Manejo de solicitudes (requests) HTTP

La base del desarrollo web del lado del servidor es la capacidad de recibir solicitudes HTTP de los clientes y generar respuestas. En este documento, presentamos la API de Node.js para crear servidores HTTP y explicaremos cómo se puede utilizar para recibir y responder solicitudes. La tabla 1 pone la API HTTP de Node.js en contexto.

Tabla 1: Poner la API de Node.js en contexto.

Pregunta	Respuesta
¿Qué es?	Los módulos http y https contienen las funciones y clases necesarias para crear servidores HTTP y HTTPS, recibir solicitudes y generar respuestas.
¿Por qué es útil?	Recibir y responder solicitudes HTTP es la característica principal del desarrollo de aplicaciones web del lado del servidor.
¿Cómo se utiliza?	Los servidores se crean con la función createServer, que emite eventos cuando se reciben solicitudes. Las funciones de devolución de llamada se invocan para manejar la solicitud y generar una respuesta.
¿Existen dificultades o limitaciones?	Las funciones de controlador pueden volverse complejas y mezclar las declaraciones que coinciden con las solicitudes con las declaraciones que generan respuestas. Los paquetes de terceros, como el paquete Express presentado en este documento, se basan en la API de Node.js para agilizar el manejo de solicitudes.
¿Existen alternativas?	No. Las API HTTP y HTTPS de Node.js son fundamentales para el desarrollo de aplicaciones web del lado del servidor. Los paquetes de terceros pueden hacer que la API sea más fácil de usar, pero se basan en las mismas características.

La tabla 2 resume el documento.

Tabla 2: Resumen del documento.

Problema	Solución	Listado
Listado de solicitudes HTTP	Usa la función createServer para crear un objeto Server y usa el método listen para comenzar a escuchar solicitudes.	4
Inspecciona una solicitud HTT	Usa las funciones proporcionadas por la clase IncomingRequest.	5
Analiza la URL de una solicitud	Usa la clase URL en el módulo url.	6
Crea una respuesta HTTP	Usa las funciones proporcionadas por la clase ServerResponse.	7
Escucha de solicitudes HTTP	Usa las funciones proporcionadas por el módulo https.	8, 9

Detecta solicitudes HTTPS	Verifica el valor de la propiedad socket.encrypted en el objeto IncomingRequest.	10
Redirecciona solicitudes inseguras	Envía un encabezado 302 al puerto HTTPS	11, 12
Simplifica el procesamiento de solicitudes	Usa un enrutador de terceros y clases de solicitud y respuesta mejoradas.	13-19

Preparación para este documento

En este documento, seguiremos utilizando el proyecto de aplicación web creado en el documento anterior. Para preparar este documento, reemplaza el contenido del archivo handler.ts en la carpeta src con el código que se muestra en el listado 1.

Listado 1: Reemplazo del contenido del archivo handler.ts en la carpeta src.

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

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

Reemplaza el contenido del archivo server.ts en la carpeta src con el código que se muestra en el listado 2.

Listado 2: Reemplazo del contenido del 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}`);
});
```

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
```

El archivo server.ts en la carpeta src se compilará para producir un archivo JavaScript puro llamado server.js en la carpeta dist.

El código JavaScript será ejecutado por el entorno de ejecución de Node.js, que comenzará a escuchar las solicitudes HTTP.

Abre un navegador web y solicita http://localhost:5000 y verás la respuesta que se muestra en la figura 1.

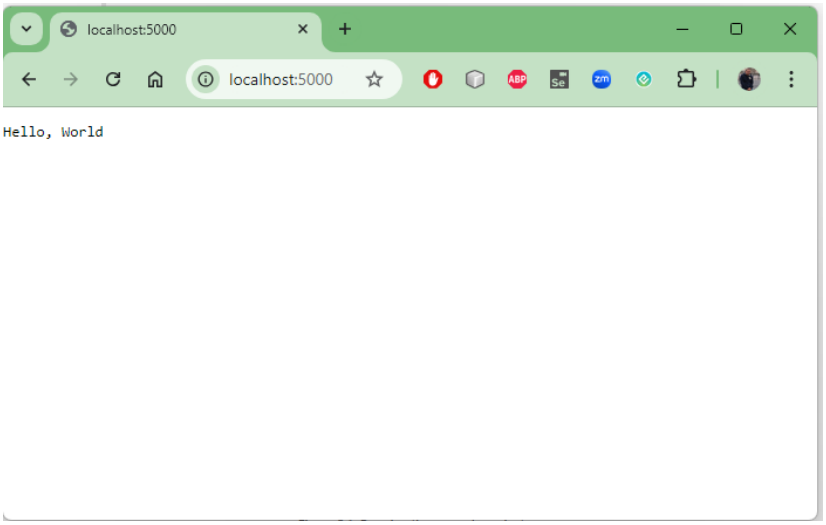

Figura 1: Ejecución del proyecto de ejemplo.

Escuchar solicitudes HTTP

En el documento anterior, creamos un servidor web simple para que pudiéramos demostrar la forma en que se ejecuta el código JavaScript. Al hacerlo, nos saltamos los detalles de cómo funcionaba el código, pero ahora es momento de volver atrás y profundizar en los detalles.

La función createServer en el módulo http se utiliza para crear objetos Server que se pueden usar para escuchar y procesar solicitudes HTTP. El objeto Server requiere configuración antes de comenzar a escuchar solicitudes y los métodos y propiedades más útiles definidos por la clase Server se describen en la tabla 3.

Una vez que se ha configurado el objeto Server, emite eventos que denotan cambios importantes en el estado. Los eventos más útiles se describen en la tabla 4.

Tabla 5.3: Métodos y propiedades útiles del servidor.

Nombre	Descripción
listen(port)	Este método comienza a escuchar solicitudes en un puerto específico.
close()	Este método deja de escuchar solicitudes.
requestTimeout	Esta propiedad obtiene o establece el período de espera de la solicitud, que también se puede utilizar mediante el objeto de configuración que se pasa a la función createServer.

Tabla 5.4: Eventos útiles del servidor.

Nombre	Descripción
listening	Este evento se activa cuando el servidor comienza a escuchar solicitudes.
request	Este evento se activa cuando se recibe una nueva solicitud. La función de devolución de llamada que maneja este evento se invoca con argumentos que representan la solicitud y la respuesta HTTP.
error	Este evento se activa cuando hay un error de red.

El uso de eventos para invocar funciones de devolución de llamada (callback) es típico del modelo de ejecución de código JavaScript descrito en el documento previo. El evento de solicitud se activará cada vez que se reciba una solicitud HTTP, y el modelo de ejecución de JavaScript significa que solo se manejará una solicitud HTTP a la vez.

La API de Node.js a menudo permite que los controladores de eventos se especifiquen a través de otros métodos. La función createServer utilizada para crear un objeto Server acepta un argumento de función opcional que se registra como un controlador para el evento de solicitud, y el método Server.listen acepta un argumento de función opcional que se utiliza para manejar el evento de escucha.

Estas características de conveniencia se pueden utilizar para combinar las declaraciones que crean y configuran el servidor HTTP con las funciones de devolución de llamada que manejan los eventos, como se muestra en el listado 4.

