

## Manejo de datos de formulario

En este documento, mostraremos las formas en que las aplicaciones Node.js pueden recibir datos de formulario y explicar las diferencias, incluido el soporte de archivos de carga. Esta parte del curso también explica cómo desinfectar datos de formulario para que pueda incluirse de forma segura en los documentos HTML y cómo validar los datos antes de que se usen. La tabla 1 pone este documento en contexto. La tabla 2 resume el documento.

Tabla 1: Poner los formularios HTML en contexto.

Pregunta	Respuesta
¿Qué son?	Los formularios HTML permiten a los usuarios proporcionar datos ingresando valores en campos de formulario.
¿Por qué son útiles?	Los formularios son las únicas formas en que los valores de datos se pueden recopilar de los usuarios de manera estructurada.
¿Cómo se usan?	Los documentos HTML contienen un elemento de formulario que contiene uno o más elementos que permiten ingresar datos, como un elemento de entrada.
¿Hay alguna trampa o limitaciones?	Los datos que se ingresan en un formulario deben desinfectarse antes de la inclusión en la salida HTML y validados antes de que las aplicaciones lo utilicen.
¿Hay alguna alternativa?	Los formularios son la única forma de solicitar eficientemente datos de los usuarios.

Tabla 2: Resumen del documento.

Problema	Solución	Listado
Recibir datos del usuario.	Usar un formulario HTML configurado para enviar datos al servidor.	1-10
Recibir datos utilizados para operaciones no idempotentes.	Configurar el formulario para usar solicitudes de publicación HTTP.	11, 12
Recibir datos complejos, incluido el contenido de los archivos.	Usar la codificación de forma multipart.	13-16
Evitar que los datos del usuario se interpreten como elementos HTML.	Desinfectar los datos recibidos del usuario.	17-21
Asegurarse de que la aplicación reciba datos útiles.	Validar los datos recibidos del usuario.	22-27, 30-32
Proporcionar comentarios de validación inmediata al usuario.	Validar los datos en el navegador antes de enviar el formulario.	28-29

## Preparándose para este documento

Esta parte del curso utiliza el proyecto Part2App del documento anterior. Ejecuta los comandos que se muestran en el listado 1 en la carpeta Parte2App para eliminar archivos que ya no son necesarios.

Listado 1: Eliminar archivos.

```
PS C:\proyectosnode\part2app> rm ./templates/**/*.handlebars
PS C:\proyectosnode\part2app> rm ./templates/**/*.custom
PS C:\proyectosnode\part2app> rm ./src/client/*_custom.js
PS C:\proyectosnode\part2app> rm ./src/server/*custom*.ts
PS C:\proyectosnode\part2app>
```

A continuación, reemplaza el contenido del archivo client.js en la carpeta src/client con el contenido que se muestra en el listado 2.

Listado 2: El contenido del archivo client.js en la carpeta src/client.

```
document.addEventListener('DOMContentLoaded', () => {
  // do nothing
});
```

Este es un marcador de posición hasta más adelante en el documento, cuando se necesitará el código del lado del cliente nuevamente. Reemplaza el contenido del archivo index.html en la carpeta static con los elementos que se muestran en el listado 3.

Listado 3: El contenido del 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>
    <form>
      <div class="m-2">
        <label class="form-label">Name</label>
        <input name="name" class="form-control" />
      </div>
      <div class="m-2">
        <label class="form-label">City</label>
        <input name="city" class="form-control" />
      </div>
    </form>
```

```
</body>
</html>
```

El documento HTML contiene un formulario HTML simple que le pide al usuario su nombre y ciudad. Para mantener el código que maneja los formularios separados del resto de la aplicación, agrega un archivo llamado form.ts a la carpeta src/server con el contenido que se muestra en el listado 4. No necesitas mantener el código de formularios separado; Solo lo hemos hecho para hacer que los ejemplos sean más fáciles de seguir.

Listado 4: El contenido del archivo forms.ts en la carpeta src/server.

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

export const registerFormMiddleware = (app: Express) => {
  // no middleware yet
}

export const registerFormRoutes = (app: Express) => {
  // no routes yet
}
```

El listado 5 actualiza el servidor para usar las funciones definidas en el listado 4.

Listado 5: Configuración del servidor en el archivo server.ts en la carpeta src/server.

```
import { createServer } from "http";
import express, { Express } from "express";
import { testHandler } from "../testHandler";
import httpProxy from "http-proxy";
import helmet from "helmet";
import { engine } from "express-handlebars";
import * as helpers from "../template_helpers";
import { registerFormMiddleware, registerFormRoutes } from "../forms";

const port = 5000;

const expressApp: Express = express();

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

expressApp.set("views", "templates/server");

expressApp.engine("handlebars", engine());
```

```

expressApp.set("view engine", "handlebars");

expressApp.use(helmet());
expressApp.use(express.json());
registerFormMiddleware(expressApp);
registerFormRoutes(expressApp);

expressApp.get("/dynamic/:file", (req, resp) => {
  resp.render(`${req.params.file}.handlebars`,
    { message: "Hello template", req,
      helpers: { ...helpers }
    });
});

expressApp.post("/test", testHandler);
expressApp.use(express.static("static"));
expressApp.use(express.static("node_modules/bootstrap/dist"));
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 listado 6 **elimina un ayudante del diseño utilizado por las plantillas del lado del servidor y agrega un elemento de script para el paquete JavaScript creado por Webpack**. Algunos ejemplos en este documento se basan en plantillas, y eliminar el ayudante simplifica la representación de plantillas, mientras que agregar el elemento de script permitirá que el código del lado del cliente se use en el contenido generado a partir de las plantillas.

Listado 6: Cambiar elementos en el archivo main.handlebars en la carpeta templates/server/layouts.

```

<!DOCTYPE html>
<html>
  <head>
    <script src="/bundle.js"></script>
    <link href="css/bootstrap.min.css" rel="stylesheet" />
  </head>
  <body>
    {{{ body }}}
  </body>
</html>

```

Finalmente, crea un archivo llamado `data.json` en la carpeta `part2app` con el contenido que se muestra en el listado 7. Este archivo se usará para demostrar cómo se pueden usar formularios para enviar archivos al servidor.

Listado 7: El contenido del archivo `data.json` en la carpeta `part2app`.

```
[  
  { "city": "London", "population": 8982000 },  
  { "city": "Paris", "population": 2161000 },  
  { "city": "Beijing", "population": 21540000 }  
]
```

Ejecuta el comando que se muestra en el listado 8 en la carpeta `part2App` para iniciar las herramientas de desarrollo y comenzar a escuchar las solicitudes HTTP.

Listado 11.8: Inicio de las herramientas de desarrollo.

```
npm start
```

Abre un navegador web y solicita `http://localhost:5000`. Verás los elementos del formulario definidos en el listado 3, cuya apariencia se ha diseñado utilizando el paquete Bootstrap CSS, como se muestra en la figura 1.

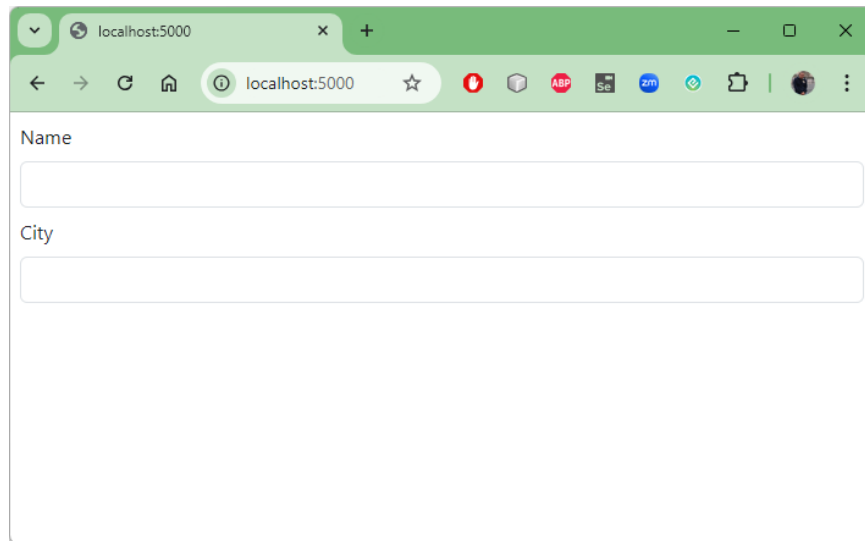
A screenshot of a web browser window. The address bar shows 'localhost:5000'. The page content consists of two text input fields. The first field is labeled 'Name' and the second field is labeled 'City'. Both fields are empty and have a light gray border. The browser's developer tools are open at the bottom, showing various icons for debugging.

Figura 1: Ejecutando la aplicación de ejemplo.

## Recibir datos del formulario

Los datos del formulario se pueden enviar utilizando solicitudes HTTP GET o POST y la elección del método determina cómo se presentan los datos contenidos en el formulario. El

Listado 9 completa el formulario para especificar la URL a la que se enviarán los datos del formulario y agrega botones que envían los datos del formulario con diferentes métodos HTTP.

Listado 9: Completar el formulario 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>
    <form action="/form">
      <div class="m-2">
        <label class="form-label">Name</label>
        <input name="name" class="form-control" />
      </div>
      <div class="m-2">
        <label class="form-label">City</label>
        <input name="city" class="form-control" />
      </div>
      <div class="m-2">
        <button class="btn btn-primary" formmethod="get">
          Submit (GET)
        </button>
        <button class="btn btn-primary" formmethod="post">
          Submit (POST)
        </button>
      </div>
    </form>
  </body>
</html>
```

El elemento de atributo de acción en el elemento de formulario le dice al navegador que envíe los datos del formulario al URL /form. Los elementos del botón están configurados con el atributo formmethod, que especifica qué método HTTP debe usar el navegador.

---

### Nota

Estamos utilizando atributos aplicados a los elementos del botón para que los mismos datos del formulario se procesen de diferentes maneras. En ejemplos posteriores, adoptaremos un enfoque más convencional y usaremos atributos aplicados al elemento del formulario.

---

## Recibir datos de formulario de las solicitudes GET

Las solicitudes GET son la forma más simple de recibir datos del formulario porque el navegador incluye los nombres y valores de campo del formulario en la cadena de consulta de URL. El listado 10 define un controlador para las solicitudes de obtener el formulario.

