);
      } else {
        resp.write("Did not receive an array");
      }
    });
    resp.end();
  } else {
    req.pipe(resp);
  }
}

const createFromJsonTransform = () => new Transform({
  readableObjectMode: true,
  transform(data, encoding, callback) {
    callback(null, JSON.parse(data));
  }
});
```

Si la solicitud HTTP tiene un encabezado Content-Type que indica que la carga útil es JSON, entonces se utiliza el transformador para analizar los datos, que son recibidos por el controlador de la solicitud mediante el evento de datos. La carga útil analizada se verifica para ver si es un arreglo y, si lo es, se utiliza su longitud para generar una respuesta.

Utiliza un navegador para solicitar `http://localhost:5000` (o asegúrate de volver a cargar el navegador para que los cambios en el listado 18 surtan efecto), haz clic en el botón Enviar solicitud y verás la respuesta que se muestra en la figura 8.

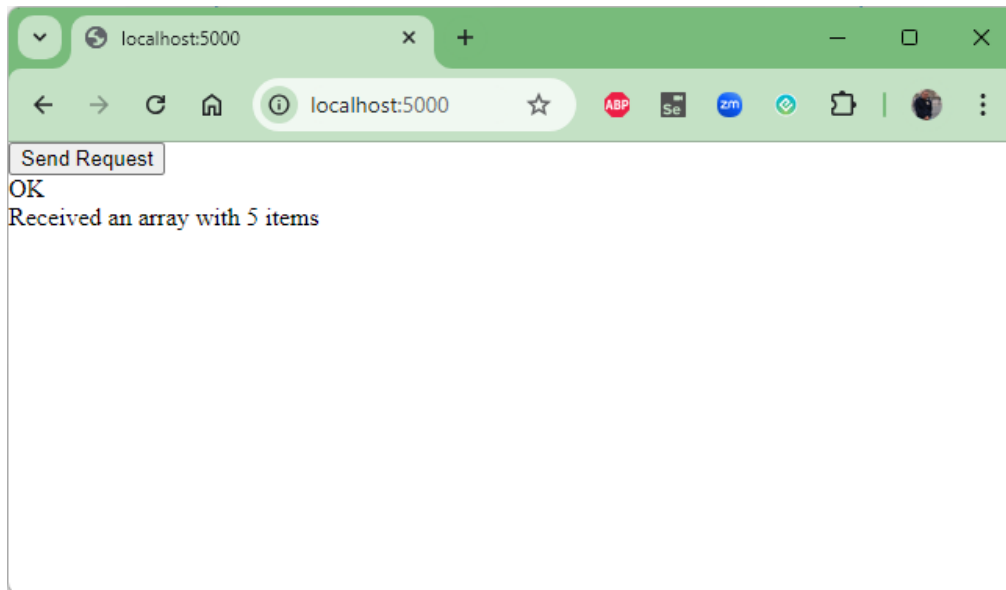


Figura 8: Uso de un transformador en modo de objeto.

Uso de mejoras de terceros

En las secciones que siguen, describimos mejoras útiles proporcionadas por el paquete Express para tratar con flujos y tareas relacionadas con HTTP. Express no es el único paquete que proporciona este tipo de funciones, pero es una buena opción predeterminada para nuevos proyectos y te brinda una base desde la cual comparar alternativas.

Trabajar con archivos

Una de las tareas más importantes de un servidor web es responder a las solicitudes de archivos, que proporcionan a los navegadores HTML, JavaScript y otro contenido estático que requiere la parte del lado del cliente de la aplicación.

Node.js proporciona una API integral para manejar archivos en el módulo `fs`, y tiene soporte para leer y escribir secuencias, junto con funciones prácticas que leen o escriben archivos completos, como la función `readFileSync` que usamos para leer el contenido de un archivo HTML.

La razón por la que no hemos descrito la API en detalle es que trabajar directamente con archivos dentro de un proyecto de servidor web es increíblemente peligroso y debe evitarse siempre que sea posible. Existe un gran margen para crear solicitudes maliciosas cuyas rutas intenten acceder a archivos fuera de las ubicaciones esperadas, por ejemplo. Y, a través de la experiencia personal y de amigos, hemos aprendido a no permitir que los clientes creen o modifiquen archivos en el servidor bajo ninguna circunstancia.

Los amigos han trabajado en demasiados proyectos en los que las solicitudes maliciosas pudieron sobrescribir archivos del sistema o simplemente abrumar a los servidores escribiendo tantos datos que se agotó el espacio de almacenamiento. La mejor manera de manejar archivos es usar un paquete bien probado, en lugar de escribir código personalizado, y es por esta razón que no hemos descrito las características del módulo `fs`.

Si decides ignorar nuestra advertencia, puedes encontrar detalles del módulo `fs` y las características que proporciona en <https://nodejs.org/dist/latest-v20.x/docs/api/fs.html>.

El paquete Express tiene soporte integrado para atender solicitudes de archivos. Para prepararte, agrega un archivo llamado `client.js` a la carpeta `static` con el contenido que se muestra en el listado 20.

Listado 20: El contenido del archivo `client.js` en la carpeta `static`.