Listado 4: Uso de las características de conveniencia 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(handler);

//server.on("request", handler);
```

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

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

Este código tiene el mismo efecto que el listado 2, pero es más conciso y más fácil de leer.

Si hay error al ejecutar, prueben con esto (Taskkill /IM node.exe /F en la línea de comandos)

Descripción del objeto de configuración Server

Los argumentos para la función createServer son un objeto de configuración y una función de manejo de solicitudes. El objeto de configuración se utiliza para cambiar la forma en que se reciben las solicitudes, y las configuraciones más útiles se describen en la tabla 5.

Tabla 5: Configuraciones útiles del objeto de configuración createServer.

Nombre	Descripción
IncomingMessage	Esta propiedad especifica la clase que se utiliza para representar las solicitudes. El valor predeterminado es la clase IncomingMessage, definida en el módulo http.
ServerResponse	Esta propiedad especifica la clase que se utiliza para representar las respuestas. El valor predeterminado es la clase ServerResponse, definida en el módulo http.
requestTimeout	Esta propiedad especifica la cantidad de tiempo, en milisegundos, que se permite a un cliente para enviar solicitudes, después de lo cual la solicitud expira. El valor predeterminado es 300,000 milisegundos.

El objeto de configuración se puede omitir si se requieren los valores predeterminados. La función de controlador (handler) se invoca cuando se ha recibido una solicitud HTTP y sus parámetros son objetos cuyos tipos son los especificados por las propiedades IncomingMessage y ServerResponse, o los tipos predeterminados si no se ha modificado la configuración.

El código del listado 4 omite el objeto de configuración, lo que significa que se utilizarán los tipos predeterminados para representar la solicitud y la respuesta HTTP cuando se invoque la función de controlador para el evento de solicitud, de esta manera:

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

Los ejemplos posteriores de este documento demuestran el uso de diferentes tipos, pero las representaciones predeterminadas de la solicitud y la respuesta HTTP proporcionan todas las características necesarias para procesar HTTP, como se explica en las siguientes secciones.

Comprensión de las solicitudes HTTP

Node.js representa las solicitudes HTTP mediante la clase IncomingMessage, que se define en el módulo http. Los cuatro bloques de construcción principales de una solicitud HTTP son:

- El método HTTP, que describe la operación que el cliente desea realizar.
- La URL, que identifica el recurso al que se debe aplicar la solicitud.
- Los encabezados, que proporcionan información adicional sobre la solicitud y las capacidades del cliente.
- El cuerpo de la solicitud, que proporciona los datos necesarios para la operación solicitada.

La clase IncomingMessage proporciona acceso a todos estos bloques de construcción, lo que permite inspeccionarlos para que el servidor pueda generar una respuesta adecuada.

La tabla 6 enumera las propiedades proporcionadas para los primeros tres bloques de construcción de la solicitud y explicaremos cómo manejar el cuerpo de la solicitud en el siguiente documento.

Los encabezados HTTP pueden ser difíciles de manejar y las propiedades headers y headersDistinct normalizan los encabezados para que sean más fáciles de usar. Algunos encabezados HTTP solo deben aparecer una vez en una solicitud, por lo que Node.js elimina los valores duplicados.

Otros encabezados pueden tener múltiples valores, y estos se concatenan en un único valor de cadena mediante la propiedad headers y en una matriz de cadenas mediante la propiedad headersDistinct. La excepción es el encabezado set-cookie, que siempre se presenta como un arreglo de cadenas. (Describiremos cómo se usan las cookies en detalle en la Parte 2).

Tabla 6: Propiedades útiles de IncomingMessage.

Nombre	Descripción
headers	Esta propiedad devuelve un objeto IncomingHttpHeaders, que define propiedades para encabezados comunes y también se puede utilizar como un objeto llave/valor que asigna los nombres de los encabezados en la solicitud a los valores de encabezado. Los encabezados están normalizados, como se describe a continuación.
headersDistinct	Esta propiedad devuelve un objeto llave/valor que asigna los nombres de los encabezados en la solicitud a los valores de encabezado. Los valores están normalizados, como se describe a continuación en esta tabla.
httpVersion	Esta propiedad devuelve un valor de cadena que contiene la versión de HTTP utilizada en la solicitud.
method	Esta propiedad devuelve un valor de cadena que contiene el método HTTP especificado por la solicitud. Este valor puede no estar definido.
url	Esta propiedad devuelve un valor de cadena que contiene la URL de la solicitud. Este valor puede no estar definido.
socket	Esta propiedad devuelve un objeto que representa el socket de red utilizado para recibir la conexión, lo que resulta útil al detectar solicitudes HTTPS, como se muestra en la sección Detección de solicitudes HTTPS.

Sugerencia

La clase IncomingRequest también define la propiedad rawHeaders, que proporciona acceso a los encabezados tal como se recibieron, sin normalización. Esta propiedad puede ser útil si necesitas realizar una normalización personalizada, pero las propiedades headers y headersDistinct son más útiles para proyectos de desarrollo convencionales.

Como regla general, la propiedad headers es más útil para mostrar o registrar encabezados, mientras que la propiedad headersDistinct es más útil cuando se usan encabezados para decidir qué tipo de respuesta producir. El listado 5 actualiza el ejemplo para registrar los detalles de la solicitud en la consola Node.js.

Listado 5: Registro de detalles de la solicitud en el archivo handler.ts en la carpeta src.

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

export const handler = async (req: IncomingMessage, resp: ServerResponse) => {
  console.log(`---- HTTP Method: ${req.method}, URL: ${req.url}`);
  console.log(`host: ${req.headers.host}`);
  console.log(`accept: ${req.headers.accept}`);
  console.log(`user-agent: ${req.headers["user-agent"]}`)

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

Este ejemplo escribe el método HTTP, la URL de la solicitud y tres encabezados: el encabezado host, que especifica el nombre de host y el puerto al que se envió la solicitud; el encabezado accept, que especifica los formatos que el cliente está dispuesto a aceptar en la respuesta; y el encabezado user-agent, que identifica al cliente.

Usamos la propiedad headers en el listado 5, que nos permite acceder a los encabezados usando propiedades que corresponden al nombre del encabezado, de esta manera:

```
...
console.log(`host: ${req.headers.host}`);
...
```

No todos los nombres de encabezado HTTP se pueden usar como nombres de propiedad de JavaScript, y no hay ninguna propiedad para el encabezado user-agent porque los nombres de propiedad de JavaScript no pueden contener guiones. En cambio, tenemos que acceder al encabezado del agente de usuario especificando el nombre de la propiedad como una cadena, de esta manera:

```
...
console.log(`user-agent: ${req.headers["user-agent"]}`)
...
```

Utiliza un navegador para solicitar http://localhost:5000 y verás un resultado similar al siguiente, aunque puedes ver diferentes valores para los encabezados (y hemos omitido los valores del encabezado para abreviar):

```
---- HTTP Method: GET, URL: /
host: localhost:5000
accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0
.8,application/signed-exchange;v=b3;q=0.7
user-agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/128.0.0.0 Safari/537.36
---- HTTP Method: GET, URL: /favicon.ico
host: localhost:5000
accept: image/avif,image/webp,image/apng,image/svg+xml,image/*,*/*;q=0.8
user-agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/128.0.0.0 Safari/537.36
```

Este resultado muestra dos solicitudes porque los navegadores a menudo solicitarán /favicon.ico, que se utiliza como el icono de la pestaña. Es posible que no veas la solicitud

favicon.ico si utilizaste recientemente tu navegador para los ejemplos del documento anterior, donde se produjo una respuesta 404 Not Found.
Puedes borrar la memoria caché de tu navegador si deseas ver ambas solicitudes, pero no es importante para los ejemplos que siguen.

Análisis (parsing) de URL

Node.js proporciona la clase URL en el módulo url para analizar las URL en sus partes, lo que facilita la inspección de las URL para tomar decisiones sobre qué tipo de respuesta se enviará. Las URL se analizan creando un nuevo objeto URL y leyendo las propiedades descritas en la tabla 7.

Tabla 7: Propiedades útiles de URL.

Nombre	Descripción
hostname	Esta propiedad devuelve una cadena que contiene el componente de nombre de host de la URL.
pathname	Esta propiedad devuelve una cadena que contiene el componente de nombre de ruta de la URL.
port	Esta propiedad devuelve una cadena que contiene el componente de puerto de la URL. El valor será una cadena vacía si la solicitud se ha realizado al puerto predeterminado para el protocolo de la URL (como el puerto 80 para solicitudes HTTP no seguras).
protocol	Esta propiedad devuelve una cadena que contiene el componente de protocolo de la URL.
search	Esta propiedad devuelve una cadena que contiene la parte de consulta completa de la URL.
searchParams	Esta propiedad devuelve un objeto URLSeachParams que proporciona acceso de llave/valor a la parte de consulta de la URL.

El listado 6 crea un nuevo objeto URL para analizar la URL de la solicitud.

Listado 6: Análisis de una URL en el archivo handler.ts en la carpeta src.

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

export const handler = async (req: IncomingMessage, resp: ServerResponse) => {
  console.log(`---- HTTP Method: ${req.method}, URL: ${req.url}`);
  // console.log(`host: ${req.headers.host}`);
  // console.log(`accept: ${req.headers.accept}`);
  // console.log(`user-agent: ${req.headers["user-agent"]}`)

  const parsedURL = new URL(req.url ?? "", `http://${req.headers.host}`);
  console.log(`protocol: ${parsedURL.protocol}`);
```

```
    console.log(`hostname: ${parsedURL.hostname}`);
    console.log(`port: ${parsedURL.port}`);
    console.log(`pathname: ${parsedURL.pathname}`);
    parsedURL.searchParams.forEach((val, key) => {
      console.log(`Search param: ${key}: ${val}`)
    });

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

La creación de un objeto URL para analizar una URL requiere un poco de trabajo. La propiedad IncomingMessage.url devuelve una URL relativa, que el constructor de la clase URL aceptará como argumento, pero solo si la parte base de la URL (el protocolo, el nombre de host y el puerto) se especifica como segundo argumento. El nombre de host y el puerto se pueden obtener del encabezado de solicitud del host, de esta manera:

```
...
const parsedURL = new URL(req.url ?? "", `http://${req.headers.host}`);
...
```

La parte que falta es el protocolo. El ejemplo solo acepta solicitudes HTTP no seguras regulares, por lo que podemos especificar http como protocolo, con la tranquilidad de saber que será correcto. Demostraremos cómo determinar el protocolo correctamente cuando demostremos el uso de HTTPS más adelante en el documento.

Las propiedades descritas en la tabla 7 se pueden utilizar para inspeccionar las partes individuales de la URL una vez que se ha creado el objeto URL, y el ejemplo escribe los valores de protocolo, nombre de host, puerto y ruta de acceso.

La clase URL analiza la sección de consulta de la URL y la presenta como un conjunto de pares llave/valor, que también se escriben. Utiliza un navegador para solicitar la siguiente URL: http://localhost:5000/ myrequest?first=Bob&last=Smith

Esta URL tiene una ruta y una consulta, y verás un resultado similar al siguiente cuando se analice la URL:

```
(Event) Server listening on port 5000
---- HTTP Method: GET, URL: /myrequest?first=Bob&last=Smith
protocol: http:
hostname: localhost
port: 5000
pathname: /myrequest
Search param: first: Bob
```

Search param: last: Smith
---- HTTP Method: GET, URL: /favicon.ico
protocol: http:
hostname: localhost
port: 5000
pathname: /favicon.ico

El resultado muestra que el navegador ha enviado una segunda solicitud, para /favicon.ico, además de la URL que se solicitó explícitamente.

Comprensión de las respuestas HTTP

El propósito de inspeccionar una solicitud HTTP es determinar qué tipo de respuesta se requiere.
Las respuestas se generan utilizando las características proporcionadas por la clase ServerResponse, las más útiles de las cuales se describen en la tabla 8.

Tabla 8: Miembros útiles de ServerResponse.

Nombre	Descripción
sendDate	Esta propiedad booleana determina si Node.js genera automáticamente el encabezado Date y lo agrega a la respuesta. El valor predeterminado es true.
setHeader(name, value)	Este método establece un encabezado de respuesta utilizando el nombre y el valor especificados
statusCode	Esta propiedad de número se utiliza para establecer el código de estado de respuesta.
statusMessage	Esta propiedad de cadena se utiliza para establecer el mensaje de estado de respuesta.
writeHead(code, msg, headers)	Este método se utiliza para establecer el código de estado y, opcionalmente, el mensaje de estado y los encabezados de respuesta.
write(data)	Este método escribe datos en el cuerpo de la respuesta, que se expresa como una cadena o un búfer. Este método acepta argumentos opcionales que especifican la codificación de los datos y una función de devolución de llamada que se invoca cuando se completa la operación.
end()	Este método le dice a Node.js que la respuesta está completa y se puede enviar al cliente. El método se puede invocar con un argumento de datos opcional, que se agregará al cuerpo de la respuesta, una codificación para los datos y una función de devolución de llamada que se invocará cuando se haya enviado la respuesta.

El enfoque básico para generar una respuesta es establecer el código de estado y el mensaje de estado, definir los encabezados que ayudarán al cliente a procesar la respuesta, escribir los datos para el cuerpo (si lo hay) y luego enviar la respuesta al cliente.

El listado 7 inspecciona las solicitudes que se reciben para determinar cómo se utilizan las características proporcionadas por la clase ServerResponse para crear una respuesta.

Listado 7: Generación de respuestas HTTP en el archivo handler.ts en la carpeta src.

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

export const handler = async (req: IncomingMessage, resp: ServerResponse) => {
  const parsedURL = new URL(req.url ?? "", `http://${req.headers.host}`);
  if (req.method !== "GET" || parsedURL.pathname == "/favicon.ico") {
    resp.writeHead(404, "Not Found");
    resp.end();
    return;
  } else {
    resp.writeHead(200, "OK");
    if (!parsedURL.searchParams.has("keyword")) {
      resp.write("Hello, HTTP");
    } else {
      resp.write(`Hello, ${parsedURL.searchParams.get("keyword")}`);
    }
    resp.end();
    return;
  }
};
```

Este ejemplo genera tres respuestas diferentes. Para las solicitudes que no especifican el método HTTP GET o request /favicon.ico, el código de estado se establece en 404, que le indica al navegador que el recurso solicitado no existe, el mensaje de estado legible para humanos se establece en Not Found y se llama al método final para completar la solicitud.

Para todas las demás solicitudes, el código de estado se establece en 200, lo que indica una respuesta exitosa y el mensaje de estado se establece en OK. Se comprueba el componente de consulta de la URL de solicitud para ver si hay un parámetro de palabra clave y, si lo hay, se incluye el valor en el cuerpo de la respuesta.

Observa que utilizamos la palabra clave return después de llamar al método end. Esto no es un requisito, pero es un error establecer encabezados o escribir datos después de que se haya llamado al método end, y regresar explícitamente desde la función evita este problema.

Utiliza un navegador para solicitar http://localhost:5000/favicon.ico, http://localhost:5000?keyword=World y http://localhost:5000 y verás las respuestas que se muestran en la figura 2. (El navegador generalmente solicita el archivo favicon.ico en segundo plano, pero solicitarlo explícitamente hace que sea más fácil ver la respuesta HTTP 404).

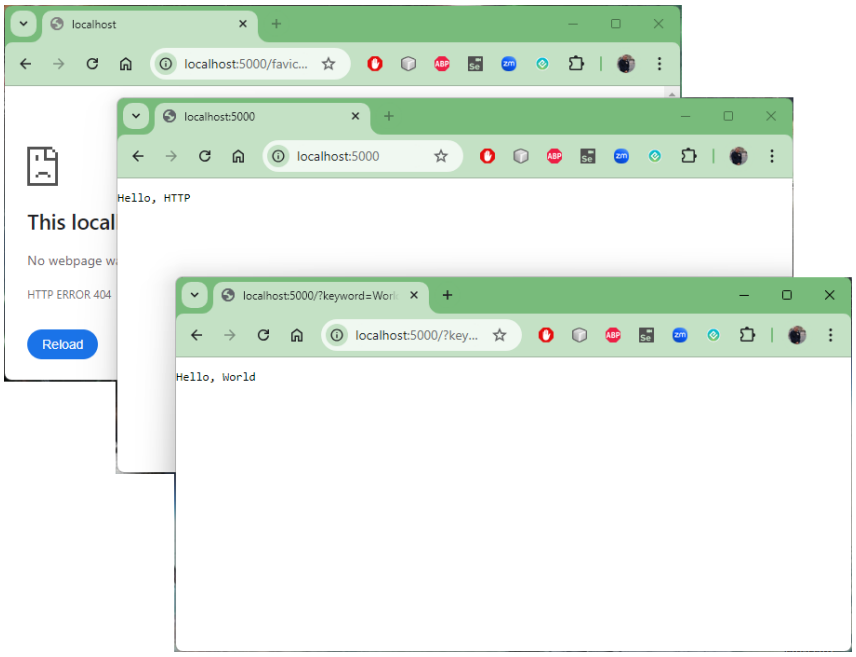

Figura 2: Generación de respuestas HTTP.

