# Uso de sesiones

En este documento, explicamos cómo las aplicaciones Node.js pueden correlacionar las solicitudes HTTP para crear sesiones, que permiten que los resultados de una solicitud afecten el resultado de futuras solicitudes. La tabla 1 pone este documento en contexto. La tabla 2 resume el documento.

Tabla 1: Poner las sesiones en contexto.

| Pregunta | Respuesta |
|---|---|
| ¿Qué son? | Las sesiones correlacionan las solicitudes realizadas por un usuario, lo que permite asociar las solicitudes entre sí. |
| ¿Por qué son útiles? | Las sesiones permiten implementar funciones de aplicaciones con estado mediante solicitudes HTTP sin estado. |
| ¿Cómo se utilizan? | Las cookies se utilizan para transmitir pequeñas cantidades de datos o un ID de sesión asociado con los datos almacenados por el servidor, que identifica las solicitudes relacionadas. |
| ¿Existen limitaciones o inconvenientes? | Los navegadores a veces utilizan cookies de formas que no son útiles para administrar sesiones, pero con cuidado, las sesiones tienen pocos inconvenientes. |
| ¿Existen alternativas? | Las sesiones basadas en cookies son la única forma confiable de correlacionar solicitudes HTTP, pero no todas las aplicaciones requieren correlación de solicitudes. |

Tabla 2: Resumen del documento.

| Problema | Solución | Listado |
|---|---|---|
| Correlacionar solicitudes HTTP relacionadas. | Establecer y leer cookies. | 2-5, 8-10 |
| Evitar que se alteren los datos almacenados en las cookies. | Firmar y verificar cookies. | 6, 7 |
| Almacenar grandes cantidades de datos. | Usar sesiones en las que la aplicación almacena los datos y se accede a ellos mediante una llave almacenada en una cookie. | 11-15, 19-21 |
| Almacenar de forma persistente los datos de la sesión. | Usar una base de datos. | 16-18 |

## Preparación para este documento

Este documento/práctica utiliza el proyecto part2app del documento anterior. No se requieren cambios para este documento.

Ejecutar el comando que se muestra en el listado 1 en la carpeta part2app para iniciar las herramientas de desarrollo.

Listado 1: Iniciar las herramientas de desarrollo.

```
npm start
```

Utilizar un navegador para solicitar http://localhost:5000, completar el formulario y hacer clic en el botón Enviar (**Submit**), como se muestra en la figura 1.

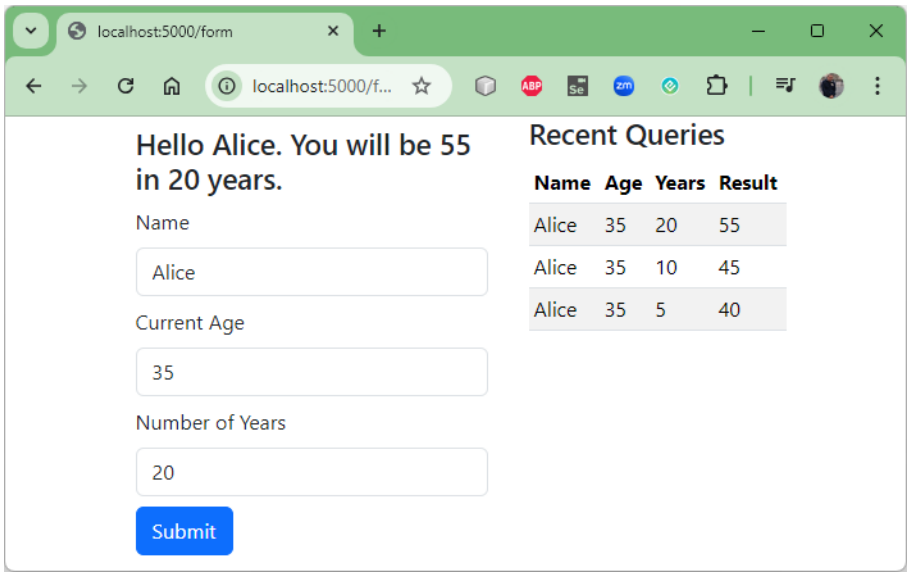


Figura 1: Ejecutar la aplicación de ejemplo.

## Correlación de solicitudes HTTP sin estado

Las solicitudes HTTP no tienen estado (*stateless*), lo que significa que cada solicitud es autónoma y no contiene información que la asocie con ninguna otra solicitud, incluso cuando se realiza desde el mismo navegador.

Puedes ver el problema que esto crea al abrir dos ventanas del navegador y completar el formulario con el mismo nombre pero con diferentes edades y cantidad de años, simulando dos usuarios con el mismo nombre.

La única información con la que el servidor puede trabajar son los datos del formulario y no tiene forma de averiguar que se trata de solicitudes de diferentes usuarios, por lo que los usuarios ven los datos de los demás y cualquier otro dato creado por usuarios con el mismo nombre, como se muestra en la figura 2.

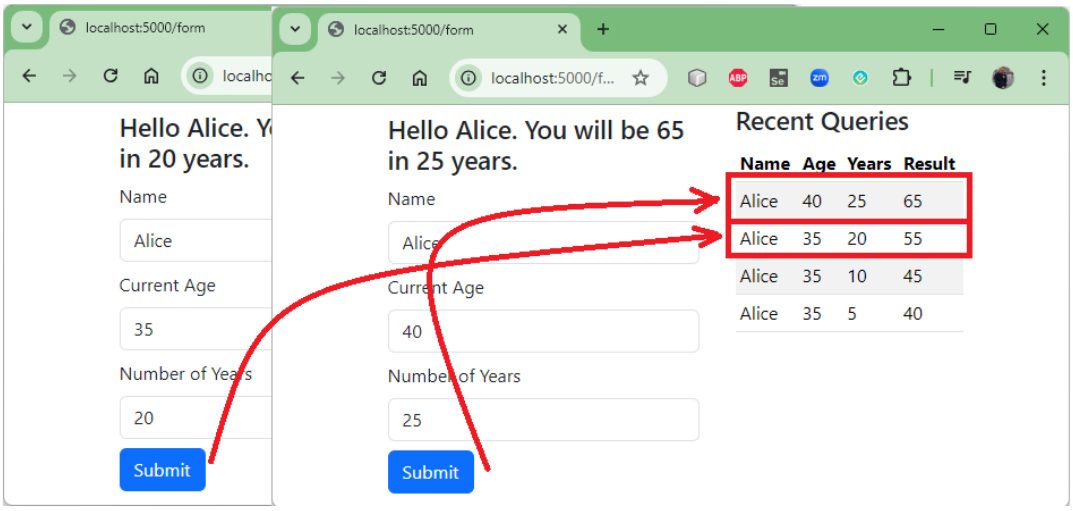


Figura 2: El efecto de las solicitudes sin estado.

La mayoría de las aplicaciones tienen *estado*, y eso significa que el servidor debe poder correlacionar las solicitudes para que la aplicación pueda reflejar acciones pasadas en respuestas futuras. En el caso del ejemplo, esto permitiría a la aplicación mostrar solo las solicitudes realizadas por un usuario y no solo todas las solicitudes realizadas por cualquier persona que tenga el mismo nombre.