Listado 10: Manejo de solicitudes GET en el archivo forms.ts en la carpeta src/server.

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

export const registerFormMiddleware = (app: Express) => {
  // no middleware yet
}

export const registerFormRoutes = (app: Express) => {
  app.get("/form", (req, resp) => {

    for (const key in req.query) {
      resp.write(`${key}: ${req.query[key]}\n`);
    }
    resp.end();
  });
}
```

La ruta utiliza el método GET para que coincida con las solicitudes GET enviadas a la URL /form. Express decodifica cadenas de consulta de URL y las presenta a través de la propiedad Request.query.

En el listado 10, los parámetros y valores de la cadena de consulta se utilizan para generar la respuesta. Usa un navegador para solicitar <http://localhost:5000>, completa el formulario con Alice Smith como el nombre y London como la ciudad, y haz clic en el botón Submit (Get).

El navegador enviará una solicitud GET a la URL /formulario e incluirá los valores que se ingresaron en el formulario, como este:

Los datos serán recibidos por el servidor, la cadena de consulta se analizará y los datos del formulario se utilizarán en la respuesta, como se muestra en la figura 2.

La limitación de las solicitudes GET es que deben ser *idempotentes*, lo que significa que cada solicitud de URL dada siempre debe tener el mismo efecto y siempre devolver el mismo resultado. Dicho de otra manera, los datos del formulario enviados con una solicitud GET son efectivamente una solicitud para leer datos que no se espera que cambien con cada solicitud.

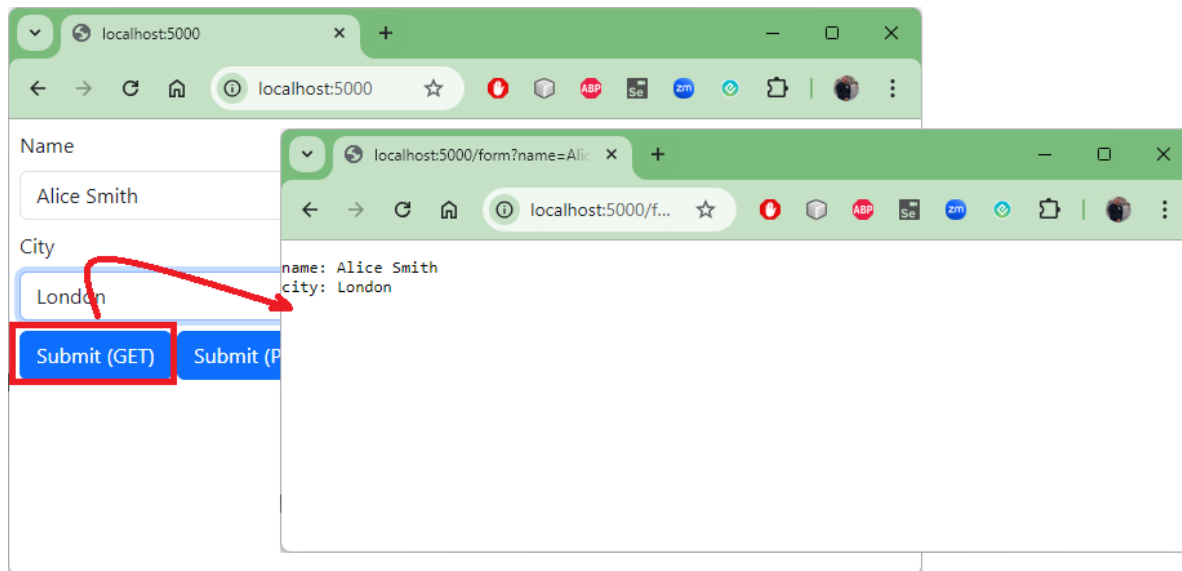


Figura 2: Manejo de datos de formulario de una solicitud GET.

Esto es importante porque los cachés HTTP pueden almacenar las respuestas para obtener solicitudes y usarlas para responder a las solicitudes de la misma URL, lo que significa que algunas solicitudes pueden no recibir el servidor de backend. Por esta razón, la mayoría de los datos de formulario se envían utilizando solicitudes de publicación, que no se almacenarán en caché, pero que pueden ser más complejas de procesar.

### Recibir datos de formulario de las solicitudes POST

Las solicitudes de publicación HTTP incluyen los datos del formulario en el cuerpo de la solicitud, que deben leerse y decodificarse antes de que puedan usarse. El listado 11 agrega una ruta que maneja las solicitudes POST, lee el cuerpo y la usa como respuesta.

Listado 11: Agregar un controlador al archivo form.ts en la carpeta src/server.

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

export const registerFormMiddleware = (app: Express) => {
  // no middleware yet
}

export const registerFormRoutes = (app: Express) => {
  app.get("/form", (req, resp) => {

    for (const key in req.query) {
      resp.write(`${key}: ${req.query[key]}\n`);
    }
    resp.end();
  });
}
```



```

});

app.post("/form", (req, resp) => {
  resp.write(`Content-Type: ${req.headers["content-type"]}\n`)
  req.pipe(resp);
});
}

```

Node.js y Express leen los encabezados de la solicitud HTTP y dejan el cuerpo para que pueda leerse como un flujo. La nueva ruta en el listado 11 coincide con las solicitudes POST enviadas a /form y crean una respuesta que contiene el encabezado de tipo de contenido de la solicitud y el cuerpo de la solicitud.

Usa un navegador para solicitar `http://localhost: 5000`, completa el formulario con los mismos detalles que en la sección anterior y haz clic en el botón Submit (POST). El navegador enviará una solicitud de publicación al servidor con los datos del formulario en el cuerpo de solicitud, produciendo la respuesta que se muestra en la figura 3.

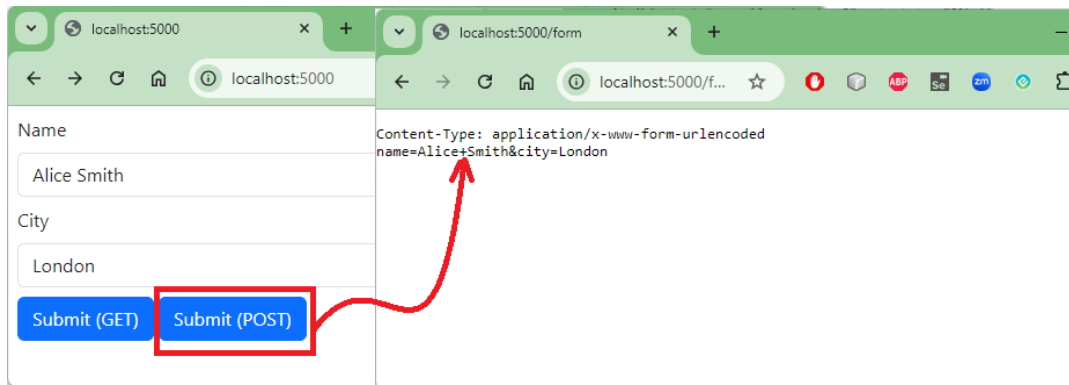


Figura 3: Manejo de datos de formulario de una solicitud posterior.

El navegador ha establecido el encabezado de tipo de contenido en `application/x-www-form-urlencoded`, que indica que los valores de datos del formulario están codificados de la misma manera que cuando los datos están incluidos en la cadena de consulta, con pares de valor de nombre separados por caracteres `=` y combinado con caracteres `&`, como este:

```

...
name=Alice+Smith&city=London
...

```

Puedes decodificar los datos del formulario tú mismo, pero Express incluye middleware que detecta el encabezado Content-Type y decodifica los datos del formulario en un mapa de

**llave/valor.** El listado 12 permite el middleware y utiliza los datos que produce en la respuesta.

Listado 12: Uso del middleware Express en el archivo forms.ts en la carpeta src/server.

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

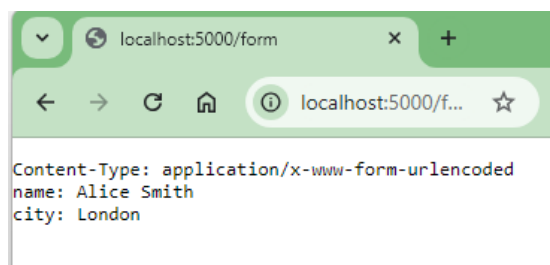
export const registerFormMiddleware = (app: Express) => {
  app.use(express.urlencoded({extended: true}))
}

export const registerFormRoutes = (app: Express) => {
  app.get("/form", (req, resp) => {

    for (const key in req.query) {
      resp.write(`${key}: ${req.query[key]}\n`);
    }
    resp.end();
  });

  app.post("/form", (req, resp) => {
    resp.write(`Content-Type: ${req.headers["content-type"]}\n`)
    for (const key in req.body) {
      resp.write(`${key}: ${req.body[key]}\n`);
    }
    resp.end();
  });
}
```

El componente del middleware se crea utilizando el método `Express.urlencoded` y la opción de configuración extendida requerida se usa para especificar si los cuerpos de solicitud se procesan utilizando la misma biblioteca que analiza las cadenas de consulta o, como aquí, una opción más sofisticada que permite tipos de datos más complejos a ser procesados. Para ver los datos decodificados, solicita `http://localhost:5000`, completa el formulario y haz clic en el botón Submit (POST). Los nombres y valores de los elementos de forma individuales se mostrarán en la respuesta, en lugar de la cadena codificada por URL, como esta:



## Recibir datos multipart

El formato de `application/x-www-form-urlencoded` es el valor predeterminado y funciona bien para recopilar valores de datos básicos de un usuario. Para formularios donde el usuario envía archivos, se usa el formato `multipart/form-data`, que es más complejo, pero permite que se envíe una combinación de tipos de datos en el cuerpo de la solicitud HTTP. El listado 13 agrega un elemento de entrada que permite al usuario seleccionar un archivo y un botón en el formulario HTML que envía los datos utilizando el formato `multipart/form-data`.

Listado 13: Agregar elementos 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>
    <form action="/form">
      <div class="m-2">
        <label class="form-label">Name</label>
        <input name="name" class="form-control" />
      </div>
      <div class="m-2">
        <label class="form-label">City</label>
        <input name="city" class="form-control" />
      </div>
      <div class="m-2">
        <label class="form-label">File</label>
        <input name="datafile" type="file" class="form-control" />
      </div>
      <div class="m-2">
        <button class="btn btn-primary" formmethod="get">
          Submit (GET)
        </button>
        <button class="btn btn-primary" formmethod="post">
          Submit (POST)
        </button>
        <button class="btn btn-primary" formmethod="post"
          formenctype="multipart/form-data">
          Submit (POST/MIME)
        </button>
      </div>
    </form>
```

```
</body>
</html>
```

El nuevo elemento de entrada tiene un atributo de tipo establecido en el archivo (set to file), que le dice al navegador que debe presentar al usuario un elemento para elegir un archivo.