Compatibilidad con solicitudes HTTPS

La mayoría de las aplicaciones web utilizan HTTPS, donde las solicitudes HTTP se envían a través de una conexión de red cifrada mediante el protocolo TLS/SSL. El uso de HTTPS garantiza que la solicitud y la respuesta no se puedan inspeccionar mientras atraviesan redes públicas.

Para soportar (admitir) SSL se necesita un certificado que establezca la identidad del servidor y se utilice como base para el cifrado que protege las solicitudes HTTPS. En este documento, vamos a utilizar un certificado autofirmado, que es suficiente para el desarrollo y las pruebas, pero no se debe utilizar para la implementación.

Consulta https://letsencrypt.org si deseas un certificado para la implementación. El servicio Let's Encrypt cuenta con el respaldo de una organización sin fines de lucro y ofrece certificados gratuitos adecuados para su uso con HTTPS.

Creación del certificado autofirmado

La forma más sencilla de crear un certificado auto-firmado es utilizar el paquete OpenSSL, que es un kit de herramientas de código abierto para tareas relacionadas con la seguridad. El proyecto OpenSSL se puede encontrar en https://www.openssl.org y OpenSSL es parte de muchas distribuciones populares de Linux. Se puede encontrar una lista de binarios e instaladores, incluidos los instaladores para Windows, en:
https://wiki.openssl.org/index.php/binaries.

Alternativamente, el cliente Git incluye OpenSSL en la carpeta usr/bin (C:\Program Files\Git\usr\bin en Windows), que se puede usar para crear certificados autofirmados sin necesidad de instalar el paquete OpenSSL.

Asegúrate de que el ejecutable OpenSSL esté en la ruta del símbolo del sistema y ejecuta el comando que se muestra en el listado 8 en la carpeta webapp, ingresando el comando completo en una línea. Usa la terminal de símbolo del sistema, no la de Node.js

Listado 8: Generar un certificado autofirmado.

```
openssl req -x509 -newkey rsa:4096 -keyout key.pem -out cert.pem -sha256 -days 3650 -nodes
```

Este comando solicita los detalles que se incluirán en el certificado. Presione la tecla Enter para seleccionar el valor predeterminado para cada opción:

```
-----
Country Name (2 letter code) [AU]:
State or Province Name (full name) [Some-State]:
Locality Name (eg, city) []:
Organization Name (eg, company) [Internet Widgits Pty Ltd]:
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:
Email Address []:
```

Los detalles no importan porque el certificado se usará solo para el desarrollo. Cuando se complete el comando, habrá dos archivos nuevos en la carpeta webapp: el archivo cert.pem (que contiene el certificado autofirmado) y el archivo key.pem (que contiene la llave privada para el certificado).

Manejo de solicitudes HTTPS

El siguiente paso es utilizar la API proporcionada por Node.js para recibir solicitudes HTTPS, como se muestra en el listado 9.

Listado 9: Manejo de solicitudes HTTPS en el archivo server.ts en la carpeta src.

```
import { createServer } from "http";
import { handler } from "./handler";
import { createServer as createHttpsServer } from "https";
import { readFileSync } from "fs";

const port = 5000;
const https_port = 5500;

const server = createServer(handler);

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

const httpsConfig = {
  key: readFileSync("key.pem"),
  cert: readFileSync("cert.pem")
};

const httpsServer = createHttpsServer(httpsConfig, handler);

httpsServer.listen(https_port,
 () => console.log(`HTTPS Server listening on port ${https_port}`));
```

El proceso para recibir solicitudes HTTPS es similar al HTTP normal, en la medida en que la función para crear un servidor HTTPS se llama createServer, que es el mismo nombre que se usa para HTTP. Para usar ambas versiones de la función createServer en el mismo archivo de código, hemos usado un alias en la declaración de importación, como este:

```
...
import { createServer as createHttpsServer } from "https";
...
```

Esta declaración importa la función createServer del módulo https y la palabra clave *as* se usa para asignar un nombre que no entre en conflicto con otras importaciones. En este caso, el nombre que he elegido es createHttpsServer.

Se requiere un objeto de configuración para especificar los archivos de certificado que se crearon en la sección anterior con propiedades denominadas key y cert:

```
...
const httpsConfig = {
```

```
  key: readFileSync("key.pem"),
  cert: readFileSync("cert.pem")
};
..
```

Se pueden asignar valores de cadena o de Buffer a las propiedades key y cert. Utilizamos las funciones readFileSync del módulo fs para leer el contenido de los archivos key.pem y cert.pem, lo que produce valores de Buffer que contienen arreglos de bytes.

Comprensión de las lecturas de archivos síncronos

En el documento anterior, explicamos que puede tener sentido usar operaciones de bloqueo cuando sabes que no hay otro trabajo que realizar en el hilo principal. En este caso, necesitas leer el contenido de los archivos key.pem y cert.pem como parte del inicio de la aplicación. Hay poco beneficio en usar una devolución de llamada o una promesa porque necesitamos el contenido de esos archivos para configurar Node.js para que escuche solicitudes HTTPS y el uso de operaciones sin bloqueo produce un código como este:

```
...
readFile("key.pem", (err, keyBuffer) => {
  readFile("cert.pem", (err, certBuffer) => {
    const server = createServer(handler);

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

    const httpsServer = createHttpsServer({
      key: keyBuffer, cert: certBuffer
    }, handler);

    httpsServer.listen(https_port,
      () => console.log(
        `HTTPS Server listening on port ${https_port}`));
  });
});
...
```

Este código muestra que *puedes* leer los archivos usando la función readFile sin bloqueo, pero las devoluciones de llamada anidadas son más difíciles de entender. Las promesas tampoco ayudan porque la palabra clave await solo se puede usar dentro de funciones, lo que significa que se debes usar la sintaxis then demostrada en el tema anterior. Es importante evitar bloquear el hilo principal en casi todas las situaciones, pero hay algunas ocasiones en las que no importa y las funciones sin bloqueo son menos útiles.

Hay muchas opciones de configuración disponibles, descritas en https://nodejs.org/dist/latestv20.x/docs/api/https.html#httpscreateserveroptions-requestlistener, pero las opciones key y cert son suficientes para comenzar. El objeto de

configuración se pasa a la función createServer, a la que hemos dado el alias createHttpsServer en este ejemplo, y se llama al método listen en el resultado para comenzar a escuchar solicitudes HTTPS:

```
...
const httpsServer = createHttpsServer(httpsConfig, handler);

httpsServer.listen(https_port,
  () => console.log(`HTTPS Server listening on port ${https_port}`));
...
```

Abre un navegador web y solicita https://localhost:5500, que enviará una solicitud HTTPS al puerto en el que Node.js se ha configurado para escuchar. Los navegadores mostrarán advertencias para los certificados autofirmados y, por lo general, tendrás que confirmar que deseas continuar, como se muestra en la figura 5.3, que muestra la advertencia presentada por Chrome.