```
document.addEventListener('DOMContentLoaded', function() {
  document.getElementById("btn").addEventListener("click", sendReq);
});

sendReq = async () => {
  let payload = [];
  for (let i = 0; i < 5; i++) {
    payload.push({ id: i, message: `Payload Message: ${i}\n` });
  }
  const response = await fetch("/read", {
    method: "POST", body: JSON.stringify(payload),
    headers: {
      "Content-Type": "application/json"
    }
  })
  document.getElementById("msg").textContent = response.statusText;
```



```

    document.getElementById("body").textContent = await response.text();
  }

```

Este es el mismo código JavaScript usado en ejemplos anteriores, pero colocado en un archivo separado, que es la forma típica de distribuir JavaScript a los clientes. El listado 21 actualiza el archivo HTML para vincularlo al nuevo archivo JavaScript, y también incluye el archivo de imagen que se agregó al proyecto al comienzo del documento.

Listado 21: Cambio de contenido en el archivo index.html en la carpeta static.

```

<!DOCTYPE html>
<html>
  <head>
    <script src="client.js"></script>
  </head>
  <body>
    
    <button id="btn">Send Request</button>
    <div id="msg"></div>
    <div id="body"></div>
  </body>
</html>

```

Una vez preparado el contenido, el siguiente paso es configurar Express para que sirva los archivos. Express incluye compatibilidad con componentes de middleware, lo que significa simplemente controladores de solicitudes que pueden inspeccionar e interceptar todas las solicitudes HTTP que recibe el servidor. Los componentes de middleware se configuran con el método `use` y el listado 22 configura el componente de middleware que Express proporciona para servir archivos.

Listado 22: Cómo agregar compatibilidad con archivos estáticos en el archivo server.ts de la carpeta src.

```

import { createServer } from "http";
import express, { Express } from "express";
//import { basicHandler } from "../handler";
import { readHandler } from "../readHandler";

const port = 5000;

const expressApp: Express = express();

//expressApp.get("/favicon.ico", (req, resp) => {
//  resp.statusCode = 404;

```

```
// resp.end();
//});
//expressApp.get("*", basicHandler);
expressApp.post("/read", readHandler);

expressApp.use(express.static("static"));

const server = createServer(expressApp);

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

El objeto `express`, que es la exportación predeterminada del módulo `express`, define un método denominado `static` que crea el componente de middleware que sirve archivos estáticos. El argumento del método `static` es el directorio que contiene los archivos, que también se denomina `static`. El resultado es un controlador de solicitudes que se puede registrar con el método `Express.use`.

El componente de middleware intentará hacer coincidir las URL de las solicitudes con los archivos del directorio estático. El nombre del directorio que contiene los archivos se omite en las URL, por lo que una solicitud para `http://localhost:5000/client.js`, por ejemplo, se manejará devolviendo el contenido del archivo `client.js` en la carpeta estática.

El método estático puede aceptar un objeto de configuración, pero los valores predeterminados están bien elegidos y se adaptan a la mayoría de los proyectos, incluido el uso de `index.html` como predeterminado para las solicitudes.

Si necesitas cambiar la configuración, puedes ver las opciones en:
<https://expressjs.com/en/4x/api.html#express.static>.

El componente de middleware establece los encabezados de respuesta para ayudar al cliente a procesar el contenido de los archivos que se utilizan. Esto incluye configurar el encabezado `Content-Length` para especificar la cantidad de datos que contiene el archivo y el encabezado `Content-Type` para especificar el tipo de datos.