**Uso de cookies para correlacionar solicitudes**

La forma más común de correlacionar solicitudes es con una cookie. Las cookies son pequeños fragmentos de texto que un servidor incluye en un encabezado de respuesta HTTP.

El navegador incluye las cookies en solicitudes posteriores, lo que significa que, si el servidor crea cookies con identificadores únicos, esas solicitudes se pueden identificar como relacionadas. (Existen otras formas de correlacionar solicitudes, como incluir identificadores únicos en las URL, pero las cookies son el método más sólido y confiable).

Las cookies se pueden configurar como cualquier encabezado de respuesta. Agrega un archivo llamado cookies.ts a la carpeta src/server, con el contenido que se muestra en el listado 2.

Listado 2: El contenido del archivo cookies.ts en la carpeta src/server.

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

const setheaderName = "Set-Cookie";

export const setCookie = (resp: ServerResponse, name: string,
    val: string) => {
```

```
  let cookieVal: any[] = [`${name}=${val}; Max-Age=300; SameSite=Strict }`];
  if (resp.hasHeader(setheaderName)) {
     cookieVal.push(resp.getHeader(setheaderName));
  }
  resp.setHeader("Set-Cookie", cookieVal);
}

export const setJsonCookie = (resp: ServerResponse, name: string,
     val: any) => {
  setCookie(resp, name, JSON.stringify(val));
}
```

Las cookies se envían al navegador mediante el encabezado Set-Cookie, y el valor del encabezado es un nombre de cookie, un valor y uno o más atributos que le indican al navegador cómo administrar la cookie. Una respuesta puede configurar varias cookies al incluir varios encabezados Set-Cookie.

Por este motivo, el código del listado 2 verifica si existe un encabezado Set-Cookie y agrega su valor al arreglo de valores que se pasan al método setHeader. Cuando se escribe la respuesta, Node.js agregará un encabezado Set-Cookie para cada elemento del arreglo.

**Precaución para el usuario**

Se requiere el consentimiento para las cookies en algunas partes del mundo, especialmente dentro de la EU con el Reglamento General de Protección de Datos (GDPR, General Data Protection Regulation).

No somos abogados y no estamos calificados para brindar asesoramiento legal, pero debes asegurarte de comprender las leyes en cada región donde tu aplicación tiene usuarios y asegurarte de cumplir con las reglas.

Un encabezado producido por la función setCookie en el listado 2 se verá así:

```
...
Set-Cookie: user=Alice; Max-Age=300; SameSite=Strict
...
```

El nombre de la cookie es user, su valor es Alice y la cookie ha sido configurada con los atributos Max-Age y SameSite, que le indican al navegador durante cuánto tiempo se valora la cookie y cuándo enviarla. Los atributos de la cookie se describen en la tabla 3.

Tabla 3: Atributos de las cookies.

| Nombre | Descripción |
|---|---|
| Domain=value | Este atributo especifica el dominio de la cookie, como se describe después de esta tabla. |
| Expires=date | Este atributo especifica la hora y la fecha en que caduca la cookie. El formato de los datos se describe en: https://developer.mozilla.org/enUS/docs/Web/HTTP/Headers/Date. Para la mayoría de los proyectos, el atributo Max-Age es más fácil de usar. |
| HttpOnly | Este atributo le indica al navegador que evite que el código JavaScript lea la cookie. Esto rara vez se configura para aplicaciones web que tienen código JavaScript del lado del cliente. |
| Max-Age=second | Este atributo especifica la cantidad de segundos hasta que caduca la cookie. Este atributo tiene prioridad sobre Expires. |
| Path=path | Este atributo especifica una ruta que debe estar en la URL para que el navegador incluya la cookie. |
| SameSite=policy | Este atributo le indica al navegador si la cookie debe incluirse en las solicitudes entre sitios, como se describe más adelante. Las opciones de política son Estricta, Laxa y Ninguna. |
| Secure | Cuando se configura esta opción, el navegador solo incluirá la cookie en las solicitudes HTTPS y no en las solicitudes HTTP simples. |

Dos de los atributos de las cookies requieren una explicación adicional. El atributo Domain se utiliza para ampliar el rango de solicitudes para las cuales el navegador incluirá una cookie. Si se envía una solicitud a https://users.acme.com, por ejemplo, las cookies que se devuelvan no se incluirán en las solicitudes a https://products.acme.com, lo que puede ser un problema para algunos proyectos. Esto se puede resolver con el atributo Domain, que se puede configurar en acme.com, y se le indica al navegador que incluya la cookie de manera más amplia.

El atributo SameSite se utiliza para controlar si la cookie se incluirá en las solicitudes que se originan fuera del sitio que creó la cookie, conocido como contexto de primera parte (*first-party*) o del mismo sitio (*same-site*). Las opciones para el atributo SameSite son: Strict, lo que significa que las cookies solo se incluyen para las solicitudes realizadas desde el mismo sitio web que creó la cookie, Lax, que le indica al navegador que incluya la cookie cuando siga un enlace, pero no para las solicitudes entre sitios, como el correo electrónico, y None, lo que significa que la cookie siempre se incluye.

Imaginemos que un usuario ha visitado previamente https://www.acme.com y ha recibido una cookie, tras lo cual navega a www.example.com. La respuesta de www.example.com contiene un enlace que lleva de vuelta a www.acme.com. Si la cookie se creó con la opción Strict, el navegador no enviará la cookie en la solicitud, pero se incluirá con la opción Lax. La opción None también hará que el navegador incluya la cookie y también permitirá que se incluya en solicitudes que se realicen dentro de frames o que sean para imágenes.

Al revisar las cookies creadas por el código del listado 2, puedes ver que se ha utilizado el atributo Max-Age para otorgarle a la cookie una vida útil de 300 segundos (5 minutos) y que la política SameSite está configurada como estricta (Strict), lo que significa que las cookies no se incluirán en solicitudes desde fuera del dominio de la cookie:

```
...
Set-Cookie: user=Alice; Max-Age=300; SameSite=Strict
...
```

La función setJsonCookie produce cookies con la misma configuración, pero acepta objetos arbitrarios que se serializan en formato JSON antes de usarse como valor de la cookie.

---

**Cómo evitar las cookies sin atributos de caducidad y Max-Age**

Una cookie que se crea sin los atributos Expires o Max-Age es una cookie de *sesión*, que es un término confuso porque este tipo de cookie no es especialmente útil para crear sesiones de usuario, un proceso que se demuestra más adelante en este documento. El nombre "cookies de sesión" significa que una cookie es válida para una sesión de navegación, lo que significa que se invalidan cuando el usuario cierra la ventana del navegador, por ejemplo.

Los navegadores han cambiado desde que se creó este tipo de cookie, y se deben evitar las cookies de sesión porque dejar que el navegador decida cuándo invalidar una cookie puede producir resultados inesperados, y las cookies pueden tener una vida útil larga e impredecible, especialmente ahora que los navegadores permiten a los usuarios resucitar las pestañas del navegador mucho después de que se hayan cerrado.

Las cookies siempre deben tener una vida útil fija con los atributos Expires o Max-Age.

---

## Recepción de cookies

El navegador incluye cookies en las solicitudes mediante el encabezado Cookie, que contiene uno o más pares name=value, separados por punto y coma (el carácter ;). Los atributos utilizados con el encabezado SetCookie no están incluidos, por lo que el encabezado se ve así:

```
...
Cookie: user=Alice; otherCookie=othervalue
...
```

El listado 3 define una función para analizar el encabezado y extraer las cookies individuales. También hay un método para analizar los valores de las cookies JSON.

Listado 3: Análisis (parsing) de cookies en el archivo cookies.ts en la carpeta src/server.

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

const setheaderName = "Set-Cookie";

export const setCookie = (resp: ServerResponse, name: string,
      val: string) => {
  let cookieVal: any[] = [`${name}=${val}; Max-Age=300; SameSite=Strict }`];
  if (resp.hasHeader(setheaderName)) {
    cookieVal.push(resp.getHeader(setheaderName));
  }
  resp.setHeader("Set-Cookie", cookieVal);
}

export const setJsonCookie = (resp: ServerResponse, name: string,
      val: any) => {
  setCookie(resp, name, JSON.stringify(val));
}

export const getCookie = (req: IncomingMessage,
      key: string): string | undefined => {
  let result: string | undefined = undefined;
  req.headersDistinct["cookie"]?.forEach(header => {
    header.split(";").forEach(cookie => {
      const { name, val }
        = /^(?<name>.*)=(?<val>.*)$/.exec(cookie)?.groups as any;
      if (name.trim() === key) {
        result = val;
      }
    })
  });
  return result;
}

export const getJsonCookie = (req: IncomingMessage, key: string) : any => {
  const cookie = getCookie(req, key);
  return cookie ? JSON.parse(cookie) : undefined;
}
```

La función getCookie utiliza funciones de expresión regular y procesamiento de cadenas de JavaScript para dividir la cadena de cookies y obtener el nombre y el valor para localizar una cookie específica. Este no es un enfoque eficiente porque el encabezado de cookies se procesa cada vez que se solicita una cookie, pero muestra cómo se puede manejar el encabezado y se mejorará más adelante en este documento.

**Configuración (setting) y lectura de cookies**

El listado 4 actualiza el código que maneja las solicitudes /form para configurar una cookie que realiza un seguimiento de las solicitudes del usuario. El contenido de la cookie se actualiza cada vez que se recibe una nueva solicitud, y el valor de la cookie se lee de cada solicitud y se agrega a los datos de contexto que se pasan a la plantilla que se utiliza para generar una respuesta.

Listado 4: Uso de cookies en el archivo forms.ts en la carpeta src/server.

```
import express, { Express } from "express";
import repository from "./data";
import { getJsonCookie, setJsonCookie } from "./cookies";

const rowLimit = 10;

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

export const registerFormRoutes = (app: Express) => {

  app.get("/form", async (req, resp) => {
    resp.render("age", {
      history: await repository.getAllResults(rowLimit),
      personalHistory: getJsonCookie(req, "personalHistory")
    });
  });

  app.post("/form", async (req, resp) => {
    const nextage = Number.parseInt(req.body.age)
      + Number.parseInt(req.body.years);

    await repository.saveResult({...req.body, nextage });

    let pHistory = [{
      name: req.body.name, age: req.body.age,
      years: req.body.years, nextage},
      ...(getJsonCookie(req, "personalHistory") || [])].splice(0, 5);

    setJsonCookie(resp, "personalHistory", pHistory);

    const context = {
      ...req.body, nextage,
      history: await repository.getAllResults(rowLimit),
```

```
            personalHistory: pHistory
        };
        resp.render("age", context);
    });
}
```

Se utiliza una cookie para almacenar los últimos cinco resultados creados para el usuario. Cada nueva solicitud POST crea un nuevo encabezado Set-Cookie en la respuesta, con un nuevo tiempo de expiración de cinco minutos. Si el usuario continúa enviando solicitudes, se crearán nuevas cookies, lo que extenderá efectivamente la sesión del usuario. Si no se realiza ninguna solicitud antes de que caduque la cookie, el navegador la descartará y no la incluirá en futuras solicitudes.

El listado 5 actualiza la vista parcial que muestra las consultas recientes para mostrar el historial personal cuando esté disponible.

Listado 5: Visualización de datos en el archivo history.handlebars en la carpeta templates/server/partials.

```
{{#if personalHistory }}
    <h4>Your History</h4>
    <table class="table table-sm table-striped my-2">
        {{#each personalHistory }}
            <tr>
                <td>{{ this.name }} </td>
                <td>{{ this.age }} </td>
                <td>{{ this.years }} </td>
                <td>{{ this.nextage }} </td>
            </tr>
        {{/each }}
    </table>
{{/if }}

<h4>Recent Queries</h4>

<table class="table table-sm table-striped my-2">
  <thead>
    <tr>
      <th>Name</th><th>Age</th><th>Years</th><th>Result</th>
    </tr>
  </thead>
  <tbody>
    {{#unless history }}
      <tr><td colspan="4">No data available</td></tr>
```

```
        {{/unless }}
        {{#each history }}
          <tr>
            <td>{{ this.name }} </td>
            <td>{{ this.age }} </td>
            <td>{{ this.years }} </td>
            <td>{{ this.nextage }} </td>
          </tr>
        {{/each }}
      </tbody>
    </table>
```

Los navegadores comparten cookies entre pestañas, por lo que la forma más confiable de probar los cambios en el ejemplo es abrir una pestaña del navegador normal y una pestaña de navegación privada o de incógnito. Navega a http://localhost:5000 con ambas pestañas y completa el formulario usando el mismo nombre pero diferentes edades y años. Envía los formularios y verás que cada pestaña del navegador tiene su propio historial, como se muestra en la figura 3.

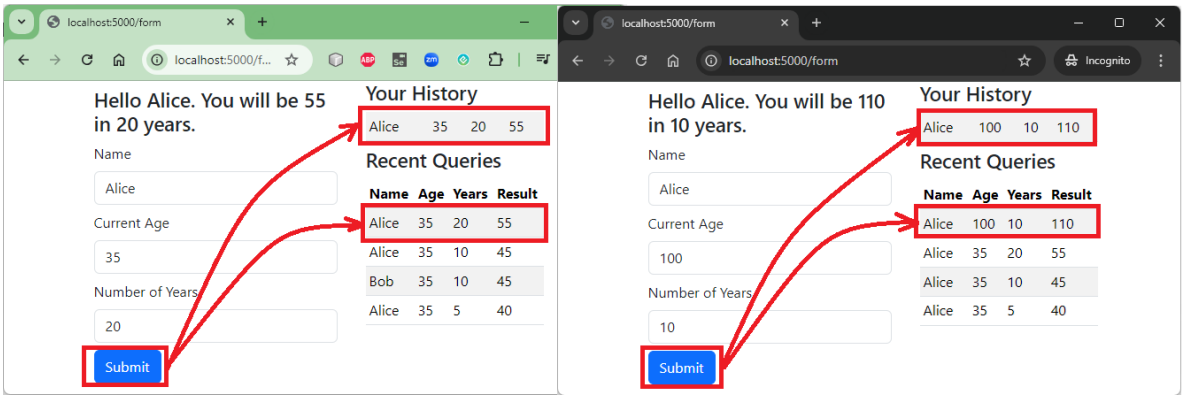


Figura 3: Uso de cookies para correlacionar solicitudes.