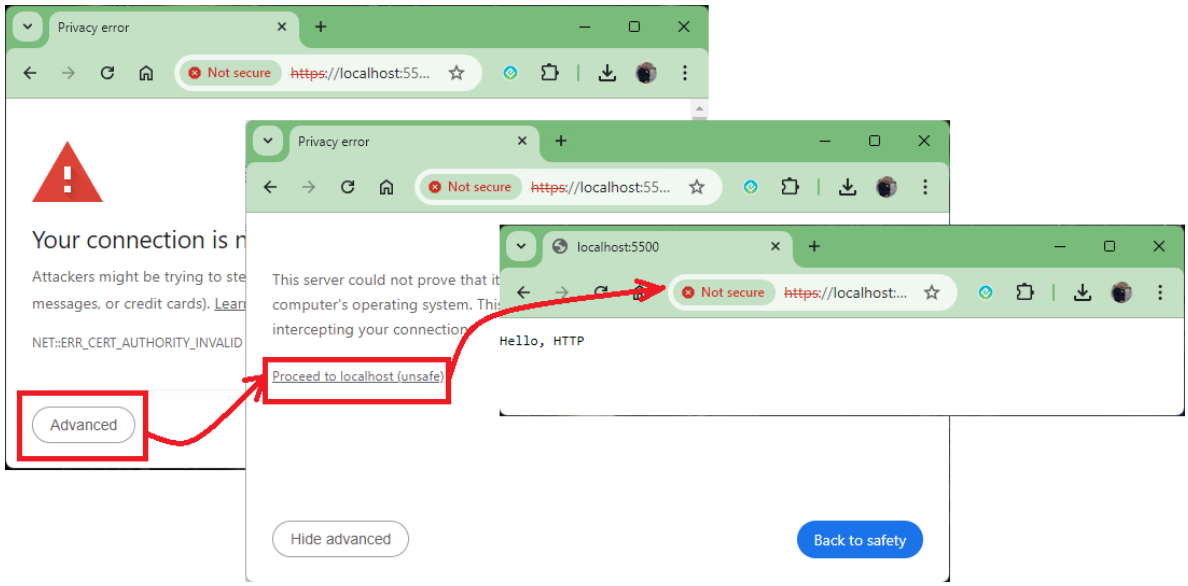

Figura 3: Aceptación de un certificado autofirmado.

Node.js sigue escuchando solicitudes HTTP regulares en el puerto 5000, lo cual puedes confirmar solicitando http://localhost:5000.

Detección de solicitudes HTTPS

La API de Node.js utiliza las clases IncomingMessage y ServerResponse para solicitudes HTTP y HTTPS, lo que significa que se puede utilizar la misma función de controlador para ambos tipos de solicitud. Sin embargo, puede ser útil saber qué tipo de solicitud se está

procesando para que se puedan generar diferentes respuestas, como se muestra en el listado 10.

Listado 10: Detección de solicitudes HTTPS en el archivo handler.ts en la carpeta src.

```
import { IncomingMessage, ServerResponse } from "http";
import { TLSSocket } from "tls";
import { URL } from "url";

export const isHttps = (req: IncomingMessage) : boolean => {
  return req.socket instanceof TLSSocket && req.socket.encrypted;
}

export const handler = (req: IncomingMessage, resp: ServerResponse) => {
  const protocol = isHttps(req) ? "https" : "http";

  const parsedURL =
    new URL(req.url ?? "", `${protocol}://${req.headers.host}`);
  if (req.method !== "GET" || parsedURL.pathname == "/favicon.ico") {
    resp.writeHead(404, "Not Found");
    resp.end();
    return;
  } else {
    resp.writeHead(200, "OK");
    if (!parsedURL.searchParams.has("keyword")) {
      resp.write(`Hello, ${protocol.toUpperCase()}`);
    } else {
      resp.write(`Hello, ${parsedURL.searchParams.get("keyword")}`);
    }
    resp.end();
    return;
  }
};
```

La propiedad socket definida por la clase IncomingMessage devolverá una instancia de la clase TLSSocket para solicitudes seguras y esta clase define una propiedad cifrada que siempre devuelve verdadero.

Verificar si esta propiedad existe permite identificar conexiones HTTPS y HTTP para que se puedan producir diferentes respuestas.

Un patrón de implementación común para Node.js es usar un proxy que recibe solicitudes HTTPS de los clientes y las distribuye a los servidores Node.js mediante HTTP simple. En esta situación, generalmente puedes verificar el encabezado de solicitud X-Forwarded-Proto,

que los servidores proxy usan para transmitir detalles del cifrado utilizado por el cliente. Checa https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-Forwarded-Proto para obtener más detalles.

Usa un navegador para solicitar http://localhost:5000 y https://localhost:5500 y verás las respuestas que se muestran en la figura 4.

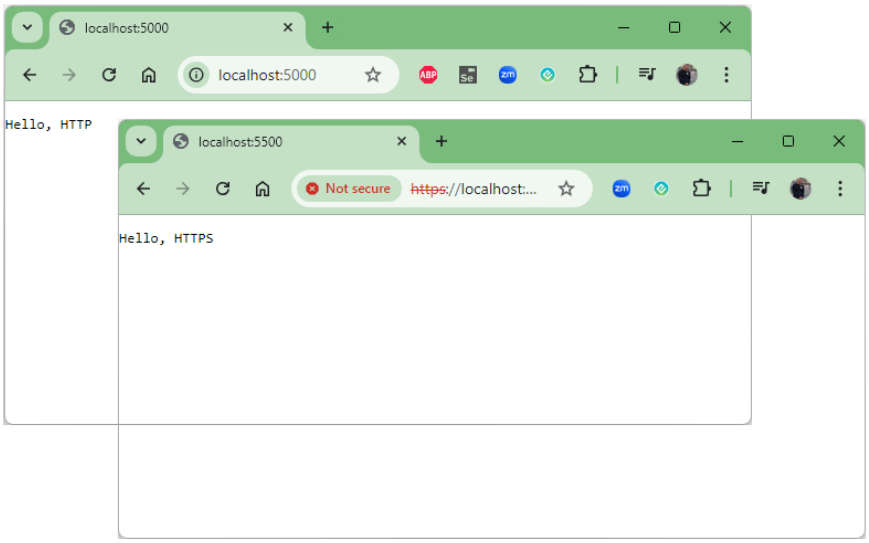

Figura 4: Identificación de solicitudes HTTPS.

Redireccionamiento de solicitudes inseguras

HTTPS se ha convertido en la forma preferida de ofrecer funcionalidad web y es una práctica común responder a las solicitudes HTTP regulares con una respuesta que indica al cliente que utilice HTTPS en su lugar, como se muestra en el listado 11.

Listado 11: Redireccionamiento de solicitudes HTTP en el archivo handler.ts en la carpeta src.

```
import { IncomingMessage, ServerResponse } from "http";
import { TLSSocket } from "tls";
import { URL } from "url";

export const isHttps = (req: IncomingMessage) : boolean => {
  return req.socket instanceof TLSSocket && req.socket.encrypted;
}

export const redirectionHandler
    = (req: IncomingMessage, resp: ServerResponse) => {
  resp.writeHead(302, {
    "Location": "https://localhost:5500"
```

```
  });
  resp.end();
}
export const handler = (req: IncomingMessage, resp: ServerResponse) => {
   const protocol = isHttps(req) ? "https" : "http";

  const parsedURL =
    new URL(req.url ?? "", `${protocol}://${req.headers.host}`);
  if (req.method !== "GET" || parsedURL.pathname == "/favicon.ico") {
    resp.writeHead(404, "Not Found");
    resp.end();
    return;
  } else {
    resp.writeHead(200, "OK");
    if (!parsedURL.searchParams.has("keyword")) {
      resp.write(`Hello, ${protocol.toUpperCase()}`);
    } else {
      resp.write(`Hello, ${parsedURL.searchParams.get("keyword")}`);
    }
    resp.end();
    return;
  }
};
```

El nuevo controlador utiliza el método writeHead para establecer el código de estado en 302, que denota una redirección, y establece el encabezado Location, que especifica la URL que el navegador debe solicitar en su lugar.

El listado 12 aplica el nuevo controlador para que se utilice para generar respuestas para todas las solicitudes HTTP.

