

Tema 3. Opciones avanzadas de Express.js

3.2. Autenticación de usuarios

A la hora de proteger ciertas rutas o secciones de una aplicación web, podemos utilizar diversos mecanismos. En lo que a las aplicaciones web "tradicionales" se refiere (es decir, aquellas que sirven contenido visible en un navegador, como por ejemplo, y sobre todo, contenido HTML), el mecanismo de autenticación por antonomasia es la autenticación basada en sesiones.

Sin embargo, cuando queremos extender la aplicación web más allá del uso de un navegador, y permitir que otros tipos de clientes se conecten al *backend* (por ejemplo, aplicaciones de escritorio, o aplicaciones móviles), la autenticación basada en sesiones se queda corta o no sirve, y es necesario recurrir a otros mecanismos más universales, como la autenticación por token.

Veremos en esta sesión ambos mecanismos, comentando las particularidades de cada uno, y viendo cómo configurarlos en aplicaciones Express.

3.2.1. Autenticación basada en sesiones

La autenticación basada en sesiones permite autenticar usuarios en aplicaciones web basadas en navegadores, y "recordar" el usuario que se validó en sus sucesivas visitas. Para ello, utilizan las **sesiones**, que comprenden un conjunto de interacciones de un cliente con un servidor en un determinado período. Cuando abrimos un navegador y accedemos a una web, automáticamente se inicia la sesión en dicha web, y mientras no cerremos el navegador o la sesión manualmente, la aplicación recuerda (o puede recordar, si quiere) que ya hemos accedido, y los pasos que hemos ido dando en la actual sesión.

Cuando intentamos acceder a una zona restringida de ciertas webs, como por ejemplo nuestra página personal de una entidad bancaria, o los comentarios en un foro, la aplicación nos pide que nos validemos. Cuando introducimos un login y password, ésta los coteja con los que tenga almacenados y, si son correctos, almacena en la sesión datos sobre nuestro usuario, como por ejemplo, y sobre todo, nuestro *nick* o *login*, y el perfil de usuario que tenemos en la web (es decir, el rol: si somos administradores, editores, visitantes, etc). Así, para cada nueva petición que hagamos en esa misma sesión, el servidor comprueba quiénes somos y qué rol tenemos, y en función de eso, nos permite hacer unas cosas u otras. Al finalizar, podemos cerrar la sesión (*logout*), y borrar los datos que se hayan guardado en ella de nuestra visita.

En esta sesión utilizaremos un ejemplo sencillo que puedes descargar, sobre el que probaremos dicha autenticación.

3.2.1.1. Definición de sesiones en Express

Para poder trabajar con sesiones en Express, vamos a instalar el módulo *express-session*. Es un *middleware* que permite, en cada petición que requiera una comprobación, determinar si el usuario ya se ha validado y con qué credenciales, antes de dejarle acceder a lo que busca o no.

Así que lo primero que haremos será instalar el módulo:

```
npm install express-session
```

Después, lo incorporamos a nuestro servidor Express junto con el resto de módulos:

```
const express = require('express');  
const session = require('express-session');  
...
```

A continuación, configuramos la sesión dentro de la aplicación Express:

```
let app = express();  
...  
app.use(session({  
  secret: '1234',  
  resave: true,  
  saveUninitialized: false  
}));
```

Los parámetros de configuración que hemos empleado son:

- **secret**: una clave de cifrado para la sesión, que se empleará para enviarla cifrada entre cliente y servidor.
- **resave**: se emplea para refrescar la sesión con cada nuevo acceso, de forma que mientras sigamos accediendo a la aplicación dentro del tiempo de caducidad establecido para la sesión, éste se renueva automáticamente
- **saveUninitialized**: sirve para guardar sesiones aunque no se hayan completado. Se utiliza si queremos almacenar en sesión datos de usuarios que no se hayan validado, por ejemplo. En nuestro caso no habilitaremos esta opción.

3.2.1.2. Validación

En todo proceso de autenticación debe haber una validación previa, donde el usuario envíe sus credenciales y se cotejen con las existentes en la base de datos, antes de dejarle acceder.