## Firma de cookies

Los usuarios pueden cambiar el contenido de las cookies y los navegadores facilitan la adición, eliminación y modificación de cookies. Las herramientas para desarrolladores de Chrome F12, por ejemplo, permiten editar las cookies en el panel **Application/Cookies**.

Esto significa que no se puede confiar en las cookies a menos que se pueda verificar su contenido para garantizar que no hayan sido alteradas. Agrega un archivo llamado cookies_signed.ts a la carpeta src/server con el contenido que se muestra en el listado 6.

Listado 6: El contenido del archivo cookies_signed.ts en la carpeta src/server.

```
import { createHmac, timingSafeEqual } from "crypto";

export const signCookie = (value: string, secret: string) => {
   return value + "." + createHmac("sha512", secret)
     .update(value).digest("base64url");
}

export const validateCookie = (value: string, secret: string) => {
   const cookieValue = value.split(".")[0];
   const compareBuf = Buffer.from(signCookie(cookieValue, secret));
   const candidateBuf = Buffer.from(value);
   if (compareBuf.length == candidateBuf.length &&
     timingSafeEqual(compareBuf, candidateBuf)) {
       return cookieValue;
   }
   return undefined;
}
```

Node.js proporciona una API de criptografía integral en el módulo crypto, que incluye compatibilidad con códigos de autenticación de mensajes basados en hash (HMACs, **Hash-based message authentication codes**), que son códigos hash creados utilizando una llave secreta que se puede utilizar para verificar datos. La función signCookie en el listado 6 utiliza la API de Node.js para crear un código hash que se puede utilizar como valor de cookie.

La función createHmac se utiliza para crear el generador de código hash, utilizando el algoritmo SHA-512 y la llave secreta:

```
...
createHmac("sha512", secret).update(value).digest("base64url");
..
```

El método update se utiliza para aplicar el algoritmo hash al valor de la cookie, y el método digest devuelve el código hash en la codificación URL Base64, lo que permite que el código hash se incluya de forma segura en la cookie. El resultado es el valor de los datos, seguido de un punto, seguido del código hash, que tendrá este aspecto:

```
...
myCookieData.hn5jneGWS_oBL7ww5IHZm9KuzfUwWnnDz01vhNc5xNMwb-kQnxb357Tp
...
```

Los códigos hash reales son más largos, pero lo importante es que el valor de la cookie no está cifrado y el usuario puede verlo. El usuario puede editar la cookie, pero el código hash permite detectar esos cambios.

Cuando se envía la cookie, el método validationCookie genera un nuevo código hash para el valor de la cookie y lo compara con el que se recibió en la cookie. Los códigos hash son unidireccionales, lo que significa que se validan generando un nuevo código hash para el valor de la cookie incluido en la solicitud HTTP y comparándolo con el código hash anterior.

El módulo de cifrado (crypto) de Node.js proporciona la función timingSafeEqual, que realiza una comparación byte a byte de dos objetos Buffer, que se crean a partir de los dos códigos hash que se van a comparar.

El usuario puede modificar el valor de la cookie, pero no tiene la llave secreta necesaria para generar un código hash válido para el valor modificado. Si el código hash recibido de la solicitud no coincide, los datos de la cookie se descartan. El listado 7 actualiza las funciones setCookie y getCookie para que todas las cookies creadas por la aplicación estén firmadas.

**Precaución**

Ten cuidado de no enviar llaves secretas a repositorios de código fuente públicos, como GitHub. Un enfoque es definir datos confidenciales en archivos .env, que se pueden excluir de las confirmaciones de código.

Listado 7: Firma de cookies en el archivo cookies.ts en la carpeta src/server.

```
import { IncomingMessage, ServerResponse } from "http";
import { signCookie, validateCookie } from "./cookies_signed";

const setheaderName = "Set-Cookie";
const cookieSecret = "mysecret";

export const setCookie = (resp: ServerResponse, name: string,
    val: string) => {
  const signedCookieVal = signCookie(val, cookieSecret);
  let cookieVal: any[] =
    [`${name}=${signedCookieVal}; Max-Age=300; SameSite=Strict`];
  if (resp.hasHeader(setheaderName)) {
    cookieVal.push(resp.getHeader(setheaderName));
  }
  resp.setHeader("Set-Cookie", cookieVal);
}

export const setJsonCookie = (resp: ServerResponse, name: string,
    val: any) => {
  setCookie(resp, name, JSON.stringify(val));
}
```

```
export const getCookie = (req: IncomingMessage,
      key: string): string | undefined => {
  let result: string | undefined = undefined;
  req.headersDistinct["cookie"]?.forEach(header => {
    header.split(";").forEach(cookie => {
      const { name, val }
        = /^(?<name>.*)=(?<val>.*)$/.exec(cookie)?.groups as any;
      if (name.trim() === key) {
        result = validateCookie(val, cookieSecret)
      }
    })
  });
  return result;
}

export const getJsonCookie = (req: IncomingMessage, key: string) : any => {
  const cookie = getCookie(req, key);
  return cookie ? JSON.parse(cookie) : undefined;
}
```

No hay cambios en el comportamiento de la aplicación, pero si utilizas las herramientas de desarrollo de tu navegador para modificar una cookie, encontrarás que se ignora cuando el navegador envía una solicitud.

## Uso de un paquete para administrar cookies

Los ejemplos anteriores no solo demostraron cómo se pueden utilizar los encabezados Set-Cookie y Cookie, sino que también mostraron que trabajar directamente con cookies puede ser complicado. Express incluye soporte para analizar cookies, así como para generar JSON y cookies firmadas, sin la necesidad de formatear o analizar manualmente los encabezados.

El análisis de cookies.cpp se realiza utilizando un componente de middleware, que no está incluido en el paquete principal de Express. Ejecuta los comandos que se muestran en el listado 8 en la carpeta part2app para instalar el paquete de análisis y la descripción TypeScript de su API.

Listado 8: Instalación del paquete de middleware de cookies.

```
npm install cookie-parser@1.4.6
npm install --save-dev @types/cookie-parser@1.4.6
```

El listado 9 habilita el middleware de análisis de cookies y especifica la llave secreta que se utilizará para las cookies firmadas.