Listado 12: Aplicación de un controlador en el archivo server.ts en la carpeta src.

```
import { createServer } from "http";
import { handler, redirectionHandler } from "./handler";
import { createServer as createHttpsServer } from "https";
import { readFileSync } from "fs";

const port = 5000;
const https_port = 5500;

const server = createServer(redirectionHandler);

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

```
const httpsConfig = {
  key: readFileSync("key.pem"),
  cert: readFileSync("cert.pem")
};

const httpsServer = createHttpsServer(httpsConfig, handler);

httpsServer.listen(https_port,
  () => console.log(`HTTPS Server listening on port ${https_port}`));
```

Si utilizas el navegador para solicitar http://localhost:5000, la respuesta enviada por el nuevo controlador hará que el navegador solicite https://localhost:5500. Si examinas las conexiones de red realizadas por el navegador en la ventana de herramientas para desarrolladores de F12, verás la respuesta de redirección y la solicitud HTTPS posterior, como se muestra en la figura 5.

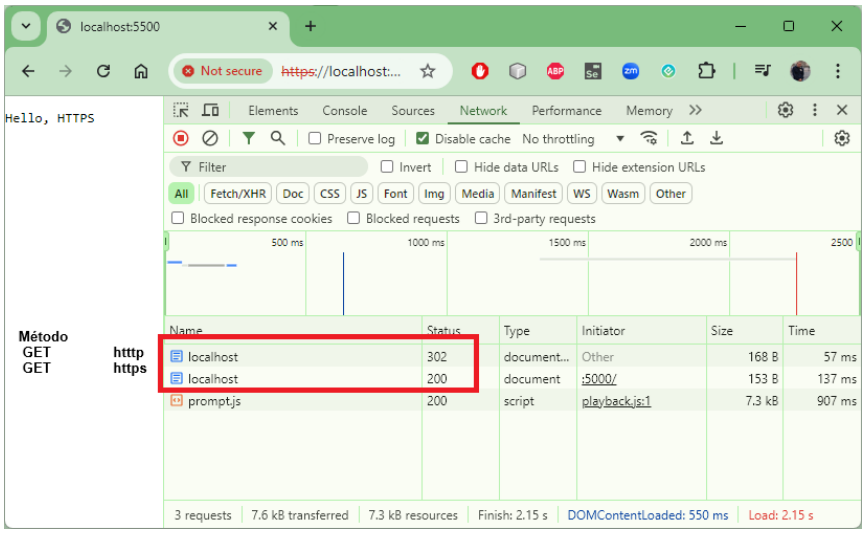

Figura 5: Redirección de solicitudes HTTP.

Uso de la Seguridad de transporte estricta HTTP (HSTS)

Redireccionar las solicitudes HTTP a una URL HTTPS significa que la comunicación inicial entre el cliente y el servidor no está cifrada, lo que presenta la posibilidad de que la solicitud HTTP sea secuestrada por un ataque de intermediario que redirija a los clientes a una URL maliciosa. El encabezado de Seguridad de transporte estricta HTTP (HSTS, HTTP Strict Transport Security) se puede utilizar para indicar a los navegadores que no utilicen únicamente solicitudes HTTPS para un dominio. Consulta: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Strict-TransportSecurity para obtener más detalles.

Entendiendo HTTP/2

Todos los ejemplos de este documento usan HTTP/1.1, que suele ser el protocolo predeterminado para el desarrollo de aplicaciones web Node.js.
HTTP/2 es una actualización del protocolo HTTP que tiene como objetivo mejorar el rendimiento.
HTTP/2 usa una única conexión de red para intercalar múltiples solicitudes del cliente, envía encabezados en un formato binario compacto y permite que el servidor "envíe" contenido al cliente antes de que lo solicite. Node.js brinda soporte para HTTP/2 en el módulo http2 e incluso incluye una API de compatibilidad que usa el enfoque que se muestra en este documento para manejar solicitudes HTTP/1.1 y HTTP/2 con el mismo código. (Consulta https://nodejs.org/dist/latest-v20.x/docs/api/http2.html para obtener más detalles).

Pero HTTP/2 no es una opción automática para los proyectos Node.js, aunque es más eficiente. Esto se debe a que HTTP/2 beneficia a las aplicaciones que tienen un gran volumen de solicitudes, y las aplicaciones de ese tamaño utilizan un proxy para recibir solicitudes y distribuirlas a múltiples servidores Node.js. El proxy recibe solicitudes HTTP/2 de los clientes, pero se comunica con Node.js mediante solicitudes HTTP/1.1 porque las características de HTTP/2 no tienen mucho impacto dentro del centro de datos.

Para las aplicaciones que no utilizan un proxy, el volumen de solicitudes es lo suficientemente pequeño como para que las eficiencias de HTTP/2 no justifiquen la complejidad adicional que HTTP/2 agrega al desarrollo, como requerir cifrado para todas las solicitudes.

La mayoría de las aplicaciones Node.js aún utilizan HTTP/1.1 y puedes ver esto reflejado en la forma en que los paquetes de código abierto para Node.js, como el paquete Express que usaremos en la siguiente sección, siguen siendo muy populares a pesar de que no admiten HTTP/2.

Uso de mejoras de terceros

La API que Node.js proporciona para HTTP y HTTPS es completa, pero puede producir código extenso que es difícil de leer y mantener. Una de las ventajas del desarrollo de JavaScript es la enorme variedad de paquetes de código abierto que están disponibles y hay muchos paquetes que se basan en la API de Node.js para simplificar el manejo de solicitudes.

El más popular de estos paquetes es Express. Ejecuta los comandos que se muestran en el listado 13 en la carpeta webapp para instalar el paquete Express y los tipos TypeScript para Express en el proyecto de ejemplo.

Listado 5.13: Instalación del paquete Express.

```
npm install express@4.18.2
npm install --save-dev @types/express@4.17.20
```

Si tuvieran como resultado lo siguiente: found 2 high severity vulnerabilities, les están haciendo una auditoria en la instalación, lo corrigen de la siguiente manera:

1. npm install -g npm@latest
2. npm cache clean –force
3. npm set audit false

Sugerencia

No te preocupes si no te gusta la forma en que funciona Express porque hay muchos otros paquetes disponibles que hacen cosas similares. Una búsqueda rápida en la web de alternativas a Express te dará varias opciones para considerar. Ten en cuenta al elegir un paquete que, como señalamos previamente, no todos los paquetes de JavaScript reciben soporte a largo plazo de sus creadores, y vale la pena considerar qué tan ampliamente se ha adoptado un paquete antes de usarlo en un proyecto.

Express tiene muchas funciones, que se describen en detalle en https://expressjs.com, pero las dos más útiles son el enrutador de solicitudes y los tipos de solicitud/respuesta mejorados, ambos descritos en las secciones siguientes.

Uso del enrutador Express

Las funciones del controlador de solicitudes que utilizan la API de Node.js combinan las instrucciones que inspeccionan las solicitudes con el código que genera las respuestas. Se requiere una nueva rama de código cada vez que la aplicación admite una nueva URL, como se muestra en el listado 14.

Listado 14: Admisión de una nueva URL en el archivo handler.ts en la carpeta src.

```
import { IncomingMessage, ServerResponse } from "http";
import { TLSSocket } from "tls";
import { URL } from "url";

export const isHttps = (req: IncomingMessage) : boolean => {
  return req.socket instanceof TLSSocket && req.socket.encrypted;
}

export const redirectionHandler
      = (req: IncomingMessage, resp: ServerResponse) => {
  resp.writeHead(302, {
    "Location": "https://localhost:5500"
  });
  resp.end();
}
```

```
export const handler = (req: IncomingMessage, resp: ServerResponse) => {
    const protocol = isHttps(req) ? "https" : "http";

  const parsedURL =
    new URL(req.url ?? "", `${protocol}://${req.headers.host}`);
  if (req.method !== "GET" || parsedURL.pathname == "/favicon.ico") {
    resp.writeHead(404, "Not Found");
    resp.end();
    return;
  } else {
    resp.writeHead(200, "OK");
    if (parsedURL.pathname == "/newurl") {
      resp.write("Hello, New URL");
    } else if (!parsedURL.searchParams.has("keyword")) {
      resp.write(`Hello, ${protocol.toUpperCase()}`);
    } else {
            resp.write(`Hello, ${parsedURL.searchParams.get("keyword")}`);
          }
    resp.end();
    return;
  }
};
```

Cada nueva incorporación hace que el código sea más complejo y aumenta las posibilidades de un error de codificación que no coincida con las solicitudes correctas o genere la respuesta incorrecta.

El enrutador Express resuelve este problema separando la coincidencia de solicitudes de la generación de respuestas.
El primer paso para utilizar el enrutador Express es refactorizar el código del controlador de solicitudes existente en funciones independientes que generen respuestas sin las instrucciones que inspeccionan las solicitudes, como se muestra en el listado 15.

Listado 15: Refactorización en el archivo handler.ts en la carpeta src.