Observa que podemos eliminar algunos de los controladores existentes del ejemplo. El controlador para `favicon`. Las solicitudes `ico` ya no son necesarias porque el nuevo middleware generará automáticamente respuestas de “no encontrado” cuando las solicitudes pidan archivos que no existen. La ruta `catch-all` ya no es necesaria porque el middleware estático responde a las solicitudes con el contenido del archivo `index.html`.

Utiliza un navegador para solicitar `http://localhost:5000` y verás la respuesta que se muestra en la figura 9, que también muestra los tipos de datos que ha recibido el navegador.

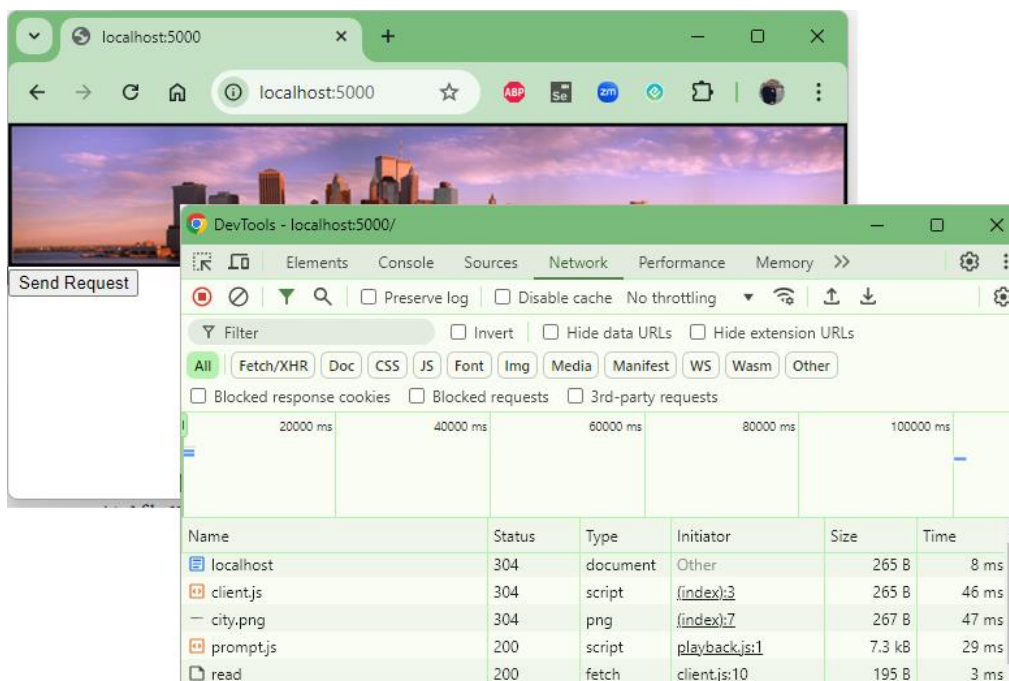


Figura 9: Uso del middleware static Express.

Entrega de archivos desde paquetes del lado del cliente

Una fuente de archivos estáticos son los paquetes que se agregan al proyecto Node.js, pero cuyos archivos están destinados al consumo por parte de navegadores (u otros clientes HTTP). Un buen ejemplo es el paquete CSS Bootstrap, que contiene hojas de estilo CSS y archivos JavaScript que se utilizan para dar estilo al contenido HTML que muestran los navegadores.

Si estás utilizando un framework del lado del cliente como Angular o React, estos archivos CSS y JavaScript se incorporarán en un solo archivo comprimido como parte del proceso de creación del proyecto.

Para los proyectos que no utilizan estos frameworks, la forma más sencilla de hacer que los archivos estén disponibles es configurar instancias adicionales del middleware de archivos estáticos. Para prepararte, ejecuta el comando que se muestra en el listado 23 en la carpeta webapp para agregar el paquete Bootstrap al proyecto de ejemplo.

Listado 23: Cómo agregar un paquete al proyecto de ejemplo.

```
npm install bootstrap@5.3.2
```

El listado 24 configura Express para servir archivos desde el directorio de paquetes.