El listado 14 actualiza el controlador del formulario para que las solicitudes de aplicación de application/x-www-form-urlencoded y multipart/form-data se manejen de manera diferente, lo cual es importante porque afecta la forma en que los navegadores tratan con los archivos.

Listado 14: Seleccionar el tipo de contenido en el archivo forms.ts en la carpeta src/server.

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

export const registerFormMiddleware = (app: Express) => {
  app.use(express.urlencoded({ extended: true }))
}

export const registerFormRoutes = (app: Express) => {
  app.get("/form", (req, resp) => {

    for (const key in req.query) {
      resp.write(`${key}: ${req.query[key]}\n`);
    }
    resp.end();
  });

  app.post("/form", (req, resp) => {
    resp.write(`Content-Type: ${req.headers["content-type"]}\n`)
    if (req.headers["content-type"]?.startsWith("multipart/form-data")) {
      req.pipe(resp);
    } else {
      for (const key in req.body) {
        resp.write(`${key}: ${req.body[key]}\n`);
      }
      resp.end();
    }
  });
}
```

Usa un navegador para solicitar <http://localhost:5000> y completa el formulario, eligiendo el archivo data.json creado al inicio del documento para el campo de archivo. La codificación del formulario determina cómo el navegador trata con los archivos. Haz clic en Submit (POST) para enviar el formulario con una solicitud POST en la codificación application/x-www-form-urlencoded y en el botón Submit (POST/MIME) para enviar el formulario con

una solicitud POST utilizando la codificación multipart/form-data. Ambos resultados se muestran en la figura 4.

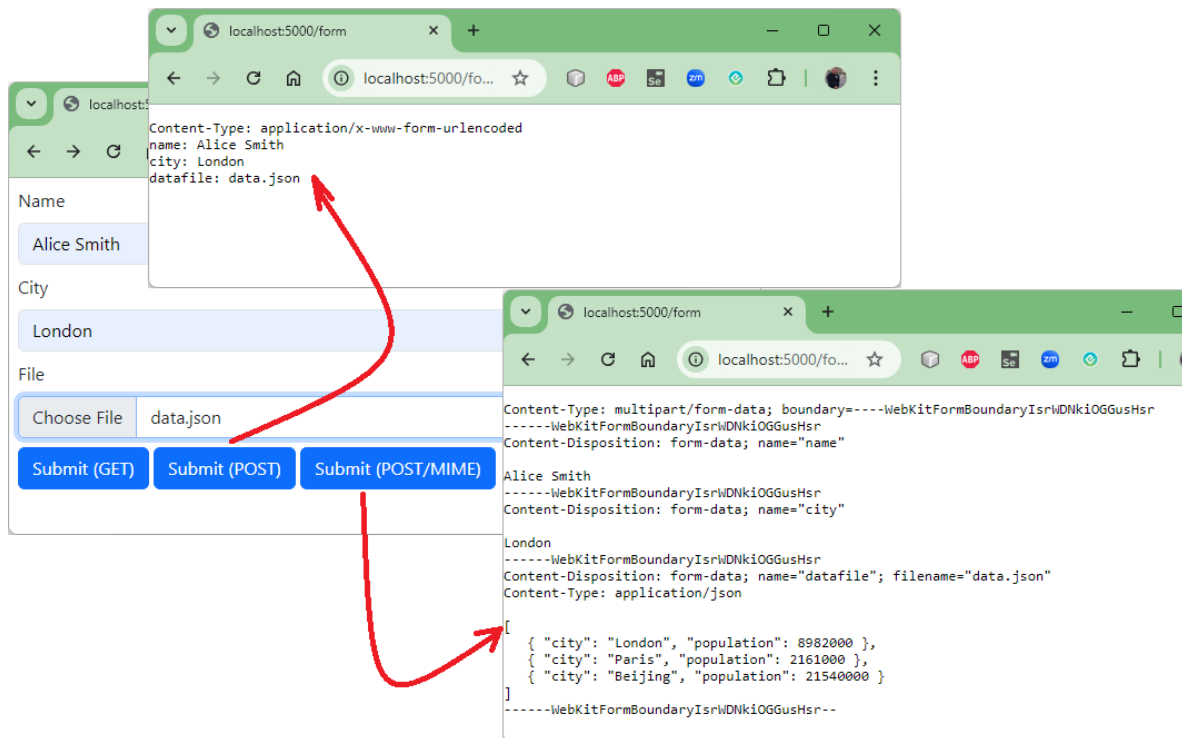


Figura 4: Enviar datos del formulario en diferentes codificaciones.

Para la codificación de application/x-www-form-urlencoded, el navegador incluye solo el nombre del archivo, así:

```
...
Content-Type: application/x-www-form-urlencoded
name: Alice
city: London
datafile: data.json
...
```

La codificación multipart/form-data incluye el contenido del archivo, pero para hacerlo, la estructura del cuerpo de solicitud se vuelve más compleja, como esta:

```
...
Content-Type: multipart/form-data; boundary=----WebKitFormBoundary41AOY4gvNpCTJzUy
-----WebKitFormBoundary41AOY4gvNpCTJzUy
Content-Disposition: form-data; name="name"
Alice
```

```

-----WebKitFormBoundary41AOY4gvNpCTJzUy
Content-Disposition: form-data; name="city"
London
-----WebKitFormBoundary41AOY4gvNpCTJzUy
Content-Disposition: form-data; name="datafile"; filename="data.json"
Content-Type: application/json
[
  { "city": "London", "population": 8982000 },
  { "city": "Paris", "population": 2161000 },
  { "city": "Beijing", "population": 21540000 }
]
-----WebKitFormBoundary41AOY4gvNpCTJzUy--
...

```

El cuerpo de solicitud contiene múltiples partes, cada una de las cuales está separada por una cadena límite, que se incluye en el encabezado de tipo de contenido:

```

...
Content-Type: multipart/form-data; boundary=-----WebKitFormBoundary41AOY4gvNpCTJzUy
...

```

Cada parte del cuerpo puede contener un tipo diferente de datos y viene completo con encabezados que describen el contenido. En el caso de la parte del cuerpo para el archivo, los encabezados proporcionan el nombre dado al campo form, el nombre del archivo que se ha elegido y el tipo de contenido en el archivo:

```

...
Content-Disposition: form-data; name="datafile"; filename="data.json"
Content-Type: application/json
...

```

La codificación de multipart/form-data se puede decodificar manualmente, pero no es una buena idea porque ha habido tantas implementaciones no conformes a lo largo de los años que requieren un manejo o soluciones especiales. Express no incluye soporte incorporado para procesar solicitudes multipart/form-data, pero varios paquetes de JavaScript pueden hacerlo.

Una opción es Multer (<https://github.com/expressjs/multer> que funciona bien con express. Ejecuta los comandos que se muestran en el listado 15 para instalar el paquete Multer y las definiciones de tipo que describen la API que proporciona a TypeScript.

Listado 15: Instalación de un paquete.

```
npm install multer@1.4.5-lts.1
npm install --save-dev @types/multer@1.4.11
```

El listado 16 configura el paquete Multer y lo aplica al controlador de formulario.

Listado 11.16: procesar solicitudes multipart en el archivo form.ts en la carpeta src/server.

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

const fileMiddleware = multer({ storage: multer.memoryStorage() });

export const registerFormMiddleware = (app: Express) => {
  app.use(express.urlencoded({ extended: true }));
}

export const registerFormRoutes = (app: Express) => {
  app.get("/form", (req, resp) => {

    for (const key in req.query) {
      resp.write(`${key}: ${req.query[key]}\n`);
    }
    resp.end();
  });

  app.post("/form", fileMiddleware.single("datafile"), (req, resp) => {
    resp.write(`Content-Type: ${req.headers["content-type"]}\n`);

    for (const key in req.body) {
      resp.write(`${key}: ${req.body[key]}\n`);
    }
    if (req.file) {
      resp.write(`---\nFile: ${req.file.originalname}\n`);
      resp.write(req.file.buffer.toString());
    }

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

Antes de que se pueda usar Multer, debes decirle dónde puede almacenar los archivos que recibe. El paquete viene con dos opciones de almacenamiento, que son escribir los archivos en una carpeta de disco o almacenar los datos del archivo en la memoria. Como se menciona

en la parte 1 del curso, se debe tener cuidado al escribir en el sistema de archivos y debe evitarse tanto como sea posible. Si necesitas almacenar datos de los usuarios, entonces mi consejo es usar una base de datos.

El listado 16 utiliza la opción de almacenamiento basada en memoria para crear un componente de middleware que procesará las solicitudes multipart/form-data. A diferencia de la mayoría de los otros middleware, el paquete Multer se aplica a rutas específicas para evitar que los usuarios maliciosos carguen archivos en rutas donde no se esperan:

```
...  
app.post("/form", fileMiddleware.single("datafile"), (req, resp) => {  
...  
}
```

Esta declaración aplica el middleware Multer en una sola ruta y busca archivos en un campo llamado datafile, que coincide con el atributo de nombre del elemento de entrada del archivo en el formulario HTML.

El middleware lee el cuerpo de solicitud y crea una propiedad de archivo a través de la cual se pueden leer los detalles del archivo cargado, con las propiedades más útiles descritas en la tabla 3. Las partes del cuerpo que no son archivos se presentarán a través de la propiedad del cuerpo.

Tabla 3: Propiedades útiles de descripción del archivo.

Nombre	Descripción
originalname	Esta propiedad devuelve el nombre del archivo en el sistema del usuario.
size	Esta propiedad devuelve el tamaño del archivo en bytes.
mimetype	Esta propiedad devuelve el tipo MIME del archivo.
buffer	Esta propiedad devuelve un búfer que contiene el archivo completo.

Para ver el efecto del middleware, solicita `http://localhost:5000`, completa el nombre y los campos de formulario de la ciudad, selecciona el archivo `data.json` y haz clic en el botón Submit (POST/MIME). La respuesta incluye los valores del cuerpo y las propiedades del archivo, como se muestra en la figura 5.

## Desinfección de datos del formulario

No solo debes ser cauteloso para recibir de los usuarios: cualquier dato tiene el potencial de causar problemas. El problema más común es un ataque de secuencias de comandos de sitios cruzados (XSS) donde se elabora un valor de datos para que el navegador lo interprete como elementos HTML o código JavaScript. En temas previos, mostramos cómo se puede usar una



política de seguridad de contenido para ayudar a prevenir XSS diciéndole al navegador cómo se espera que se comporte la aplicación, pero **otra buena medida es desinfectar datos que se reciben de un usuario para que no sea así**. Contiene caracteres que los navegadores interpretarán inesperadamente cuando se muestre a otro usuario. Para prepararse, el listado 17 cambia el controlador del formulario para que devuelva una respuesta HTML.

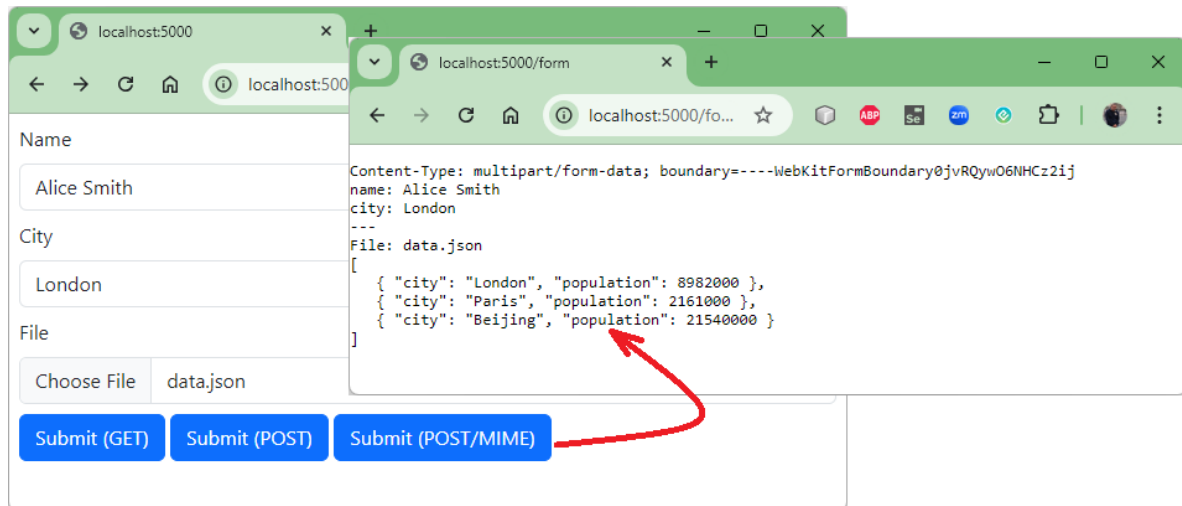


Figura 5: Carga de archivos.

Listado 17: Devolver una respuesta HTML en el archivo forms.ts en la carpeta src/server.

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

const fileMiddleware = multer({ storage: multer.memoryStorage() });

export const registerFormMiddleware = (app: Express) => {
  app.use(express.urlencoded({ extended: true }));
}

export const registerFormRoutes = (app: Express) => {
  app.get("/form", (req, resp) => {

    for (const key in req.query) {
      resp.write(`${key}: ${req.query[key]}\n`);
    }
    resp.end();
  });

  app.post("/form", fileMiddleware.single("datafile"), (req, resp) => {
    resp.setHeader("Content-Type", "text/html");
```

```

for (const key in req.body) {
  resp.write(`<div>${key}: ${req.body[key]}</div>`);
}

if (req.file) {
  resp.write(`<div>File: ${req.file.originalname}</div>`);
  resp.write(`<div>${req.file.buffer.toString()}</div>`);
}

resp.end();
});
}

```

La salida HTML es simple y sin establecimiento, que puedes ver solicitando `http://localhost:5000`, completa el formulario con los mismos detalles que para ejemplos anteriores y haciendo clic en el botón Submit (POST/MIME), como se muestra en la figura 6.

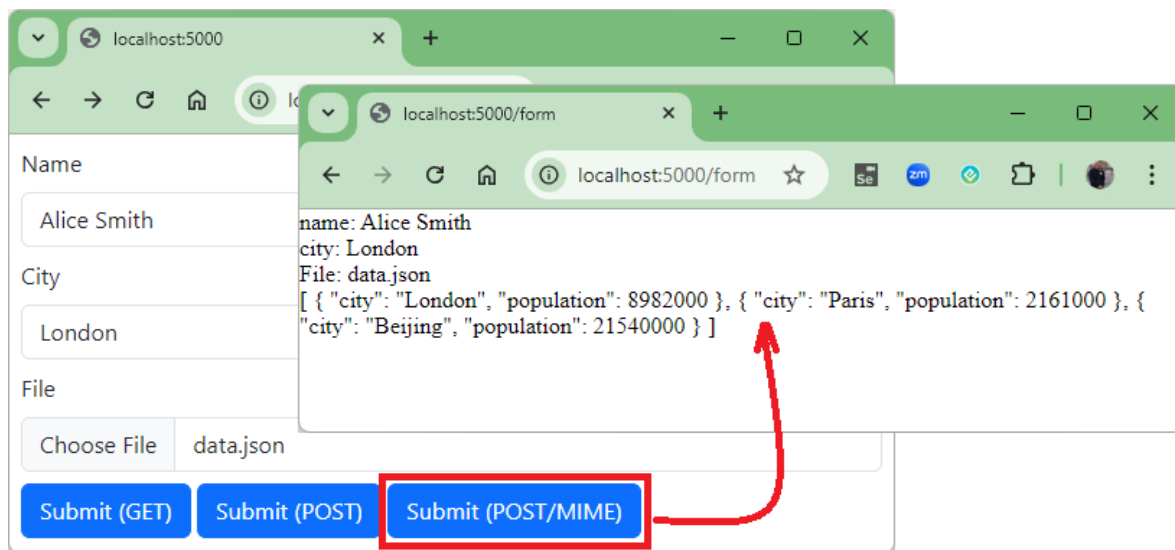


Figura 6: Producir una respuesta HTML.

Para ver el efecto del contenido inseguro, regresa a `http://localhost:5000` y completa el formulario utilizando los valores en la tabla 4

Tabla 4: valores de contenido inseguros.

Campo	Descripción
Name	<code>&lt;link href="css/bootstrap.min.css" rel="stylesheet" /&gt;</code>
City	<code>&lt;a class="btn btn-primary" href="http://amazon.com"&gt;Click Me!&lt;/a&gt;</code>

Haz clic en Submit (POST/MIME) y los valores que se ingresaron en el formulario se incluirán en la respuesta, que el navegador interpreta como un elemento de enlace para la hoja de estado de Bootstrap CSS y un elemento de anclaje que está diseñado para verse como un botón y que solicitará una URL que no sea parte de la aplicación, como se muestra en la figura 7.

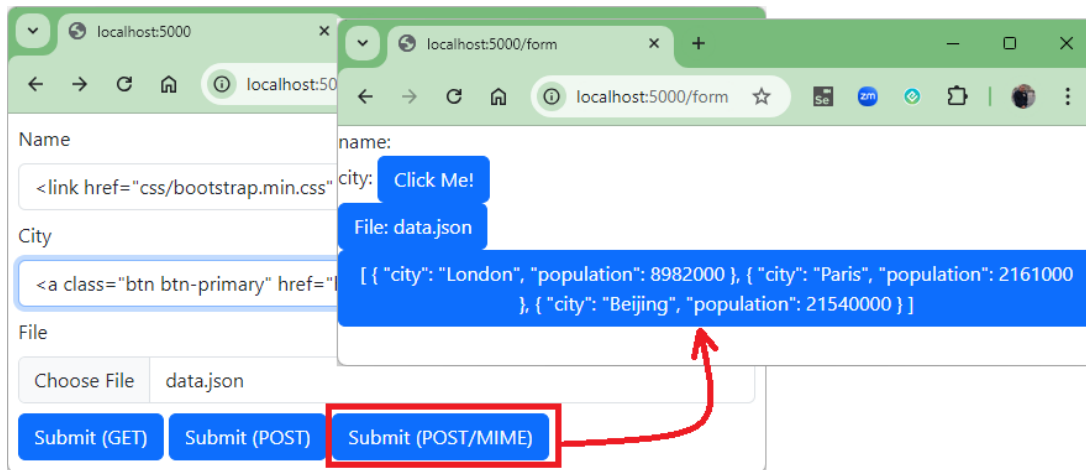


Figura 7: El efecto de mostrar contenido inseguro.

El proceso de desinfección implica reemplazar los caracteres que denotan contenido HTML con secuencias de escape que muestran el mismo carácter. La tabla 5 enumera los caracteres que generalmente se desinfectan y las secuencias de escape que los reemplazan.

Tabla 5: caracteres inseguros y secuencias de escape.

Carácter inseguro	Secuencia de escape
&	&amp;
<	&lt;
>	&gt;
=	&#x3D;
" (double quotes)	&quot;
' (single quote)	&#x27;
` (back tick)	&#x60;

Agrega un archivo llamado `sanitize.ts` a la carpeta `src/server` con el contenido que se muestra en el listado 18.

Listado 18: El contenido del archivo `sanitize.ts` en la carpeta `src/server`.

```
const matchPattern = /[<>="']/g;

const characterMappings: Record<string, string> = {
  "&": "&amp;",
```

```

"<": "&lt;";
">": "&gt;";
"\"": "&quot;";
"=": "&#x3D;";
"\"": "&#x27;";
"\"": "&#x60;";
};

export const sanitizeValue = (value: string) =>
  value?.replace(matchPattern, match => characterMappings[match]);

```

La función `sanitizeValue` aplica un patrón a una cadena para encontrar caracteres peligrosos y reemplazarlos con secuencias de escape seguras. Los valores de los datos se desinfectan ya que se incluyen en una respuesta HTML. Esto generalmente se realiza como parte del proceso de plantilla, como se muestra en breve, pero el listado 19 aplica la función `sanitizeValue` a los valores incluidos en la respuesta HTML.