```
import { IncomingMessage, ServerResponse } from "http";
import { TLSSocket } from "tls";
import { URL } from "url";

export const isHttps = (req: IncomingMessage) : boolean => {
  return req.socket instanceof TLSSocket && req.socket.encrypted;
}

export const redirectionHandler
```

```
    = (req: IncomingMessage, resp: ServerResponse) => {
  resp.writeHead(302, {
    "Location": "https://localhost:5500"
  });
  resp.end();
}

export const notFoundHandler
    = (req: IncomingMessage, resp: ServerResponse) => {
  resp.writeHead(404, "Not Found");
  resp.end();
}

export const newUrlHandler
    = (req: IncomingMessage, resp: ServerResponse) => {
  resp.writeHead(200, "OK");
  resp.write("Hello, New URL");
  resp.end();
}

export const defaultHandler
    = (req: IncomingMessage, resp: ServerResponse) => {
  resp.writeHead(200, "OK");
  const protocol = isHttps(req) ? "https" : "http";
  const parsedURL = new URL(req.url ?? "",
    `${protocol}://${req.headers.host}`);
  if (!parsedURL.searchParams.has("keyword")) {
    resp.write(`Hello, ${protocol.toUpperCase()}`);
  } else {
    resp.write(`Hello, ${parsedURL.searchParams.get("keyword")}`);
  }
  resp.end();
}
```

Las respuestas se generan de la misma manera que en los ejemplos anteriores, pero cada respuesta es creada por una función de controlador independiente, sin el código que coincide con las solicitudes.

El siguiente paso es utilizar el enrutador Express para hacer coincidir las solicitudes y seleccionar uno de los controladores del listado 15 para producir un resultado, como se muestra en el listado 16.

Listado 16: Uso del enrutador Express en el archivo server.ts en la carpeta src.

```
import { createServer } from "http";
import { redirectionHandler, newUrlHandler, defaultHandler,
        notFoundHandler } from "./handler";
import { createServer as createHttpsServer } from "https";
import { readFileSync } from "fs";
import express, { Express } from "express";

const port = 5000;
const https_port = 5500;

const server = createServer(redirectionHandler);

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

const httpsConfig = {
  key: readFileSync("key.pem"),
  cert: readFileSync("cert.pem")
};

const expressApp: Express = express();
expressApp.get("/favicon.ico", notFoundHandler);
expressApp.get("/newurl", newUrlHandler);
expressApp.get("*", defaultHandler);

const httpsServer = createHttpsServer(httpsConfig, expressApp);

httpsServer.listen(https_port,
  () => console.log(`HTTPS Server listening on port ${https_port}`));
```

El paquete Express contiene una exportación predeterminada, que es una función denominada express, y es por eso que la nueva declaración de importación se ve diferente:

```
...
import express, { Express } from "express";
...
```

La función express se invoca para crear un objeto Express, que proporciona métodos para asignar solicitudes a funciones de controlador. La tabla 9 describe los métodos más útiles, la mayoría de los cuales incorporan el método HTTP en el proceso de coincidencia.

En este documento, solo nos interesan las solicitudes GET, por lo que utilizamos el método get para especificar las rutas de URL y las funciones que generarán respuestas:

```
...
expressApp.get("/favicon.ico", notFoundHandler);
expressApp.get("/newurl", newUrlHandler);
expressApp.get("*", defaultHandler);
...
```

Tabla 9: Métodos Express útiles.

Nombre	Descripción
get(path, handler)	Este método enruta las solicitudes HTTP GET que coinciden con la ruta con la función del controlador (handler) especificado.
post(path, handler)	Este método enruta las solicitudes HTTP POST que coinciden con la ruta con la función del controlador especificado.
put(path, handler)	Este método enruta las solicitudes HTTP PUT que coinciden con la ruta con la función del controlador especificado.
delete(path, handler)	Este método enruta las solicitudes HTTP DELETE que coinciden con la ruta con la función del controlador especificado.
all(path, handler)	Este método enruta todas las solicitudes que coinciden con la ruta con la función del controlador especificado.
use(handler)	Este método agrega un componente de middleware, que puede inspeccionar e interceptar todas las solicitudes. Los temas posteriores contienen ejemplos que usan middleware.

Estas instrucciones son *rutas* y las URL se especifican como patrones que permiten comodines, como el carácter * en esta ruta:

```
...
expressApp.get("*", defaultHandler);
...
```

Esto coincide con cualquier solicitud GET y la enruta a la función defaultHandler. Express hace coincidir las solicitudes con las rutas en el orden en que están definidas, por lo que esta es una ruta general que se aplicará si las solicitudes no coinciden con las otras rutas.

Además de los métodos descritos en la tabla 9, el objeto Express también es una función de controlador que se puede utilizar con las funciones createServer de Node.js definidas en los módulos http y https:

```
...
const httpsServer = createHttpsServer(httpsConfig, expressApp);
...
```

Express procesa todas las solicitudes HTTP que Node.js recibe y las enruta al controlador adecuado.

Uso de las mejoras de solicitud y respuesta

Además del enrutamiento, Express proporciona mejoras a los objetos IncomingRequest y ServerResponse que se pasan a las funciones de controlador. El objeto que representa la solicitud HTTP se denomina Request y extiende el tipo IncomingRequest. Las mejoras más útiles de Request se describen en la tabla 10.

Tabla 10: Mejoras útiles de la solicitud Express.

Nombre	Descripción
hostname	Esta propiedad proporciona un acceso conveniente al valor del nombre de host.
params	Esta propiedad proporciona acceso a los parámetros de ruta, que se describen en la sección Uso de parámetros de ruta Express de este documento.
path	Esta propiedad devuelve el componente de ruta de la URL de la solicitud
protocol	Esta propiedad devuelve el protocolo utilizado para realizar la solicitud, que será http o https.
query	Esta propiedad devuelve un objeto cuyas propiedades corresponden a los parámetros de la cadena de consulta.
secure	Esta propiedad devuelve verdadero si la solicitud se realizó mediante HTTPS.
body	A esta propiedad se le asigna el contenido analizado del cuerpo de la solicitud, como se muestra más adelante.

El objeto que Express utiliza para representar la respuesta HTTP se denomina Response y extiende el tipo ServerResponse. Las mejoras básicas de Response más útiles se describen en la tabla 11.

Tabla 11: Mejoras básicas útiles de la respuesta de Express.

Nombre	Descripción
redirect(code, path) redirect(path)	Este método envía una respuesta de redirección. El argumento de código se utiliza para establecer el código de estado y el mensaje de respuesta. El argumento de ruta se utiliza para establecer el valor del encabezado Ubicación (Location). Si se omite el argumento de código, se envía una redirección temporal, con el código de estado 302.
send(data)	Este método se utiliza para enviar una respuesta al servidor. El código de estado se establece en 200. Este método establece encabezados de respuesta para describir el contenido, incluidos los encabezados Content-Length y Content-Type.
sendStatus(code)	Este método se utiliza para enviar una respuesta de código de estado y establecerá automáticamente el mensaje de estado para códigos de estado conocidos, de modo que un código de estado de 200 hará que se utilice el mensaje OK, por ejemplo.

Otras mejoras de Express se relacionan con funciones descritas en futuros documentos, pero las adiciones básicas descritas en la tabla 10 y la tabla 11 son suficientes para simplificar la forma en que la aplicación de ejemplo genera las respuestas, como se muestra en el listado 17.

Listado 17: Uso de las mejoras de Express en el archivo handler.ts en la carpeta src.