Listado 24. Adición de middleware en el archivo server.ts en la carpeta src.

```
import { createServer } from "http";
import express, { Express } from "express";
import { readHandler } from "../readHandler";

const port = 5000;

const expressApp: Express = express();

expressApp.post("/read", readHandler);

expressApp.use(express.static("static"));
expressApp.use(express.static("node_modules/bootstrap/dist"));

const server = createServer(expressApp);

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

Se requiere cierto conocimiento de los paquetes que estás utilizando. En el caso del paquete **Bootstrap**, sabemos que los archivos utilizados por los clientes están en la carpeta **dist**, por lo que esta es la carpeta que especificamos al configurar los clientes de middleware. El paso final es agregar una referencia a una hoja de estilo Bootstrap y aplicar los estilos que contiene, como se muestra en el listado 25.

Listado 25. Adición de una referencia de hoja de estilo en el archivo index.html en la carpeta static.

```
<!DOCTYPE html>
<html>
  <head>
    <script src="client.js"></script>
    <link href="css/bootstrap.min.css" rel="stylesheet" />
  </head>
  <body>
    
    <button id="btn" class="btn btn-primary my-2">Send Request</button>
    <div id="msg"></div>
    <div id="body"></div>
  </body>
</html>
```

El archivo `bootstrap.min.css` contiene los estilos que queremos usar, que se aplican agregando el elemento de botón a las clases. Utiliza un navegador para solicitar `http://localhost:5000` y verás el efecto de los estilos, como se muestra en la figura 10.

Consulta <https://getbootstrap.com> para obtener detalles de las características que proporciona el paquete Bootstrap, algunas de las cuales utilizaremos en documentos posteriores. Hay otros paquetes CSS disponibles si no puedes manejar Bootstrap. Una alternativa popular es Tailwind (<https://tailwindcss.com>), pero una búsqueda rápida en la web te presentará una larga lista de alternativas para considerar.

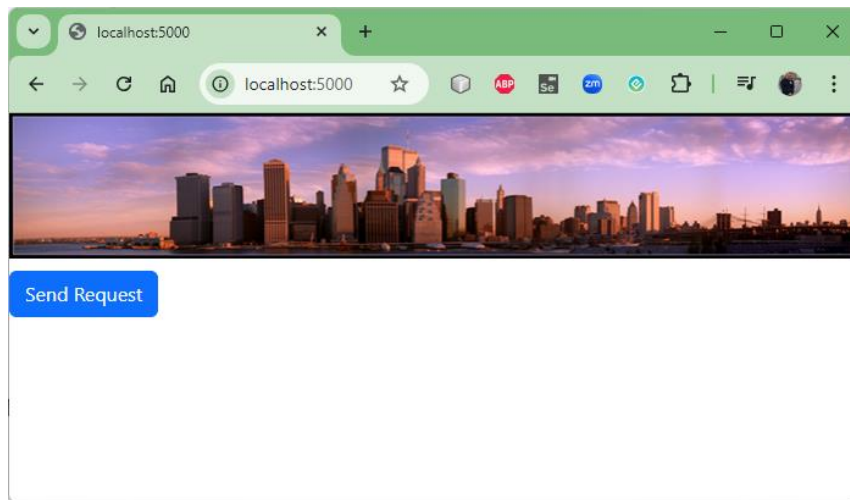


Figura 10: Uso de contenido estático de un paquete de terceros.

Envío y descarga de archivos

La clase `Response`, a través de la cual Express proporciona mejoras de `ServerResponse`, define los métodos descritos en la tabla 9 para tratar con archivos directamente.

Tabla 9: Métodos de respuesta útiles para archivos.

Nombre	Descripción
<code>sendFile(path, config)</code>	Este método envía el contenido del archivo especificado. El encabezado <code>Content-Type</code> de respuesta se establece en función de la extensión del archivo.
<code>download(path)</code>	Este método envía el contenido del archivo especificado de forma que la mayoría de los navegadores soliciten al usuario que guarde el archivo.

Los métodos `sendFile` y `download` son útiles porque brindan soluciones a problemas que no se pueden resolver utilizando el middleware estático. El listado 26 crea rutas simples que utilizan estos métodos.

Listado 26: Adición de rutas en el archivo server.ts en la carpeta src.

```
import { createServer } from "http";
import express, { Express, Request, Response } from "express";
import { readHandler } from "../readHandler";

const port = 5000;

const expressApp: Express = express();

expressApp.post("/read", readHandler);

expressApp.get("/sendcity", (req, resp) => {
  resp.sendFile("city.png", { root: "static" });
});

expressApp.get("/downloadcity", (req: Request, resp: Response) => {
  resp.download("static/city.png");
});

expressApp.get("/json", (req: Request, resp: Response) => {
  resp.json({ name: "Banco" });
});

expressApp.use(express.static("static"));
expressApp.use(express.static("node_modules/bootstrap/dist"));

const server = createServer(expressApp);

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

El método `sendFile` es útil cuando necesitas responder con el contenido de un archivo pero la ruta de solicitud no contiene el nombre del archivo. Los argumentos son el nombre del archivo y un objeto de configuración, cuya propiedad `root` especifica el directorio que contiene el archivo.