Listado 19: Desinfección de valores de salida en el archivo `forms.ts` en la carpeta `src/server`.

```

import express, { Express } from "express";
import multer from "multer";
import { sanitizeValue } from "../sanitize";

const fileMiddleware = multer({ storage: multer.memoryStorage() });

export const registerFormMiddleware = (app: Express) => {
  app.use(express.urlencoded({ extended: true }));
}

export const registerFormRoutes = (app: Express) => {
  app.get("/form", (req, resp) => {

    for (const key in req.query) {
      resp.write(`${key}: ${req.query[key]}\n`);
    }
    resp.end();
  });

  app.post("/form", fileMiddleware.single("datafile"), (req, resp) => {
    resp.setHeader("Content-Type", "text/html");

    for (const key in req.body) {
      resp.write(`<div>${key}: ${sanitizeValue(req.body[key])}</div>`);
    }
  });
}

```

```

    if (req.file) {
      resp.write(`<div>File: ${req.file.originalname}</div>`);
      resp.write(`<div>${sanitizeValue(req.file.buffer.toString())}</div>`);
    }

    resp.end();
  });
}

```

Usa un navegador para solicitar `http://localhost: 5000`, completa el formulario con los detalles en la tabla 5 y haz clic en el botón Submit (POST/MIME). Los valores recibidos del usuario se desinfectan, ya que se incluyen en la respuesta HTML para que el navegador pueda mostrar las cadenas sin interpretarlas como elementos válidos, como se muestra en la figura 8.

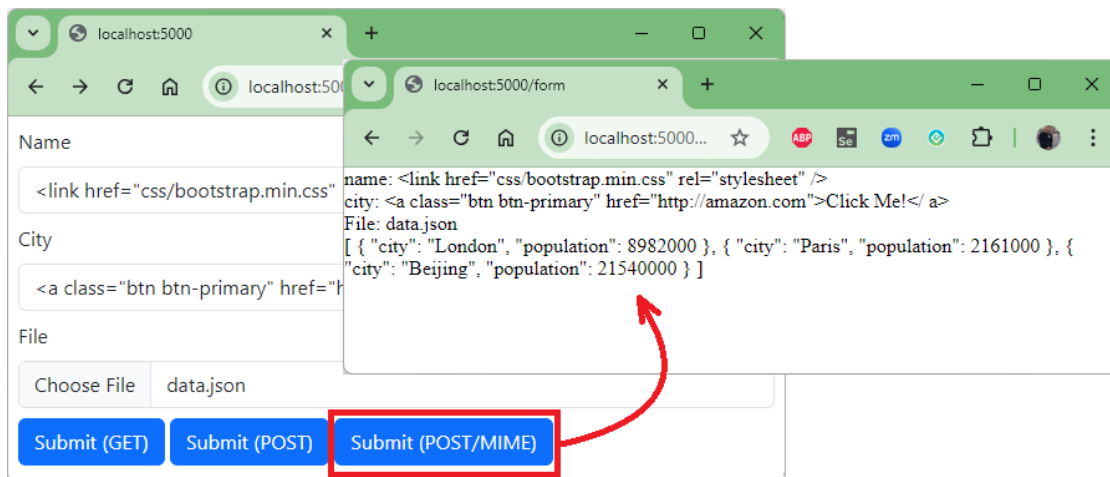


Figura 8: Desinfección de valores de datos.

### Desinfectar datos repetidamente

Debes asegurarte de que los datos se desinfecten, pero solo debes desinfectarlos una vez. Si los datos se desinfectan repetidamente, el carácter `&` se escapará repetidamente. Si comienzas con esta cadena insegura, por ejemplo:

```
<link href = "css /bootstrap.min.css" rel = "stylesheet" />
```

y desinfectas, el resultado será el siguiente:

```
&lt;link href=&#x3D;&quot;css/bootstrap.min.css&quot;
rel=&#x3D;&quot;stylesheet&quot; /&gt;
```

Los caracteres peligrosos se escapan, pero el navegador interpretará las secuencias de escape para que la cadena se vea como la original, pero no se interprete como un elemento

---

HTML. Si la cadena se desinfecta nuevamente, los caracteres &, que ya forman parte de las secuencias de escape, se reemplazarán con &amp;, produciendo este resultado:

```
&amp;lt;link href&amp;#x3D;&amp;quot;css/bootstrap.min.
css&amp;quot; rel&amp;#x3D;&amp;quot;stylesheet&amp;quot;
/&amp;gt;
```

El navegador no podrá interpretar las secuencias de escape correctamente y mostrará una cadena destrozada.

---

La mayoría de los paquetes de plantilla desinfectarán automáticamente los valores de los datos cuando se represente una plantilla, y esto incluye el paquete Handlebars agregado al proyecto en el documento anterior. Agrega un archivo llamado formData.handlebars a la carpeta templates/server con el contenido que se muestra en el listado 20.

Listado 20: El contenido del archivo formData.handlebars en la carpeta templates/server.

```
<table class="table table-sm table-striped">
  <thead>
    <tr><th>Field</th><th>Value</th></tr>
  </thead>
  <tbody>
    <tr><td>Name:</td><td>{{ name }} </td></tr>
    <tr><td>City:</td><td>{{ city }} </td></tr>
    <tr><td>File:</td><td>{{ fileData }} </td></tr>
  </tbody>
</table>
```

Handlebars desinfecta automáticamente los valores de datos en {{y}} expresiones, lo que hace que sea seguro incluir en respuestas HTML. El listado 21 actualiza el controlador de solicitud de formulario para usar la nueva plantilla.

Listado 21: Uso de una plantilla en el archivo forms.ts en la carpeta src/server.

```
import express, { Express } from "express";
import multer from "multer";
import { sanitizeValue } from "../sanitize";

const fileMiddleware = multer({ storage: multer.memoryStorage() });

export const registerFormMiddleware = (app: Express) => {
  app.use(express.urlencoded({ extended: true }))
}

export const registerFormRoutes = (app: Express) => {
  app.get("/form", (req, resp) => {
```

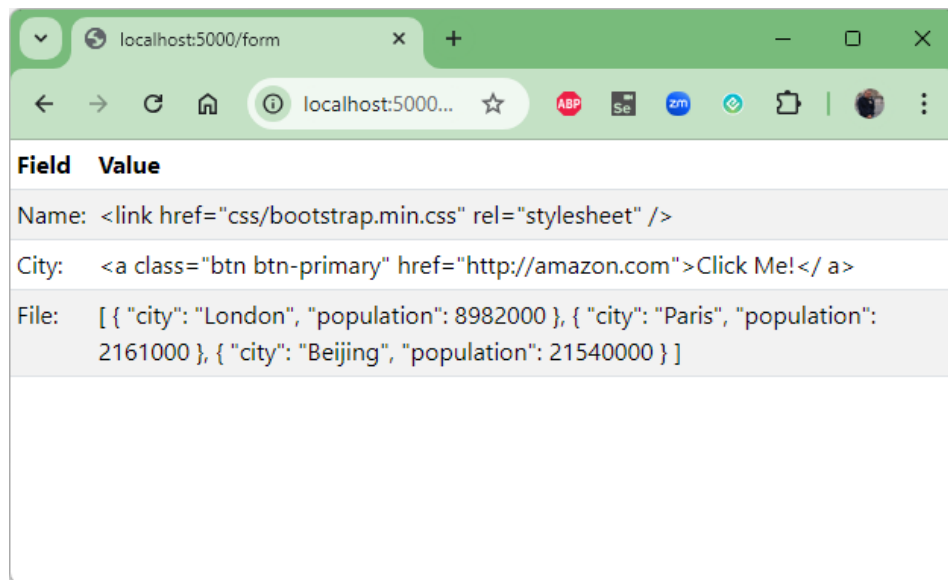
```

    for (const key in req.query) {
      resp.write(`${key}: ${req.query[key]}\n`);
    }
    resp.end();
  });

  app.post("/form", fileMiddleware.single("datafile"), (req, resp) => {
    resp.render("formData", {
      ...req.body, file: req.file,
      fileData: req.file?.buffer.toString()
    });
  });
}

```

El objeto de contexto pasado a la plantilla contiene las propiedades del cuerpo y los objetos de archivo y una propiedad `fileData` que proporciona acceso directo a los datos del archivo, ya que los Handlebars no evaluarán los fragmentos de código en plantillas. Solicita `http://localhost:5000`, completa el formulario utilizando los detalles en la tabla 4 y haz clic en el botón Submit (POST/MIME) y verás que la plantilla contiene valores seguros, como se muestra en la figura 9.



Field	Value
Name:	<link href="css/bootstrap.min.css" rel="stylesheet" />
City:	<a class="btn btn-primary" href="http://amazon.com">Click Me!</ a>
File:	[ { "city": "London", "population": 8982000 }, { "city": "Paris", "population": 2161000 }, { "city": "Beijing", "population": 21540000 } ]

Figura 9: Uso de una plantilla para desinfectar valores de datos.

### Consejo:

Handlebars siempre desinfectará los valores de datos en `{{ }}` expresiones. Si deseas incluir datos sin desinfección, usa las secuencias de caracteres `{{{y}}}` en su lugar, como se demuestra en el documento anterior.

---

## Nota

Cuando se combina con una política de seguridad de contenido, la desinfección de datos en plantillas HTML es una buena defensa básica contra los ataques XSS. Pero no es integral y pueden permanecer problemas potenciales, como cuando se insertan valores de datos del usuario en el código JavaScript que el navegador ejecutará. Se puede encontrar una buena lista de verificación para evitar tales problemas en: [https://cheatsheetseries.owasp.org/cheatsheets/Cross\\_Site\\_Scripting\\_Prevention\\_Cheat\\_Sheet.html](https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html)

---

## Validando datos del formulario

La desinfección de los datos puede ayudar a evitar que los valores maliciosos se muestren a los usuarios, pero eso no significa que los datos que recibas serán útiles. Los usuarios ingresarán casi cualquier cosa en una forma, a veces a través de un error genuino, pero principalmente porque los formularios son un obstáculo desagradable entre el usuario y su objetivo, sea lo que sea.

El resultado es que los datos recibidos de los formularios deben validarse, que es el proceso de garantizar que los datos puedan ser utilizados por la aplicación y decirle al usuario cuándo se reciben datos no válidos.