```
import { IncomingMessage, ServerResponse } from "http";
//import { TLSSocket } from "tls";
//import { URL } from "url";
import { Request, Response } from "express";

//export const isHttps = (req: IncomingMessage) : boolean => {
//   return req.socket instanceof TLSSocket && req.socket.encrypted;
//}

export const redirectionHandler
      = (req: IncomingMessage, resp: ServerResponse) => {
  resp.writeHead(302, {
     "Location": "https://localhost:5500"
  });
  resp.end();
}

export const notFoundHandler = (req: Request, resp: Response) => {
  resp.sendStatus(404);
}

export const newUrlHandler = (req: Request, resp: Response) => {
  resp.send("Hello, New URL");
}

export const defaultHandler = (req: Request, resp: Response) => {
  if (req.query.keyword) {
     resp.send(`Hello, ${req.query.keyword}`);
  } else {
     resp.send(`Hello, ${req.protocol.toUpperCase()}`);
  }
}
```

Express analiza automáticamente la URL de la solicitud y hace que sus partes sean accesibles a través de las propiedades de Response descritas en la tabla 5.10, lo que significa que no tienes que analizar la URL explícitamente. La conveniente propiedad secure significa que puedes eliminar la función isHttps.

Los métodos de Response descritos en la tabla 11 reducen la cantidad de declaraciones necesarias para producir respuestas. El método send, por ejemplo, se encarga de configurar el código de estado de la respuesta, establece algunos encabezados útiles y llama al método end para indicarle a Node.js que la respuesta está completa.

Si solicitas https://localhost:5500/newurl y https://localhost:5500?keyword=Express, verás las respuestas que se muestran en la figura 6.

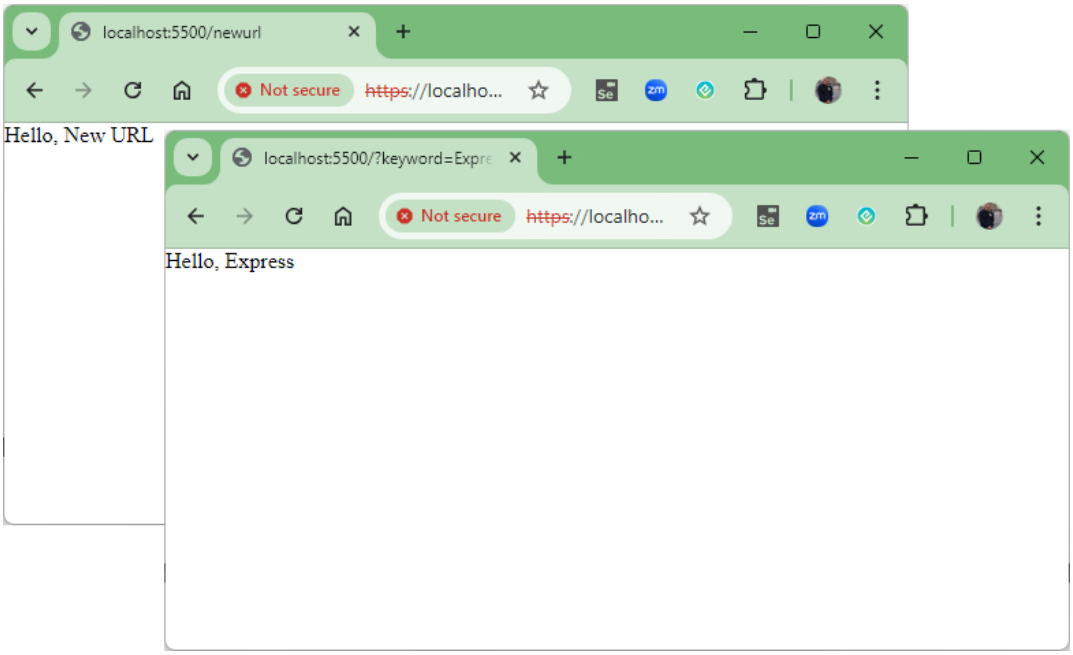

Figura 6: Generación de respuestas con Express.

Tu navegador puede usar una fuente diferente para mostrar estas respuestas, lo que sucede porque los métodos Response utilizados para generar respuestas en el listado 17 configuran el encabezado Content-Type en la respuesta como text/html. Este encabezado no se configuró en los ejemplos anteriores y altera la forma en que la mayoría de los navegadores muestran el contenido.

Uso de parámetros de ruta de Express

Es importante comprender que Express no hace nada mágico y sus características se basan en las proporcionadas por Node.js descritas anteriormente en este documento. El valor de Express es que hace que la API de Node.js sea más fácil de consumir, con el resultado de que el código es más fácil de entender y mantener.

Una característica especialmente útil que ofrece Express es la especificación de parámetros de ruta, que extraen valores de las rutas URL cuando coinciden con las solicitudes y los hacen

fácilmente accesibles a través de la propiedad Response.params, como se muestra en el listado 18.

Listado 18: Uso de parámetros de ruta en el archivo server.ts en la carpeta src.

```
...
const expressApp: Express = express();
expressApp.get("/favicon.ico", notFoundHandler);
expressApp.get("/newurl/:message?", newUrlHandler);
expressApp.get("*", defaultHandler);
...
```

La ruta modificada coincide con las solicitudes cuando la ruta comienza con /newurl. El segundo segmento en la ruta URL se asigna a un parámetro de ruta llamado message. El parámetro se denota con dos puntos (el carácter :).

Para la ruta URL /newurl/London, por ejemplo, al parámetro message se le asignará el valor London. El signo de interrogación (el carácter ?) indica que se trata de un parámetro opcional, lo que significa que la ruta coincidirá con las solicitudes incluso si no hay un segundo segmento de URL.

Los parámetros de ruta son una forma eficaz de aumentar el rango de URL con las que una ruta puede coincidir. El listado 19 utiliza la propiedad Response.params para obtener el valor del parámetro del mensaje e incorporarlo a la respuesta.

Listado 19: Consumo de un parámetro de ruta en el archivo handler.ts en la carpeta src.

```
...
export const newUrlHandler = (req: Request, resp: Response) => {
   const msg = req.params.message ?? "(No Message)";
   resp.send(`Hello, ${msg}`);
}
...
```

Utiliza un navegador para solicitar https://localhost:5500/newurl/London y verás la respuesta que se muestra en la figura 7.

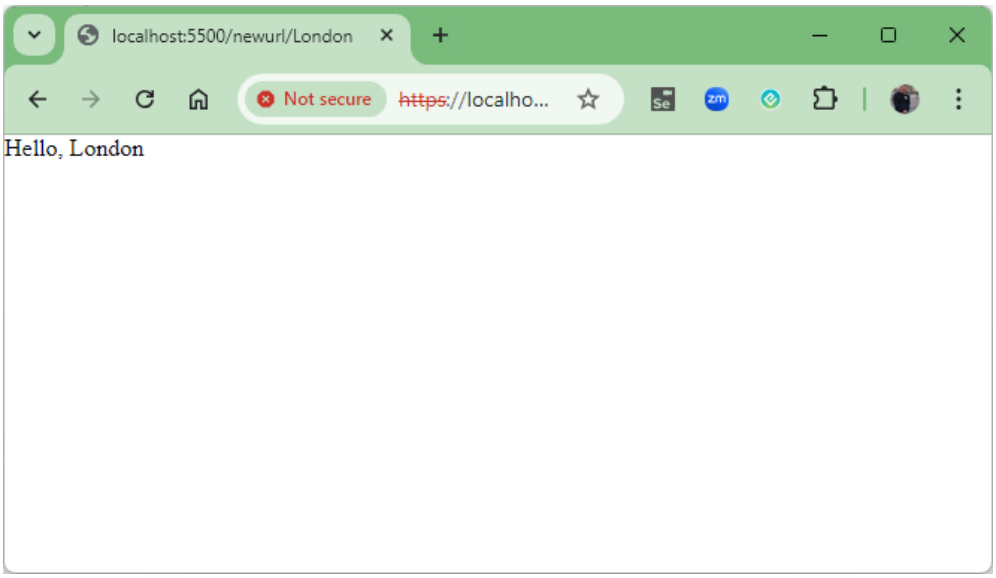

Figura 7: Uso de un parámetro de ruta.

Resumen

En este documento, describimos las características de la API que Node.js proporciona para recibir solicitudes HTTP y generar respuestas, que es la columna vertebral del desarrollo de aplicaciones web del lado del servidor:

- La API de Node.js proporciona soporte para recibir solicitudes HTTP y HTTPS.
- Node.js emite eventos cuando se reciben solicitudes e invoca funciones de devolución de llamada para manejar esas solicitudes.
- Por lo general, se requiere algún trabajo adicional, como analizar URL, cuando se utiliza la API de Node.js.
- Los paquetes de terceros, como Express, se basan en las API de Node.js para agilizar el procesamiento de solicitudes y simplificar el código que genera respuestas.

En el siguiente tema, describiremos las características que ofrece Node.js para leer y escribir datos.