Listado 9: Aplicación del middleware en el archivo forms.ts en la carpeta src/server.

```
import express, { Express } from "express";
import repository from "./data";
import { getJsonCookie, setJsonCookie } from "./cookies";
import cookieMiddleware from "cookie-parser";

const rowLimit = 10;

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

export const registerFormRoutes = (app: Express) => {

  app.get("/form", async (req, resp) => {
    resp.render("age", {
      history: await repository.getAllResults(rowLimit),
      personalHistory: getJsonCookie(req, "personalHistory")
    });
  });

  app.post("/form", async (req, resp) => {
    const nextage = Number.parseInt(req.body.age)
      + Number.parseInt(req.body.years);

    await repository.saveResult({...req.body, nextage });

    let pHistory = [{
      name: req.body.name, age: req.body.age,
      years: req.body.years, nextage},
      ...(getJsonCookie(req, "personalHistory") || [])].splice(0, 5);

    setJsonCookie(resp, "personalHistory", pHistory);

    const context = {
      ...req.body, nextage,
      history: await repository.getAllResults(rowLimit),
      personalHistory: pHistory
    };
    resp.render("age", context);
```

```
    });
}
```

El middleware llena la propiedad cookies del objeto Request para las cookies regulares y la propiedad signedCookies para las cookies firmadas. Las cookies se configuran utilizando una propiedad cookie definida por el objeto Response. El listado 10 utiliza estas características para generar las cookies que requiere la aplicación y agrega un parámetro al método setCookie para permitir que se anulen las opciones de cookies predeterminadas.

Listado 10: Uso de las características de cookies de Express en el archivo cookies.ts en la carpeta src/server.

```
//import { IncomingMessage, ServerResponse } from "http";
//import { signCookie, validateCookie } from "./cookies_signed";
import { CookieOptions, Request, Response } from "express";

//const setheaderName = "Set-Cookie";
//const cookieSecret = "mysecret";

export const setCookie = (resp: Response, name: string, val: string,
        opts?: CookieOptions) => {
    resp.cookie(name, val, {
        maxAge: 300 * 1000,
        sameSite: "strict",
        signed: true,
        ...opts
    });
}

export const setJsonCookie = (resp: Response, name: string, val: any) => {;
    setCookie(resp, name, JSON.stringify(val));
}

export const getCookie = (req: Request, key: string): string | undefined => {
    return req.signedCookies[key];
}

export const getJsonCookie = (req: Request, key: string) : any => {
    const cookie = getCookie(req, key);
    return cookie ? JSON.parse(cookie) : undefined;
}
```

Express y el middleware de cookies se encargan de crear el encabezado Set-Cookie en las respuestas y analizar los encabezados de Cookie en las solicitudes. El método

Response.cookie se utiliza para crear cookies y acepta un nombre, un valor y un objeto de configuración. El objeto de configuración tiene propiedades que corresponden a los atributos de cookies descritos en la tabla 3, aunque existen algunas rarezas. Por ejemplo, la configuración maxAge se especifica en milisegundos, en lugar de los segundos que utiliza el atributo Max-Age (por eso el valor en el listado 10 se multiplica por 1000).

El objeto de configuración aceptado por el método cookie admite una propiedad firmada, que permite la firma de cookies. La llave se obtiene de la configuración utilizada para configurar el middleware de cookies, lo cual es otra rareza, pero funciona de todos modos. Las cookies se firman utilizando un HMAC, de manera similar al código personalizado.

Las cookies recibidas en las solicitudes están disponibles a través de las propiedades Request.cookies y Request.signedCookies, que devuelven objetos cuyas propiedades corresponden a los nombres de las cookies en la solicitud. Las cookies firmadas se detectan fácilmente porque el método Response.cookie crea valores de cookies firmadas con el prefijo s., y los valores se verifican automáticamente utilizando la llave secreta con la que se configuró el middleware.

Los cambios en el listado 10 no cambian el comportamiento de la aplicación, pero las cookies tienen un formato diferente y las cookies creadas con el código personalizado no pasarán la verificación.

## Uso de sesiones

Las cookies son adecuadas para almacenar pequeñas cantidades de datos, pero esos datos deben enviarse a la aplicación con cada solicitud, y cualquier cambio en esos datos debe firmarse y enviarse en la respuesta.

Una alternativa es que la aplicación almacene los datos e incluya solo una referencia a esos datos en la cookie. Esto permite almacenar mayores cantidades de datos sin que esos datos se incluyan en cada solicitud y respuesta.

Los datos de sesión se pueden almacenar como un conjunto de pares llave/valor, lo que facilita el uso de objetos JavaScript para representar datos. Comenzaremos creando un sistema de sesión basado en memoria y luego presentaremos el almacenamiento persistente con una base de datos, utilizando una capa de repositorio para facilitar la transición. Crea la carpeta src/server/sessions y agrégale un archivo llamado repositorio.ts con el contenido que se muestra en el listado 11.

Listado 11: El contenido del archivo repositorio.ts en la carpeta src/server/sessions.

```
export type Session = {
  id: string,
  data: { [key: string]: any }
}

export interface SessionRepository {

  createSession() : Promise<Session>;

  getSession(id: string): Promise<Session | undefined>;

  saveSession(session: Session, expires: Date): Promise<void>;

  touchSession(session: Session, expires: Date) : Promise<void>
}
```

La interfaz SessionRepository define métodos para crear una sesión, recuperar una sesión almacenada previamente y guardar o actualizar una sesión. El tipo Session define los requisitos mínimos para una Session, lo que implica un ID y una propiedad de datos a la que se pueden asignar datos arbitrarios indexados por valores de cadena.

Para crear una implementación basada en memoria de la interfaz, agrega un archivo llamado memory_repository. ts a la carpeta src/server/sessions con el contenido que se muestra en el listado 12.

Listado 12: El contenido del archivo memory_repository.ts en la carpeta src/server/sessions.

```
import { Session, SessionRepository } from "./repository";
import { randomUUID } from "crypto";

type SessionWrapper = {
   session: Session,
   expires: Date
}

export class MemoryRepository implements SessionRepository {
   store = new Map<string, SessionWrapper>();

   async createSession(): Promise<Session> {
      return { id: randomUUID(), data: {} };
   }

   async getSession(id: string): Promise<Session | undefined> {
```

```
            const wrapper = this.store.get(id);
            if (wrapper && wrapper.expires > new Date(Date.now())) {
                return structuredClone(wrapper.session)
            }
        }

        async saveSession(session: Session, expires: Date): Promise<void> {
            this.store.set(session.id, { session, expires });
        }

        async touchSession(session: Session, expires: Date): Promise<void> {
            const wrapper = this.store.get(session.id);
            if (wrapper) {
                wrapper.expires = expires;
            }
        }
    }
```

El paquete de cifrado (crypto) Node.js define la función randomUUID, que genera identificadores únicos que son adecuados para su uso como identificadores de sesión. El resto de la implementación utiliza un Mapa para almacenar objetos Session, que se verifican para verificar su vencimiento cuando se leen.

Un punto a destacar es que el método getSession no devuelve la Session del almacén, sino que crea un nuevo objeto, como este:

```
...
if (wrapper && wrapper.expires > new Date(Date.now())) {
    return structuredClone(wrapper.session)
}
...
```

La función structuredClone es parte de la API estándar de JavaScript y crea una copia profunda de un objeto. Los datos de Session solo se deben modificar para las solicitudes POST porque los otros métodos HTTP son idempotentes y la creación de nuevos objetos facilita el descarte de los cambios que se realizan accidentalmente para otros métodos HTTP, lo que verás en la siguiente sección.