La validación de formulario se realiza más fácilmente con una plantilla porque facilita darle al usuario retroalimentación cuando surge un problema. Para prepararte para la validación, agrega un archivo llamado age.handlebars a la carpeta templates/server con el contenido que se muestra en el listado 22.

Listado 22: El contenido del archivo age.handlebars en la carpeta templates/server.

```
<div class="m-2">
  {{#if nextage }}
    <h4>Hello {{name}}. You will be {{nextage}} next year.</h4>
  {{/if }}
</div>
<div>
  <form action="/form" method="post">
    <div class="m-2">
      <label class="form-label">Name</label>
      <input name="name" class="form-control" value="{{name}}"/>
    </div>
    <div class="m-2">
      <label class="form-label">Current Age</label>
      <input name="age" class="form-control" value="{{age}}"/>
    </div>
  </form>
</div>
```



```

    </div>
    <div class="m-2">
      <button class="btn btn-primary">Submit</button>
    </div>
  </form>
</div>

```

Esta plantilla contiene un formulario que le pide al usuario su nombre y edad para que el servidor pueda calcular su edad el próximo año. Esta es una aplicación trivialmente simple, pero contiene suficiente funcionalidad para requerir validación. El listado 23 actualiza las rutas para el URL de /form para usar la nueva plantilla.

Listado 23: Actualización de rutas en el archivo forms.ts en la carpeta src/server.

```

import express, { Express } from "express";
//import multer from "multer";
//import { sanitizeValue } from "../sanitize";

//const fileMiddleware = multer({storage: multer.memoryStorage()});

export const registerFormMiddleware = (app: Express) => {
  app.use(express.urlencoded({extended: true}))
}

export const registerFormRoutes = (app: Express) => {
  app.get("/form", (req, resp) => {
    resp.render("age");
  });

  app.post("/form", (req, resp) => {
    resp.render("age", {
      ...req.body,
      nextage: Number.parseInt(req.body.age) + 1
    });
  });
}

```

La ruta get renderiza la plantilla de edad sin datos de contexto. La ruta post renderiza la plantilla con los datos del formulario recibidos en el cuerpo y una propiedad de nextage, que se crea analizando el valor de edad recibido del formulario a un número y agregando uno.

Usa un navegador para solicitar <http://localhost:5000/form>, ingresa un nombre y edad en el formulario, y haz clic en el botón Submit. Si repites el proceso pero proporciona una edad no

numérica, la aplicación no podrá analizar los datos del formulario y no producirá un resultado. Ambos resultados se muestran en la figura 10.

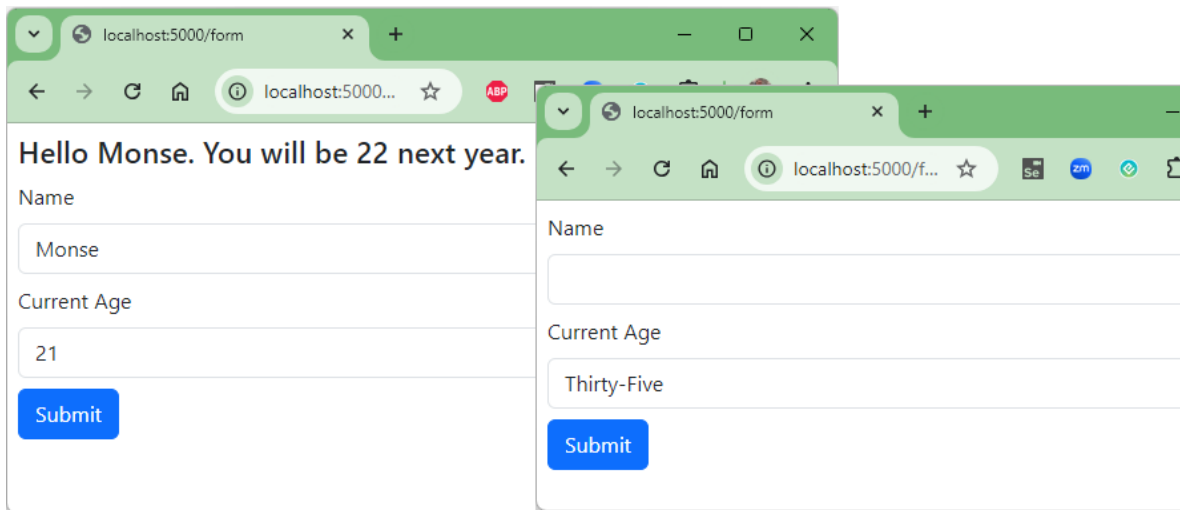


Figura 10: Una aplicación que utiliza datos de formulario para producir un resultado.

La aplicación tiene expectativas para los datos que recibe, y la validación es el proceso de garantizar que se cumplan esas expectativas.

---

### Nota

La validación es una forma de hacer que el usuario complete el formulario, pero debe tomarse un momento para preguntar si el formulario debe existir. Si deseas una mayor satisfacción del usuario con su aplicación, mantén los formularios simples y claros, y solicita solo el mínimo necesario para hacer el trabajo. Se flexible sobre los formatos que aceptarás para valores de datos complejos, como números o fechas de tarjetas de crédito, y haz que los mensajes de error de validación sean lo más claros como puedas.

---

## Creación de un validador personalizado

La validación requiere un conjunto de pruebas que se pueden aplicar para formar datos tal como se recibe. Agrega un archivo llamado `validation.ts` a la carpeta `src/server`, con el contenido que se muestra en el listado 24.

Listado 24: El contenido del archivo `validation.ts` en la carpeta `src/server`.

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

type ValidatedRequest = Request & {
  validation: {
    results: { [key: string]: {
```

```

    [key: string]: boolean, valid: boolean
  } },
  valid: boolean
}
}

export const validate = (propName: string) => {
  const tests: Record<string, (val: string) => boolean> = {};
  const handler = (req: Request, resp: Response, next: NextFunction ) => {

    // TODO - perform validation checks

    next();
  }
  handler.required = () => {
    tests.required = (val: string) => val?.trim().length > 0;
    return handler;
  };
  handler.minLength = (min: number) => {
    tests.minLength = (val:string) => val?.trim().length >= min;
    return handler;
  };
  handler.isInteger = () => {
    tests.isInteger = (val: string) => /^[0-9]+$/.test(val);
    return handler;
  }
  return handler;
}

export const getValidationResults = (req: Request) => {
  return (req as ValidatedRequest).validation || { valid : true }
}

```

Hay muchas maneras de implementar un sistema de validación, pero el enfoque adoptado en el listado 24 es seguir el patrón introducido por otros paquetes utilizados en esta parte del curso y crear un middleware Express que agrega una propiedad al objeto Request. El código aún no está completo porque no aplica verificaciones de validación. Pero sí permite definir los requisitos de validación, y ese es un buen lugar para comenzar porque el código requerido para realizar fácilmente la validación puede ser complicado.

El código inicial define tres reglas de validación: `required`, `minLength` e `isInteger`. Los paquetes de validación real, como el que presentamos más adelante en este documento, tienen docenas de reglas diferentes, pero tres son suficientes para demostrar cómo funciona la

validación de datos del formulario. La regla requerida asegura que el usuario haya suministrado un valor, la regla minLength impone un número mínimo de caracteres, y la regla isInteger asegura que el valor sea un entero.

El punto de partida es dar a TypeScript una descripción de la propiedad que se agregará al objeto Request, que es cómo los resultados de validación se presentarán a la función del controlador de solicitudes (request handler):

```
...
type ValidatedRequest = Request & {
  validation: {
    results: { [key: string]: {
      [key: string]: boolean, valid: boolean
    } },
    valid: boolean
  }
}
..
```

El tipo validatedRequest tiene todas las características definidas por Request, más una propiedad con nombre validation que devuelve un objeto con resultados y propiedades válidas. La propiedad valid devuelve un valor booleano que proporciona una indicación general del resultado de validación de datos del formulario. La propiedad de resultados proporciona información detallada sobre los campos de datos del formulario que han sido validados. El objetivo es producir un objeto que se vea así:

```
...
{
  results: {
    name: { valid: false, required: true, minLength: false },
    age: { valid: true, isNumber: true }
  },
  valid: false
}
...
```

Este objeto representa las verificaciones de validación realizadas en las propiedades de nombre (name) y edad (age). En general, los datos del formulario no son válidos e inspeccionan el detalle, puedes ver que esto se debe a que la propiedad de nombre ha fallado en sus comprobaciones de validación, específicamente porque el valor del nombre no ha pasado la regla minLength.

La función `validate` devuelve una función Express Middleware que también tiene métodos, lo que permite definir la validación encadenando las reglas de validación para una propiedad. `getValidationResults` lee la propiedad de validación agregada a la solicitud, lo que facilita el acceso a los datos de validación en el controlador de solicitudes.

## Aplicar reglas de validación

La creación de una función que también tiene métodos aprovecha la flexibilidad de JavaScript, de modo que las reglas de validación se puedan especificar llamando al método `validate` para seleccionar un campo de formulario y luego se puede llamar a los métodos en el resultado para especificar las reglas de validación. Esto no es esencial, pero permite que los requisitos de validación se expresen de manera concisa, como se muestra en el listado 25.

Listado 25: Definición de reglas de validación en el archivo `forms.ts` en la carpeta `src/server`.