Vamos a suponer, por simplicidad, que tenemos los usuarios cargados en un array, con su nombre de usuario y su password:

```
const usuarios = [  
  { usuario: 'nacho', password: '12345' },  
  { usuario: 'pepe', password: 'pepe111' }  
];
```

Ahora tendríamos que definir una ruta que, normalmente por POST, recogiera las credenciales que envía el usuario y las cotejara con ese array. Si concuerda con algún usuario almacenado, se guarda en la sesión el nombre del usuario que accedió al sistema, y se puede redirigir a alguna página de inicio. En caso contrario, se puede redirigir a una página de login:

```
app.post('/login', (req, res) => {  
  let login = req.body.login;  
  let password = req.body.password;  
  
  let existeUsuario = usuarios.filter(usuario =>  
    usuario.usuario == login && usuario.password == password);  
  
  if (existeUsuario.length > 0)  
  {  
    req.session.usuario = existeUsuario[0].usuario;  
    res.render('index');  
  } else {  
    res.render('login',  
      {error: "Usuario o contraseña incorrectos"});  
  }  
});
```

3.2.1.3. Autenticación

Una vez validado el usuario, debemos definir una función *middleware* que se encargará de aplicarse en cada ruta que queramos proteger. Lo que hará será comprobar si hay algún usuario en sesión. En caso afirmativo, dejará pasar la petición. De lo contrario, enviará a la página de validación o *login*, por ejemplo.

```
let autenticacion = (req, res, next) => {
  if (req.session && req.session.usuario)
    return next();
  else
    res.render('login');
};
```

Sólo nos queda aplicar este *middleware* en cada ruta que requiera validación por parte del usuario. Esto se hace en la misma llamada a *get*, *post*, *put* o *delete*:

```
app.get('/protegido', autenticacion, (req, res) => {
  res.render('protegido');
});
```

Notar que pasamos como segundo parámetro el *middleware* de autenticación. Si pasa ese filtro, se ejecutará el código del `get`. En caso contrario, el *middleware* está configurado para renderizar la vista de login.

3.2.1.4. Definiendo roles

Nuestra aplicación también puede tener distintos roles para los usuarios registrados. Por ejemplo, podemos tener administradores y usuarios normales. Esto se suele definir con un campo extra en la información de los usuarios:

```
const usuarios = [
  { usuario: 'nacho', password: '12345', rol: 'admin' },
  { usuario: 'pepe', password: 'pepe111', rol: 'normal' }
];
```

Cuando un usuario valide sus credenciales, además de almacenar su nombre de usuario en sesión, también podemos (debemos) almacenar su rol. Así que la ruta que valida el usuario se ve modificada para añadir este nuevo dato en sesión:

```
app.post('/login', (req, res) => {
  let login = req.body.login;
  let password = req.body.password;

  ...

  if (existeUsuario.length > 0)
  {
    req.session.usuario = existeUsuario[0].usuario;
    req.session.rol = existeUsuario[0].rol;
    res.render('index');
  } else {
    ...
  }
});
```

Para poder comprobar si un usuario validado tiene el rol adecuado para acceder a un recurso, podemos definir otra función *middleware* que compruebe si el rol del usuario es el que se necesita (el que se le pasa como parámetro a la función):

```
let rol = (rol) => {
  return (req, res, next) => {
    if (rol === req.session.rol)
      next();
    else
      res.render('login');
  }
}
```

NOTA: el ejemplo que acabamos de ver es una muestra de cómo podemos definir *middleware* que necesite parámetros adicionales además de los tres que todo *middleware* debe tener (petición, respuesta y siguiente función a llamar). Basta con definir una función con los parámetros necesarios, y que internamente devuelva la función *middleware* con los tres parámetros base.

Si queremos aplicar los dos *middleware* a una ruta determinada (es decir, comprobar si el usuario está autenticado y, además, si tiene el rol adecuado), podemos pasarlos uno tras otro, separados por comas, en la definición de la ruta. Por ejemplo, a esta ruta sólo deben poder acceder usuarios validados que tengan rol de administrador:

```
app.get('/protegidoAdmin', autenticacion,  
      rol('admin'), (req, res) => {  
    res.render('protegido_admin');  
  })
```

3.2.1.5. Otras opciones

Además de las opciones vistas anteriormente, hay algunas operaciones más que, si bien pueden ser secundarias, conviene tener presentes cuando trabajamos con autenticación basada en sesiones.