Este es un problema solo cuando se almacenan estados como objetos JavaScript, donde el objeto Session asociado con la solicitud es el mismo que el del almacén. No surge cuando los datos de Session se almacenan en una base de datos.

## Creación del middleware de Session

Las Sessions deben almacenarse después de que se haya generado la respuesta para que no se pierdan los cambios realizados en los datos de Session, y eso se puede hacer más fácilmente creando un componente de middleware Express. Agrega un archivo middleware.ts a la carpeta src/server/sessions con el contenido que se muestra en el listado 13.

Listado 13: El contenido del archivo middleware.ts en la carpeta src/server/sessions.

```
import { Request, Response, NextFunction } from "express";
import { SessionRepository, Session } from "./repository";
import { MemoryRepository } from "./memory_repository";
import { setCookie, getCookie } from "../cookies";

const session_cookie_name = "custom_session";
const expiry_seconds = 300;

const getExpiryDate = () => new Date(Date.now() + (expiry_seconds * 1_000));
export const customSessionMiddleware = () => {
   const repo: SessionRepository = new MemoryRepository();
   return async (req: Request, resp: Response, next: NextFunction) => {

      const id = getCookie(req, session_cookie_name);

      const session = (id ? await repo.getSession(id) : undefined)
                  ?? await repo.createSession();

      (req as any).session = session;

      setCookie(resp, session_cookie_name, session.id, {
         maxAge: expiry_seconds * 1000
      })

      resp.once("finish", async () => {
         if ( Object.keys(session.data).length > 0) {
           if (req.method == "POST") {
              await repo.saveSession(session, getExpiryDate());
           } else {
              await repo.touchSession(session, getExpiryDate());
           }
         }
      })

      next();
```

```
    }
}
```

Este componente de middleware lee una cookie que contiene un ID de sesión y lo utiliza para obtener la sesión del repositorio y asociarla con el objeto Request agregando una propiedad denominada session. Si no hay ninguna cookie o no se puede encontrar ninguna sesión con el ID, se inicia una nueva sesión.

La sesión solo se puede almacenar de forma segura una vez que se ha generado la respuesta y cuando se tiene la certeza de que no se realizarán más cambios. El evento finish se activa una vez que se completa una respuesta y se utiliza el método once para controlar el evento y almacenar la sesión.

Las sesiones solo se almacenan para las solicitudes HTTP POST y cuando se han asignado propiedades al objeto de datos. Para otros métodos HTTP, se utiliza el método touchSession para extender el tiempo de expiración de la sesión, pero los datos de la sesión no se almacenan.

Actualizar la expiración de la sesión después de cada solicitud crea una expiración variable, lo que significa que la sesión puede seguir siendo válida indefinidamente. Este es el enfoque más común porque significa que las sesiones son válidas mientras el usuario esté activo y se agotarán después de un período de inactividad.

## Uso de la función de sesión

El componente de middleware agrega una propiedad de sesión a las solicitudes, pero esta no es parte del tipo estándar de Request de Express y no es conocida por el compilador de TypeScript. Hay dos buenas maneras de resolver este problema: una función auxiliar que lea la propiedad de sesión o un nuevo tipo que extienda el que proporciona Express. Agrega un archivo llamado session_helpers.ts a la carpeta src/server/sessions con el contenido que se muestra en el listado 14.

Listado 14: El contenido del archivo session_helpers.ts en la carpeta src/server/sessions.

```
import { Request } from "express";
import { Session } from "./repository";

export const getSession = (req: Request): Session => (req as any).session;

declare global {
  module Express {
```

```
        interface Request {
          session: Session
        }
      }
    }
```

La función getSession recibe un objeto Request y devuelve la propiedad de sesión usando *as any* para evitar las verificaciones de tipo de TypeScript. La palabra clave *declare* se usa para indicarle a TypeScript que la interfaz Request tiene una propiedad adicional.

De los dos enfoques, el preferido por muchos desarrolladores es el de la función auxiliar, que no es tan elegante, pero que se entiende más fácilmente y hace obvio cómo se obtiene el objeto Session. El listado 15 aplica ambos enfoques para cambiar de almacenar datos de sesión en la cookie a usar el repositorio de sesiones.

Listado 15: Uso del repositorio de sesiones en el archivo forms.ts en la carpeta src/server.

```
import express, { Express } from "express";
import repository from "./data";
import { getJsonCookie, setJsonCookie } from "./cookies";
import cookieMiddleware from "cookie-parser";
import { customSessionMiddleware } from "./sessions/middleware";
import { getSession } from "./sessions/session_helpers";

const rowLimit = 10;

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

export const registerFormRoutes = (app: Express) => {

  app.get("/form", async (req, resp) => {
    resp.render("age", {
      history: await repository.getAllResults(rowLimit),
      personalHistory: getSession(req).data.personalHistory
    });
  });

  app.post("/form", async (req, resp) => {
    const nextage = Number.parseInt(req.body.age)
      + Number.parseInt(req.body.years);
```

```
    await repository.saveResult({...req.body, nextage });

    req.session.data.personalHistory = [{
      name: req.body.name, age: req.body.age,
      years: req.body.years, nextage},
      ...(req.session.data.personalHistory || [])].splice(0, 5);

    const context = {
      ...req.body, nextage,
      history: await repository.getAllResults(rowLimit),
      personalHistory: req.session.data.personalHistory
    };
    resp.render("age", context);
  });
}
```

Los cambios habilitan el middleware de sesión y almacenan el historial del usuario usando la nueva característica de sesión. Una vez más, no hay cambios en la forma en que se comporta la aplicación, porque los cambios son invisibles para el usuario.

A medida que se envía el formulario, la cookie enviada por el navegador se utiliza para cargar los datos de sesión desde el repositorio, que se utilizan en la respuesta, como se muestra en la figura 4.

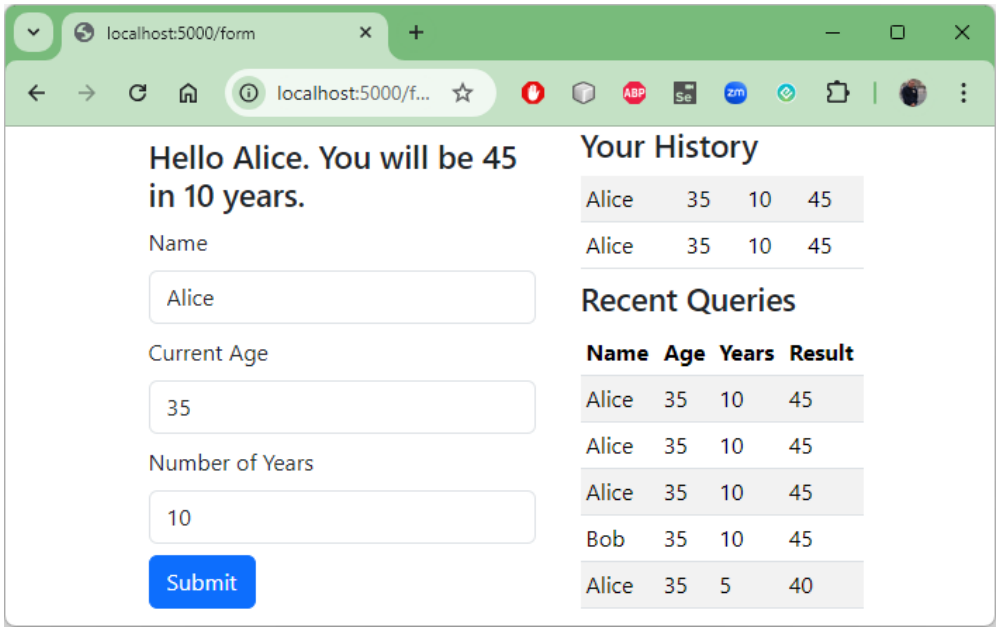


Figura 4: Uso de datos de sesión.