```
import express, { Express } from "express";
import { getValidationResults, validate } from "../validation";

export const registerFormMiddleware = (app: Express) => {
  app.use(express.urlencoded({ extended: true }))
}

export const registerFormRoutes = (app: Express) => {
  app.get("/form", (req, resp) => {
    resp.render("age", { helpers: { pass } });
  });

  app.post("/form",
    validate("name").required().minLength(5),
    validate("age").isInteger(),
    (req, resp) => {
      const validation = getValidationResults(req);
      const context = { ...req.body, validation,
        helpers: { pass }
      };
      if (validation.valid) {
        context.nextage = Number.parseInt(req.body.age) + 1;
      }
      resp.render("age", context);
    });
}

const pass = (valid: any, propname: string, test: string) => {
  let propResult = valid?.results?.[propname];
```

```

    return `display:${!propResult || propResult[test] ? "none" : "block" }`;
  }

```

El resultado de llamar a un método de regla es la función del controlador que lo define, lo que significa que se pueden seleccionar múltiples reglas encadenando las llamadas del método. El listado 25 aplica las reglas required y minLength en el campo nombre y la regla isInteger al campo de edad.

La función getValidationResults se llama dentro de la función del controlador para obtener los resultados de validación, que se utilizan para alterar el objeto de contexto utilizado para representar la vista para que el cálculo (simple) solo se realiza cuando los datos válidos se han recibido del usuario.

Los resultados de validación se incluyen en el objeto de contexto de la plantilla, que permite que un ayudante de plantilla inspeccione los resultados y controle la visibilidad de los elementos de error de validación. Los elementos que muestran errores a los usuarios siempre estarán presentes en la plantilla, y el listado 25 define un ayudante de plantilla llamado pass que se utilizará para controlar la visibilidad.

El listado 26 actualiza la plantilla para incluir los elementos del mensaje de error.

Listado 26: Agregar mensajes de validación en el archivo age.handlebars en la carpeta templates/server.

```

<div class="m-2">
  {{#if validation.valid }}
    <h4>Hello {{name}}. You will be {{nextage}} next year.</h4>
  {{/if }}
</div>
<div>
  <form id="age_form" action="/form" method="post">
    <div class="m-2">
      <label class="form-label">Name</label>
      <input name="name" class="form-control" value="{{name}}"/>
      <div class="text-danger" id="err_name_required"
        style="{{ pass validation 'name' 'required' }}">
        Please enter your name
      </div>
      <div class="text-danger" id="err_name_minLength"
        style="{{ pass validation 'name' 'minLength' }}">
        Enter at least 5 characters
      </div>
    </div>
  </div>

```

```

<div class="m-2">
  <label class="form-label">Current Age</label>
  <input name="age" class="form-control" value="{{age}}" />
  <div class="text-danger" id="err_age_isInteger"
    style="{{ pass validation 'age' 'isInteger' }}">
    Please enter your age in whole years
  </div>
</div>
<div class="m-2">
  <button class="btn btn-primary">Submit</button>
</div>
</form>
</div>

```

Las nuevas incorporaciones aseguran que los resultados solo se muestren si los datos del formulario son válidos y muestran errores de validación cuando hay un problema. Incluir los elementos de error en la plantilla será útil para la validación del lado del cliente, que se demuestra más adelante en este documento.

## Validación de datos

El paso final es completar el validador personalizado aplicando las pruebas a un valor, como se muestra en el listado 27.

Listado 27: Completar el validador en el archivo validation.ts en la carpeta src/server.

```

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

type ValidatedRequest = Request & {
  validation: {
    results: { [key: string]: {
      [key: string]: boolean, valid: boolean
    } },
    valid: boolean
  }
}

export const validate = (propName: string) => {
  const tests: Record<string, (val: string) => boolean> = {};
  const handler = (req: Request, resp: Response, next: NextFunction) => {

    const vreq = req as ValidatedRequest;
    if (!vreq.validation) {
      vreq.validation = { results: {}, valid: true };
    }
  }
}

```

```

    }
    vreq.validation.results[propName] = { valid: true };

    Object.keys(tests).forEach(k => {
      let valid = vreq.validation.results[propName][k]
        = tests[k](req.body?.[propName]);
      if (!valid) {
        vreq.validation.results[propName].valid = false;
        vreq.validation.valid = false;
      }
    });

    next();
  }
  handler.required = () => {
    tests.required = (val: string) => val?.trim().length > 0;
    return handler;
  };
  handler.minLength = (min: number) => {
    tests.minLength = (val:string) => val?.trim().length >= min;
    return handler;
  };
  handler.isInteger = () => {
    tests.isInteger = (val: string) => /^[0-9]+$/.test(val);
    return handler;
  }
  return handler;
}

export const getValidationResults = (req: Request) => {
  return (req as ValidatedRequest).validation || { valid : true }
}

```

Dejamos este paso hasta el final para que las otras partes del sistema de validación sean más fáciles de entender. Cada vez que se llama uno de los métodos de regla de validación, como `required`, se agrega una nueva propiedad al objeto asignado a las pruebas con nombre constante. Para realizar la validación, las propiedades de las pruebas se enumeran, cada prueba se realiza y el resultado se utiliza para construir los resultados de validación.

Si alguna prueba de validación falla, entonces el resultado de validación general y el resultado para el valor de campo actual se establecen en falso.



Usa un navegador para solicitar `http://localhost:5000` y haz clic en el botón Submit sin ingresar valores en los campos del formulario. La validación fallará y los mensajes de error se mostrarán al usuario, como se muestra en la figura 11.

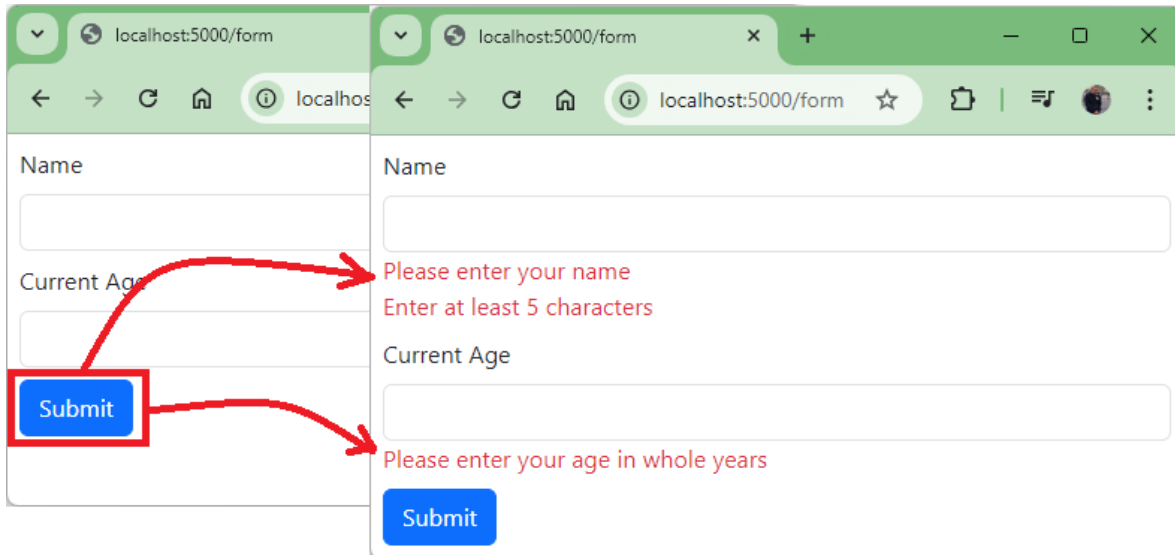


Figura 11: Mostrando errores de validación.

Se muestra un mensaje de error para cada regla de validación que falla, y el servidor de backend no generará una respuesta normal hasta que tenga éxito.

## Realización de validación del lado del cliente

La validación del lado del cliente checa valores del formulario antes de enviar el formulario, lo que puede proporcionar comentarios inmediatos al usuario. La validación del lado del cliente se usa además de la validación del lado del servidor, que aún se requiere porque los usuarios pueden deshabilitar el código JavaScript del lado del cliente o enviar datos de formulario manualmente.

---

### Comprensión de las funciones de validación del cliente HTML incorporadas

HTML admite atributos de validación en elementos de entrada, junto con una API de JavaScript que permite recibir eventos de validación, los cuales se describen en [https://developer.mozilla.org/en-us/docs/learn/forms/form\\_validation](https://developer.mozilla.org/en-us/docs/learn/forms/form_validation).

Estas características pueden ser útiles, pero no siempre se implementan de manera consistente y proporcionan solo verificaciones de validación básicas. Se requiere solo un poco más de trabajo para crear un sistema de validación más completo, por lo que no se usan en este documento.

---

La llave para el desarrollo del lado del cliente es la consistencia. Esto se puede lograr utilizando el mismo paquete para la validación del lado del cliente y el servidor, que es el enfoque que adoptamos en la siguiente sección.

De lo contrario, es importante garantizar que los campos se validen de la misma manera y produzcan los mismos mensajes de error. Agrega un archivo llamado `client_validation.js` a la carpeta `src/client` con el código que se muestra en el listado 28.

Listado 28: El contenido del archivo `client_validation.js` en la carpeta `src/client`.

```
export const validate = (propName, formdata) => {

  const val = formdata.get(propName);
  const results = { };

  const validationChain = {
    get propertyName() { return propName },
    get results () { return results }
  };
  validationChain.required = () => {
    results.required = val?.trim().length > 0;
    return validationChain;
  }
  validationChain.minLength = (min) => {
    results.minLength = val?.trim().length >= min;
    return validationChain;
  };
  validationChain.isInteger = () => {
    results.isInteger = /^[0-9]+$/.test(val);
    return validationChain;
  }
  return validationChain;
}
```

Este código JavaScript sigue un patrón similar al código TypeScript utilizado para configurar cadenas de pruebas de validación en el listado 24, aunque sin integración en Express. El listado 29 actualiza el código del lado del cliente para validar los datos del formulario.

Listado 29: Validación de datos del formulario en el archivo `client.js` en la carpeta `src/client`.