Por un lado, está la posibilidad de hacer **logout** y salir de la sesión. Para esto, podemos definir una ruta que responda a esta petición, y destruya los datos de sesión del usuario, redirigiendo después a otro recurso:

```
app.get('/logout', (req, res) => {  
  req.session.destroy();  
  res.redirect('/');  
});
```

Para poder acceder a la sesión desde las vistas, debemos definir un middleware que asocie la sesión con los recursos de la vista:

```
app.use((req, res, next) => {  
  res.locals.session = req.session;  
  next();  
});
```

Después, podemos acceder a esta sesión desde las vistas, a través de la variable `session` que hemos definido en la respuesta (`res.locals`). Por ejemplo, así podríamos ver si un usuario está ya logueado, para mostrar o no el botón de "Login":

```
{% if (session and session.usuario) %}  
  <a class="btn btn-dark" href="/logout">Logout</a>  
{% else %}  
  <a class="btn btn-dark" href="/login">Login</a>  
{% endif %}
```

Además, podemos establecer el **tiempo de vida** de la sesión, cuando la configuramos. Podemos hacerlo utilizando indistintamente el atributo `expires` o el atributo `maxAge`,

aunque con una sintaxis algo distinta según cuál utilicemos. Debemos indicar el número de milisegundos de vida, contando desde el momento actual, por lo que se suele utilizar `Date.now()` en estos cálculos. Así definiríamos, por ejemplo, una sesión de 30 minutos:

```
app.use(session({
  secret: '1234',
  resave: true,
  saveUninitialized: false,
  expires: new Date(Date.now() + (30 * 60 * 1000))
}));
```

En los recursos de la sesión tenéis un ejemplo completo para probar estos mecanismos. Se tiene una página de inicio pública, una restringida para usuarios validados y otra restringida para usuarios administradores. Se dispone también de un formulario de *login* y de una ruta de *logout*.

Ejercicios propuestos:

1. Crea una carpeta llamada **"T3_LibrosSesiones"** en la carpeta de ejercicios *"ProyectosNode/Ejercicios"*. Copia dentro el contenido de *T3_LibrosWeb* de sesiones anteriores de este tema, que incluía formularios para insertar, borrar o modificar libros.

A partir de esa base, vamos a añadir ahora autenticación basada en sesiones. Instala el *middleware express-session* en el proyecto, y configúralo como en el ejemplo visto antes. Define a mano en el servidor principal un array con nombres y passwords de usuarios autorizados, y protege las rutas que permitan hacer cualquier modificación sobre el catálogo de libros. En concreto, sólo los usuarios validados podrán:

- Ver el formulario de inserción de libros e insertar libros (enviar el formulario anterior)
- Borrar libros
- Ver el formulario de edición de libros y editar libros (enviar el formulario)

Añade para ello una vista `login.njk` al conjunto de vistas de la aplicación. Puedes emplear el mismo formulario de login que en el ejemplo, y también añade las dos rutas para mostrar el formulario y para recoger los datos y validar el usuario. En caso de validación exitosa, se renderizará la vista del listado de libros. En caso contrario, el formulario de login con un mensaje de error, como en el ejemplo proporcionado.

Finalmente, añade también una función de *logout* al menú de la aplicación, que sólo será visible si el usuario ya está validado, y que permitirá destruir su sesión y redirigir al listado de libros.

3.2.2. Autenticación basada en tokens

La autenticación basada en sesiones vista antes no es aplicable a todos los clientes web, sino sólo a los basados en navegador. Se hace necesario, por tanto, otro mecanismo de autenticación que permita ser aplicado a aplicaciones cliente de otros tipos, como por ejemplo las aplicaciones de escritorio, o móviles.

3.2.2.1. Fundamentos de la autenticación por token

Un **token** por sí mismo es una cadena de texto que carece de significado. Pero combinado con ciertas claves, es un mecanismo para proteger nuestras aplicaciones de accesos no permitidos. La autenticación basada en tokens es un método mediante el cual nos aseguramos de que cada petición a un servidor viene acompañada por un token firmado, que contiene los datos necesarios para verificar que el cliente ya ha sido validado previamente.

El mecanismo empleado para la autenticación por token es el siguiente:

1. El cliente envía sus datos de autenticación al servidor (típicamente un login y password).
2. El servidor valida esas credenciales contra algún tipo de almacenamiento (normalmente una base de datos). En el caso de que sean correctas, genera un token, una cadena codificada que permite identificar al usuario. Este token se envía al cliente, normalmente como datos de la respuesta. Internamente, puede contener algún dato que permita identificar al usuario, como su login o e-mail.

NOTA: No conviene añadir en el token (ni tampoco en las sesiones) información muy confidencial, como el password, por ejemplo, ya que puede ser fácilmente descodificable. Esto no quiere decir que el token sea un mecanismo inseguro de autenticación, ya que el servidor utiliza una palabra secreta para cifrar una parte del token, y así poder verificar que es correcto, pero el resto del token sí queda más descubierto.