## Almacenamiento de datos de sesión en una base de datos

Almacenar datos de sesión en la memoria es una buena manera de entender cómo encajan las piezas, pero no es ideal para proyectos reales donde generalmente se requiere un almacenamiento más persistente.

El enfoque convencional es almacenar datos de sesión en una base de datos, lo que garantiza que las sesiones sean persistentes y permite una gran cantidad de sesiones sin agotar la memoria del sistema.

Agrega un archivo llamado orm_models.ts a la carpeta src/server/sessions, con el contenido que se muestra en el listado 16.

Listado 16: El contenido del archivo orm_models.ts en la carpeta src/server/sessions.

```
import { DataTypes, InferAttributes, InferCreationAttributes, Model,
    Sequelize } from "sequelize";

export class SessionModel extends Model<InferAttributes<SessionModel>,
    InferCreationAttributes<SessionModel>> {
    declare id: string
    declare data: any;
    declare expires: Date
}

export const initializeModel = (sequelize: Sequelize) => {

    SessionModel.init({
        id: { type: DataTypes.STRING, primaryKey: true },
        data: { type: DataTypes.JSON },
        expires: { type: DataTypes.DATE }
    }, { sequelize });
}
```

Una sola clase de modelo puede representar una sesión y los identificadores generados por la función crypto.randomUUID se pueden usar como llaves principales. Sequelize tiene un buen soporte para trabajar con fechas de JavaScript y serializará y deserializará automáticamente los objetos cuando el tipo de una columna sea DataTypes.JSON.

Para crear un repositorio de sesiones, agrega un archivo llamado orm_repository.ts en la carpeta src/server/sessions, con el contenido que se muestra en el listado 17.

Listado 17: El contenido del archivo orm_repository.ts en la carpeta src/server/sessions.

```
import { Op, Sequelize } from "sequelize";
import { Session, SessionRepository } from "./repository";
import { SessionModel, initializeModel } from "./orm_models";
import { randomUUID } from "crypto";

export class OrmRepository implements SessionRepository {
  sequelize: Sequelize;

      constructor() {
    this.sequelize = new Sequelize({
      dialect: "sqlite",
      storage: "orm_sessions.db",
      logging: console.log,
      logQueryParameters: true
    });
    this.initModelAndDatabase();
  }

  async initModelAndDatabase() : Promise<void> {
    initializeModel(this.sequelize);
    await this.sequelize.drop();
    await this.sequelize.sync();
  }

  async createSession(): Promise<Session> {
    return { id: randomUUID(), data: {} };
  }

  async getSession(id: string): Promise<Session | undefined> {
    const dbsession = await SessionModel.findOne({
      where: { id, expires: { [Op.gt] : new Date(Date.now()) }}
    });
    if (dbsession) {
      return { id, data: dbsession.data };
    }
  }

  async saveSession(session: Session, expires: Date): Promise<void> {
    await SessionModel.upsert({
      id: session.id,
      data: session.data,
      expires
    });
  }
```

```
    async touchSession(session: Session, expires: Date): Promise<void> {
      await SessionModel.update({ expires }, { where: { id: session.id } });
    }
  }
```

**Nota**

El método initModelAndDatabase en el listado 17 llama al método drop, que restablecerá la base de datos cada vez que se inicie o reinicie la aplicación. Esto no se debe hacer en un proyecto real, pero es útil para un ejemplo y garantiza que cualquier cambio en los archivos de código se refleje en la base de datos.

El repositorio es similar al creado para los datos de la aplicación, pero hay un par de puntos que muestran cómo un ORM (Object Relational Mapping) como Sequelize puede simplificar el manejo de una base de datos, aunque con un código JavaScript complicado. El método getSession consulta la base de datos para encontrar una fila con una llave principal determinada y una fecha de vencimiento futura, lo que se hace utilizando el método findOne y una expresión where, como esta:

```
...
const dbsession = await SessionModel.findOne({
    where: { id, expires: { [Op.gt] : new Date(Date.now()) }}
});
...
```

El valor Op.gt representa una comparación mayor y permite que la búsqueda coincida con filas donde la fecha almacenada en la columna expires sea mayor que la fecha actual. Esta no es la forma más natural de expresar consultas, pero funciona y permite que las consultas se expresen sin necesidad de escribir SQL.

El método upsert de Sequelize se utiliza para actualizar una fila de datos si existe e insertar una si no existe, lo que facilita la implementación del método saveSession. El método touchSession se implementa con el método update, que permite actualizar columnas específicas.

**Nota**

No hemos agregado ningún soporte para eliminar sesiones vencidas en este documento. Como regla, evitamos eliminar cualquier dato automáticamente porque es fácil que las cosas salgan mal. El espacio de almacenamiento es relativamente asequible, pero si necesitas administrar activamente el tamaño de la base de datos de la sesión, entonces hacer una copia de seguridad antes de una limpieza manual es una opción más segura.

El paso final es actualizar el middleware de la sesión para usar el nuevo repositorio, como se muestra en el listado 18.

Listado 18: Cambio de repositorio en el archivo middleware.ts en src/server/sessions.

```
import { Request, Response, NextFunction } from "express";
import { SessionRepository, Session } from "./repository";
//import { MemoryRepository } from "./memory_repository";
import { setCookie, getCookie } from "../cookies";
import { OrmRepository } from "./orm_repository";

const session_cookie_name = "custom_session";
const expiry_seconds = 300;

const getExpiryDate = () => new Date(Date.now() + (expiry_seconds * 1_000));

export const customSessionMiddleware = () => {
   //const repo: SessionRepository = new MemoryRepository();
         const repo: SessionRepository = new OrmRepository();
   return async (req: Request, resp: Response, next: NextFunction) => {

      const id = getCookie(req, session_cookie_name);

      const session = (id ? await repo.getSession(id) : undefined)
                  ?? await repo.createSession();

      (req as any).session = session;

      setCookie(resp, session_cookie_name, session.id, {
         maxAge: expiry_seconds * 1000
      })

      resp.once("finish", async () => {
         if ( Object.keys(session.data).length > 0) {
          if (req.method == "POST") {
             await repo.saveSession(session, getExpiryDate());
          } else {
             await repo.touchSession(session, getExpiryDate());
          }
         }
      })

      next();
   }
}
```

No se requiere ningún otro cambio para usar la base de datos porque el nuevo repositorio implementa la misma interfaz que el anterior.