```
import { validate } from "../client_validation";

document.addEventListener('DOMContentLoaded', () => {
  document.getElementById("age_form").onsubmit = (ev => {
    const data = new FormData(ev.target);
```

```

const nameValid = validate("name", data)
  .required()
  .minLength(5);

const ageValid = validate("age", data)
  .isInteger();

const allValid = [nameValid, ageValid].flatMap(v_result =>
  Object.entries(v_result.results).map(([test, valid]) => {
    const e = document.getElementById(
      `err_${v_result.propertyName}_${test}`);
    e.classList.add("bg-dark-subtle");
    e.style.display = valid ? "none" : "block";
    return valid
  })).every(v => v === true);

if (!allValid) {
  ev.preventDefault();
}
});
});

```

Este código localiza el elemento del formulario en el documento HTML y registra un controlador para el evento Submit, que se emite cuando el usuario hace clic en el botón **Submit**. La API FormData del navegador se utiliza para obtener los datos en el formulario, que se prueba utilizando las funciones de validación definidas en el listado 28. Los resultados de validación se utilizan para cambiar la visibilidad de los elementos del mensaje de error en la plantilla. Si hay algún error de validación, se llama al método preventDefault en el evento Submit, que le dice al navegador que no envíe los datos al servidor.

El listado 29 conserva el mismo estilo para expresar los requisitos de validación, lo que conduce a algún código denso para procesar los resultados, encontrar los elementos que corresponden a cada prueba que se ha realizado y establecer la visibilidad del elemento.

Para este ejemplo, los elementos del mensaje de error se agregan a una clase de Bootstrap CSS cuando el cliente y no el servidor ha mostrado el código JavaScript del lado del cliente, solo para enfatizar cuándo ha mostrado un error por el servidor.

Usa un navegador para solicitar <http://localhost:5000/form> y haz clic en el botón Submit sin completar el formulario. Se mostrarán los elementos del mensaje de error, pero con un color

de fondo sólido que indica que se mostraron por el código del lado del cliente, como se muestra en la figura 12.

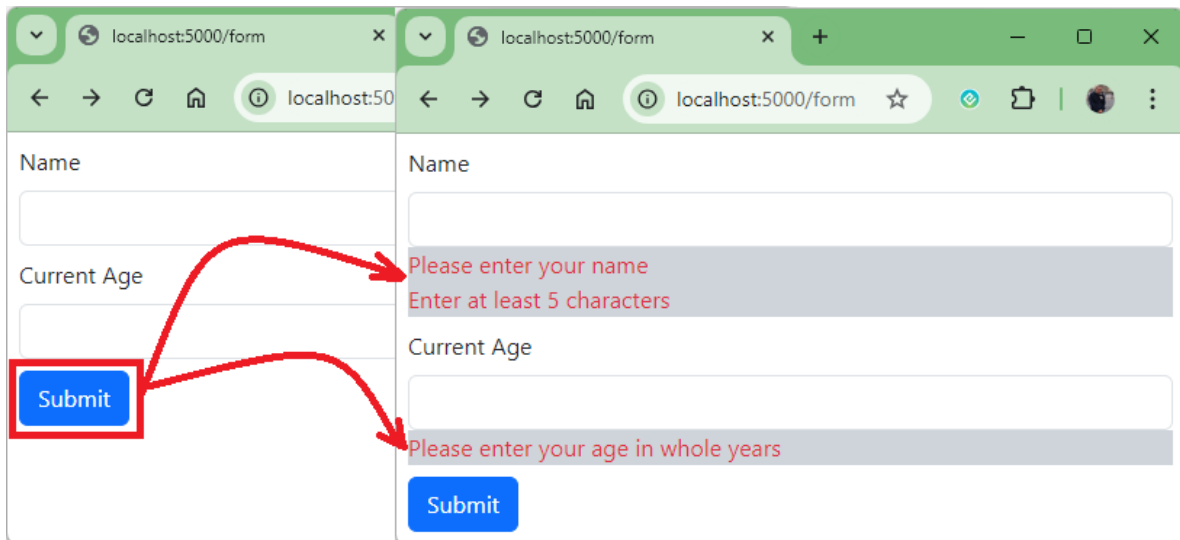


Figura 12: Uso de la validación del lado del cliente.

## Uso de un paquete para la validación

Habiendo demostrado cómo funciona la validación del formulario del lado del servidor y del lado del cliente, es hora de reemplazar las verificaciones personalizadas con las proporcionadas por una biblioteca de validación bien probada y completa.

Como con la mayoría de las áreas de la funcionalidad de JavaScript, hay muchas bibliotecas disponibles, y la que se ha elegido para este documento, validator.js, es simple y efectiva y puede usarse para la validación del lado del cliente y el servidor.

Ejecuta los comandos que se muestran en el listado 30 en la carpeta part2app para instalar los paquetes.

Listado 30: Instalación de un paquete de validación.

```
npm install validator@13.11.0
npm install --save-dev @types/validator@13.11.5
```

El listado 31 actualiza el código de validación del lado del cliente para usar las pruebas proporcionadas por el paquete validator.js.

Listado 31: Uso de un paquete de validación en el archivo client\_validation.js en la carpeta src/client.

```
import validator from "validator";

export const validate = (propName, formdata) => {

  const val = formdata.get(propName);
  const results = { };

  const validationChain = {
    get propertyName() { return propName },
    get results () { return results }
  };
  validationChain.required = () => {
    results.required = !validator.isEmpty(val, { ignore_whitespace: true });
    return validationChain;
  }
  validationChain.minLength = (min) => {
    results.minLength = validator.isLength(val, { min });
    return validationChain;
  };
  validationChain.isInteger = () => {
    results.isInteger = validator.isInt(val);
    return validationChain;
  }
  return validationChain;
}
```

El conjunto completo de pruebas proporcionadas por el paquete `validator.js` se puede encontrar en <https://github.com/validatorjs/validator.js> y el listado 31 utiliza tres de estas pruebas para reemplazar la lógica personalizada, mientras que el resto del código sigue siendo el mismo.

El mismo conjunto de cambios se puede aplicar al servidor, como se muestra en el listado 32, asegurando una validación consistente.

Listado 32: Uso de un paquete de validación en el archivo `validation.ts` en la carpeta `src/server`.

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

type ValidatedRequest = Request & {
  validation: {
    results: { [key: string]: {
```

```

    [key: string]: boolean, valid: boolean
  } },
  valid: boolean
}
}

export const validate = (propName: string) => {
  const tests: Record<string, (val: string) => boolean> = {};
  const handler = (req: Request, resp: Response, next: NextFunction ) => {

    const vreq = req as ValidatedRequest;
    if (!vreq.validation) {
      vreq.validation = { results: {}, valid: true };
    }
    vreq.validation.results[propName] = { valid: true };

    Object.keys(tests).forEach(k => {
      let valid = vreq.validation.results[propName][k]
        = tests[k](req.body?.[propName]);
      if (!valid) {
        vreq.validation.results[propName].valid = false;
        vreq.validation.valid = false;
      }
    });

    next();
  }
  handler.required = () => {
    tests.required = (val: string) =>
      !validator.isEmpty(val, { ignore_whitespace: true });
    return handler;
  };
  handler.minLength = (min: number) => {
    tests.minLength = (val:string) => validator.isLength(val, { min });
    return handler;
  };
  handler.isInteger = () => {
    tests.isInteger = (val: string) => validator.isInt(val);
    return handler;
  }
  return handler;
}

export const getValidationResults = (req: Request) => {

```

```
return (req as ValidatedRequest).validation || { valid : true }  
}
```

Solicita `http://localhost:5000/form` y envía el formulario y verás los mensajes de validación que se muestran en la figura 13. **Deshabilita JavaScript en el navegador y repite el proceso, y verás los mismos mensajes de validación, pero esta vez mostrados por el servidor, también se muestra en la figura 13.**

---

### Consejo

Para Google Chrome, puedes deshabilitar JavaScript en las ventanas de desarrollador F12 seleccionando el comando Run en el menú con tres puntos verticales e ingresar a Java en el cuadro de texto. El navegador presentará los comandos de Disable JavaScript o Enable JavaScript.

---

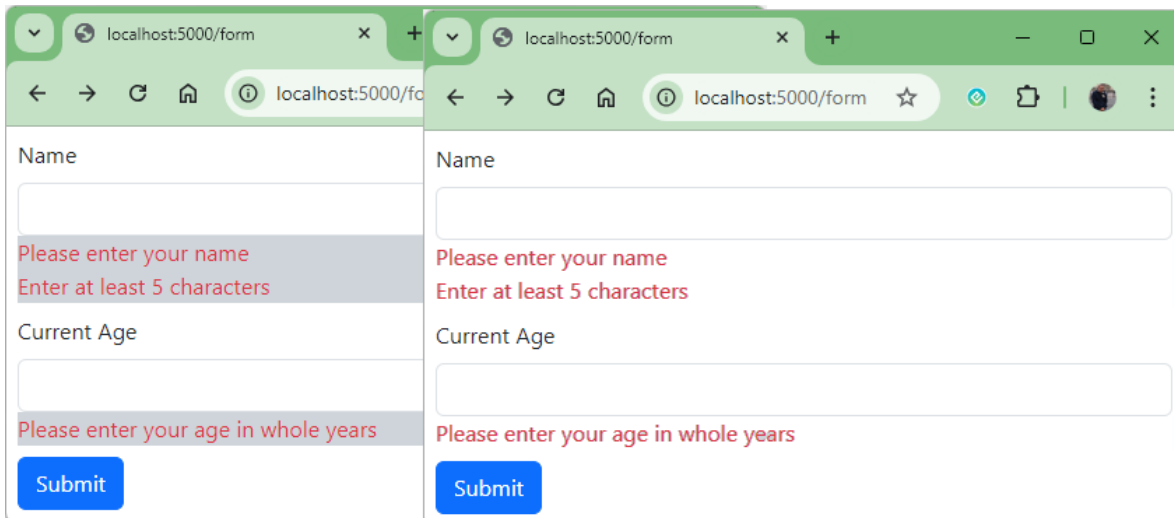


Figura 13: Uso de un paquete de validación.

No hay cambios en la forma en que la validación aparece al usuario, pero el uso de un paquete de validación aumenta la confianza de que la validación se realizará con precisión y proporciona acceso a una gama mucho más amplia de pruebas de validación.

### Resumen

**En este documento, describimos las diferentes formas en que las aplicaciones pueden recibir datos del formulario, hacer que sea seguro manejar y verificar que son los datos que requiere la aplicación:**

- Los datos del formulario se pueden enviar utilizando solicitudes GET y POST, lo que afecta cómo se codifican los datos.
- Se requiere precaución al enviar datos con solicitudes GET porque los resultados pueden almacenarse en caché.
- Hay diferentes codificaciones disponibles para los formularios enviados a través de solicitudes de publicación, incluida una codificación que permite enviar datos de archivos.
- Los datos de formulario deben desinfectarse antes de que se incluya en la salida HTML o se use en cualquier operación donde los valores puedan evaluarse como contenido confiable.
- Los datos de formulario deben validarse antes de que se utilicen para garantizar que la aplicación utilice los valores enviados por el usuario de manera segura.
- El servidor o el cliente pueden hacer la validación. La validación del lado del cliente no reemplaza la validación del lado del servidor.

En el próximo documento, explicaremos cómo se utilizan las bases de datos en las aplicaciones Node.js y cómo los datos pueden incluirse en el contenido HTML enviado al cliente.