El método `download` establece el encabezado de respuesta `Content-Disposition`, que hace que la mayoría de los navegadores traten el contenido del archivo como una descarga que debe guardarse.

Use un navegador para solicitar `http://localhost:5000/sendcity` y `http://localhost:5000/downloadcity`.

La primera URL hará que el navegador muestre la imagen en la ventana del navegador. La segunda URL solicitará al usuario que guarde el archivo. Ambas respuestas se muestran en la figura 11.

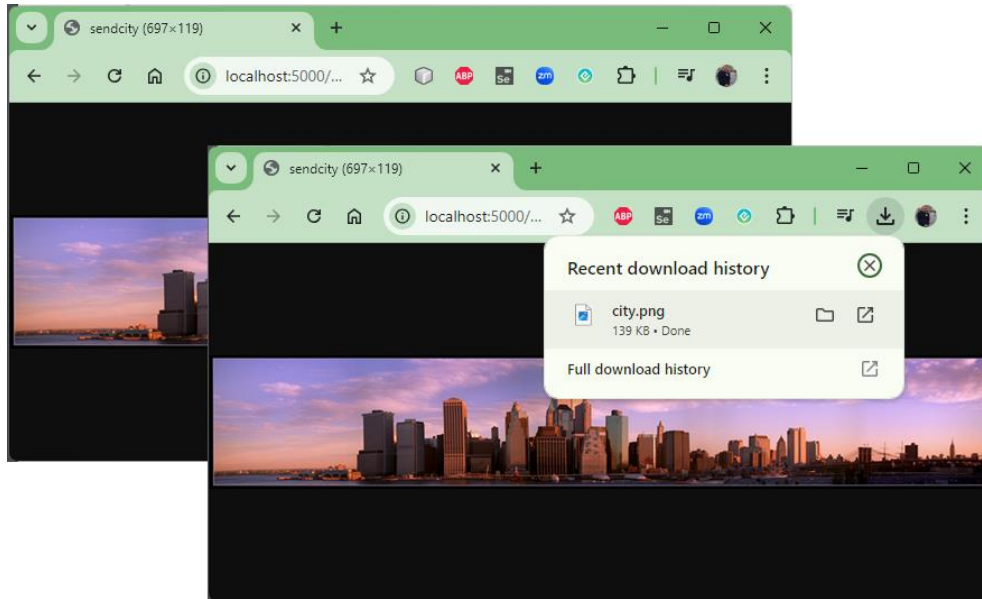


Figura 11: Uso de las mejoras de respuesta de archivo.

Decodificación y codificación automáticas de JSON

El paquete Express incluye un componente de middleware que decodifica automáticamente los cuerpos de respuesta JSON y realiza la misma tarea que el transformador de flujo que creamos anteriormente en este documento.

El listado 27 habilita este middleware al llamar al método json definido en la exportación predeterminada desde el módulo express.

Listado 27: Habilitación del middleware JSON en el archivo server.ts en la carpeta src.

```
import express, { Express, Request, Response } from "express";
import { readHandler } from "../readHandler";

const port = 5000;

const expressApp: Express = express();

expressApp.use(express.json());

expressApp.post("/read", readHandler);
```

```

expressApp.get("/sendcity", (req, resp) => {
  resp.sendFile("city.png", { root: "static" });
});

expressApp.get("/downloadcity", (req: Request, resp: Response) => {
  resp.download("static/city.png");
});

expressApp.get("/json", (req: Request, resp: Response) => {
  resp.json("{name: Bancho}");
});

expressApp.use(express.static("static"));
expressApp.use(express.static("node_modules/bootstrap/dist"));

const server = createServer(expressApp);

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

```

El componente de middleware debe registrarse antes de las rutas que lean los cuerpos de respuesta para que las solicitudes JSON se analicen antes de que coincidan con un controlador.

El método `json` puede aceptar un objeto de configuración que cambia la forma en que se analiza JSON. Los valores predeterminados son adecuados para la mayoría de los proyectos, pero consulta en:

<https://expressjs.com/en/4x/api.html#express.json> para obtener detalles de las opciones disponibles.