La diferencia clave es que verás que las consultas de la base de datos se registran en la consola de Node.js mientras se ejecuta la aplicación, comenzando con la declaración que crea la tabla de la base de datos:

```
...
Executing (default): CREATE TABLE IF NOT EXISTS `SessionModels` (`id`
VARCHAR(255)
    PRIMARY KEY, `data` JSON, `expires` DATETIME, `createdAt` DATETIME NOT
NULL,
    `updatedAt` DATETIME NOT NULL);
...
```

No se requirió SQL para preparar o consultar la base de datos y el proceso de creación y análisis de JSON se maneja automáticamente.

## Uso de un paquete para sesiones

Ahora que comprendes cómo funcionan las sesiones, es momento de reemplazar el código personalizado con un paquete de sesiones estándar, como el que ofrece Express. Ejecuta los comandos que se muestran en el listado 19 en la carpeta part2app para instalar el paquete de sesiones, el paquete de descripción de tipo para su API y un paquete que almacena sesiones en una base de datos mediante Sequelize. (Existe una amplia gama de opciones de base de datos para el paquete express-sessions, que se describe en: https://github.com/expressjs/session).

Listado 19: Instalación de paquetes.

```
npm install express-session@1.17.3
npm install connect-session-sequelize@7.1.7
npm install --save-dev @types/express-session@1.17.10
```

El listado 20 prepara la aplicación para utilizar el paquete de sesión y el paquete de almacenamiento.

Listado 20: Uso del paquete de sesión en el archivo session_helpers.ts en la carpeta src/server/sessions.

```
import { Request } from "express";
//import { Session } from "./repository";
import session, { SessionData } from "express-session";
```

```
import sessionStore from "connect-session-sequelize";
import { Sequelize } from "sequelize";
import { Result } from "../data/repository";

export const getSession = (req: Request): SessionData => (req as any).session;

//declare global {
//    module Express {
//       interface Request {
//          session: Session
//       }
//    }
//}

declare module "express-session" {
   interface SessionData {
      personalHistory: Result[];
   }
}

export const sessionMiddleware = () => {

   const sequelize = new Sequelize({
      dialect: "sqlite",
      storage: "pkg_sessions.db"
   });

   const store = new (sessionStore(session.Store))({
      db: sequelize
   });

   store.sync();

   return session({
      secret: "mysecret",
      store: store,
      cookie: { maxAge: 300 * 1000, sameSite: "strict" },
      resave: false, saveUninitialized: false
   })
}
```

Se requieren ajustes para utilizar el paquete, incluido comentar la declaración *declare* que agrega la propiedad Request.session porque hay una declaración similar definida por el paquete express-session.

Se requiere una nueva declaración *declare* para agregar propiedades personalizadas al objeto SessionData, que es el tipo que se utiliza para representar los datos de sesión por parte del paquete. Hay un tipo Session, pero tiene un propósito similar al tipo contenedor empleado por el código personalizado. En este caso, se agregó una propiedad personHistory para minimizar los cambios necesarios para utilizar el paquete.

La función sessionMiddleware crea un objeto Sequelize que utiliza SQLite y lo utiliza para crear un almacén para los datos de sesión utilizando el paquete connect-session-sequelize. El método sync se llama para inicializar la base de datos y la exportación predeterminada del paquete express-session se utiliza para crear un componente de middleware.

Las opciones de configuración para el almacén de sesiones se describen en https://github.com/expressjs/session, pero la configuración en el listado 20 especifica la llave secreta para firmar cookies, el almacén Sequelize y la configuración de cookies para que el paquete se comporte de la misma manera que el código personalizado. Se requieren pequeños cambios para usar el paquete de sesión, como se muestra en el listado 21.

**Nota**

El paquete cookie-parser se puede usar en la misma aplicación que el paquete expresssession, pero debes asegurarte de que ambos estén configurados con la misma llave secreta.

Listado 21: Uso del paquete de sesión en el archivo forms.ts en la carpeta src/server.

```
import express, { Express } from "express";
import repository from "./data";
import { getJsonCookie, setJsonCookie } from "./cookies";
import cookieMiddleware from "cookie-parser";
import { customSessionMiddleware } from "./sessions/middleware";
import { getSession, sessionMiddleware } from "./sessions/session_helpers";

const rowLimit = 10;

export const registerFormMiddleware = (app: Express) => {
   app.use(express.urlencoded({extended: true}));
   app.use(cookieMiddleware("mysecret"));
   //app.use(customSessionMiddleware());
   app.use(sessionMiddleware());
}

export const registerFormRoutes = (app: Express) => {

   app.get("/form", async (req, resp) => {
```

```
    resp.render("age", {
      history: await repository.getAllResults(rowLimit),
      personalHistory: getSession(req).personalHistory
    });
  });

  app.post("/form", async (req, resp) => {
    const nextage = Number.parseInt(req.body.age)
      + Number.parseInt(req.body.years);

    await repository.saveResult({...req.body, nextage });

    req.session.personalHistory = [{
      id: 0, name: req.body.name, age: req.body.age,
      years: req.body.years, nextage},
      ...(req.session.personalHistory || [])].splice(0, 5);

    const context = {
      ...req.body, nextage,
      history: await repository.getAllResults(rowLimit),
      personalHistory: req.session.personalHistory
    };
    resp.render("age", context);
  });
}
```

Los cambios reemplazan el middleware personalizado y leen la propiedad personalHistory directamente en el objeto devuelto por la propiedad de sesión. El esquema de la base de datos que se utiliza para almacenar sesiones es diferente, lo que se puede ver en las sentencias SQL que se escriben en la consola de Node.js, pero, por lo demás, el comportamiento de la aplicación no cambia.

## Resumen

En este documento/práctica, se explicó cómo una aplicación puede utilizar cookies para correlacionar solicitudes HTTP y crear una experiencia de usuario con estado en un protocolo sin estado:

- Las cookies se crean añadiendo el encabezado Set-Cookie a las respuestas.
- Los navegadores incluyen cookies en las solicitudes con el encabezado Cookie.

- Las cookies se configuran utilizando atributos de cookies, incluida la configuración de un tiempo de expiración, después del cual el navegador ya no incluirá la cookie en las solicitudes.
- Las cookies se pueden firmar, lo que revela cuándo se han modificado.
- Las cookies se pueden utilizar para almacenar pequeñas cantidades de datos, pero estos datos deben transferirse repetidamente entre el navegador y el servidor.
- Las cookies también se pueden utilizar para almacenar identificadores de sesión, que se utilizan para cargar datos almacenados por el servidor. Esto hace que el servidor sea más complicado, pero significa que solo se transfiere el identificador entre el navegador y el servidor.

En el próximo documento/práctica, se describirá cómo se pueden utilizar los servicios web RESTful para proporcionar datos a los clientes sin incluir HTML.