La clase `Request` a través de la cual Express proporciona mejoras a la clase `IncomingRequest` define una propiedad de cuerpo, a la que se le asigna el objeto creado por el middleware JSON.

El cuerpo `Response`, que proporciona mejoras `ServerResponse`, define el método `json`, que acepta un objeto que se serializa a JSON y se usa como cuerpo de respuesta.

El listado 28 actualiza el controlador para usar la clase `Request`, deshabilita el transformador personalizado y envía una respuesta JSON al cliente.

Listado 28: Uso del objeto JSON en el archivo `readHandler.ts` en la carpeta `src`.

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



```
//import { Transform } from "stream";
import { Request, Response } from "express";

export const readHandler = async (req: Request, resp: Response) => {
  if (req.headers["content-type"] === "application/json") {
    const payload = req.body;
    if (payload instanceof Array) {
      //resp.write(`Received an array with ${payload.length} items`)
      resp.json({ arraySize: payload.length });
    } else {
      resp.write("Did not receive an array");
    }
  }
  resp.end();
} else {
  req.pipe(resp);
}
}

//const createFromJsonTransform = () => new Transform({
//  readableObjectMode: true,
//  transform(data, encoding, callback) {
//    callback(null, JSON.parse(data));
//  }
//});
```

Utiliza un navegador web para solicitar `http://localhost:5000` y haz clic en el botón Enviar solicitud. La respuesta confirmará que el cuerpo de la solicitud JSON se analizó en un arreglo de JavaScript y que la respuesta también se envió como JSON, como se muestra en la figura 12.

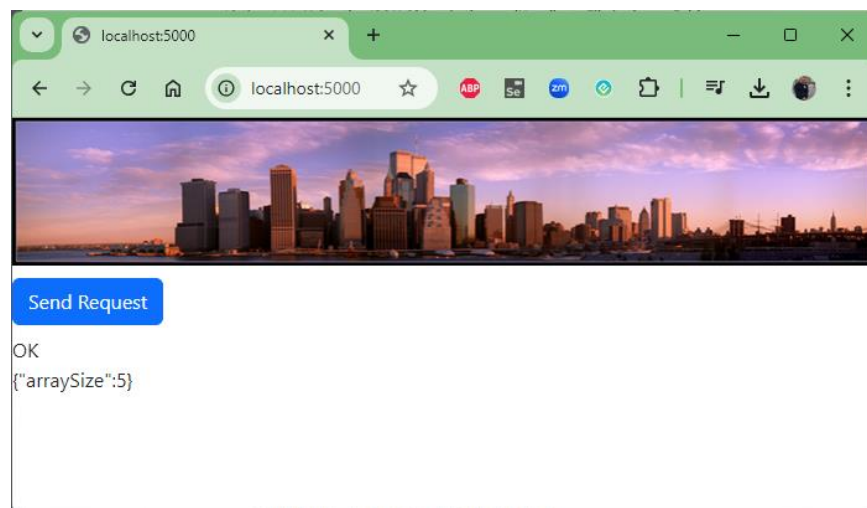


Figura 12: Uso del middleware Express JSON.

Resumen

En este documento, describimos las características de la API que Node.js proporciona para leer y escribir datos, en particular al procesar una solicitud HTTP:

- Los flujos se utilizan como representaciones abstractas de fuentes y destinos de datos, incluidas las solicitudes y respuestas HTTP.
- Los datos se almacenan en el búfer cuando se escriben en un flujo, pero es una buena idea evitar el almacenamiento en búfer excesivo porque puedes agotar los recursos del sistema.
- Los datos se pueden leer desde un flujo mediante el manejo de eventos o mediante un ciclo `for`.
- Los datos se pueden canalizar desde un flujo leíble a un flujo escribible.
- Los datos se pueden transformar a medida que se transmiten y pueden estar entre objetos JavaScript y cadenas/arreglos de bytes.
- Node.js proporciona una API para trabajar con archivos, pero los paquetes de terceros son la forma más segura de trabajar con archivos en un proyecto de servidor web.
- Los paquetes de terceros, como Express, proporcionan mejoras a los flujos de Node.js para realizar tareas comunes, como decodificar datos JSON.

En el siguiente documento, describiremos dos aspectos del desarrollo web en los que Node.js trabaja junto con otros componentes para entregar una aplicación.