3. El cliente recibe el token y lo almacena de alguna forma localmente (mediante mecanismos como `localStorage` en JavaScript o similares en otros lenguajes). Ante cada nueva petición que se haga, el cliente reenvía dicho token en las cabeceras de la petición, para que el servidor verifique que es un cliente autorizado.

Normalmente a los tokens se les asigna un tiempo de vida o una fecha de caducidad (que pueden ser minutos, días, semanas... dependiendo de lo que nos interese y del tipo de aplicación). En cada nueva reconexión, el tiempo de vida se puede renovar (no es algo automático, deberemos hacerlo nosotros), y si pasa más de ese tiempo estipulado sin que el cliente intente conectar, se le solicitará de nuevo que se autentifique.

Se suele emplear el estándar JWT (JSON Web Token), que define una forma compacta de transmitir esta información entre cliente y servidor empleando objetos JSON.

Estructura de un token JWT

Un token JWT consiste en tres partes:

- **Cabecera (*header*)**: con información sobre el tipo de token generado (en este caso, JWT), algoritmo de encriptación (HS256, por ejemplo), etc.
- **Carga (*payload*)**: contiene la información codificada por el token (login, roles, permisos concretos, caducidad, etc), lo que se conoce como declaraciones (*claims*). Conviene no enviar información confidencial en este elemento, ya que, como veremos, es fácilmente descodificable.
- **Firma (*signature*)**: se emplea para validar el token y protegerlo frente a manipulaciones. Esta firma se genera mezclando tres elementos: la cabecera, el payload y una palabra secreta.

Se generará una cadena en formato base64 con toda la información del token. Por ejemplo:

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJtZXRzYWdlIjoiSldUIFJlbGVzISIsImhhdCI6MTQ1OTQ0ODExOSwiZXhwIjoxNDU5NDU0NTE5fQ.-yIVBD5b73C75osbmwwshQNR7frWUYrqaTjTpza2y4
```

Si pegamos esta cadena en algún procesador de tokens, como el de jwt.io, podremos ver descodificadas la cabecera y el payload:

Encoded

PASTE A TOKEN HERE

eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJtZXRzYWdlIjoiSldUIFJlbGVzISIsImhhdCI6MTQ1OTQ0ODExOSwiZXhwIjoxNDU5NDU0NTE5fQ.-yIVBD5b73C75osbmwwshQNR7frWUYrqaTjTpza2y4

Decoded

EDIT THE PAYLOAD AND SECRET (ONLY HS256 SUPPORTED)

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "typ": "JWT",
  "alg": "HS256"
}
```

PAYLOAD: DATA

```
{
  "message": "JWT Rules!",
  "iat": 1459448119,
  "exp": 1459454519
}
```

VERIFY SIGNATURE

Esto quiere decir que no existe una encriptación para los datos que se envían (cabecera y *payload*), se pueden descodificar fácilmente desde base64. Pero sin conocer la palabra secreta que genera la firma, no se puede validar que ese token sea correcto. Esa palabra sólo la conoce el servidor. En el caso del ejemplo anterior, si utilizamos la palabra "L3@RNJWT" podremos obtener la validación del token en la página anterior.

Es decir, la tercera parte del token se emplea por el servidor para, una vez obtenidas la cabecera y el payload (descodificándolas desde base64) y uniendo a ellas la palabra secreta, comprobar que la cadena generada coincide con esa tercera parte. Si es así, el token es auténtico.

Ventajas de la autenticación por token

El mecanismo de autenticación por token ofrece algunas ventajas respecto al tradicional método por sesiones:

- Los tokens no tienen estado (*stateless*), lo que significa que el servidor no debe almacenar en ninguna parte el registro de usuarios autenticados para comprobar si quien entra ha sido autorizado antes. La propia cadena que se envían cliente y servidor contiene toda la información. Por contra, la información de las sesiones (o los identificadores de cada sesión) sí se almacenan en el servidor.
- Cualquier aplicación cliente soporta tokens, lo que supone que podemos emplear este mismo mecanismo para autenticar una aplicación de escritorio, móvil o web contra el mismo *backend*.
- Los tokens permiten especificar información adicional de acceso, como roles de usuario, permisos concretos a recursos, etc.

3.2.2.2. Configurando la autenticación basada en tokens

Para probar cómo funciona la autenticación basada en tokens, vamos a implementar una pequeña API REST de ejemplo, que defina un par de rutas (una pública y otra protegida), que devuelvan cierta información en formato JSON. Puedes descargar el ejemplo completo en los recursos de esta sesión.

En el servidor principal Express, hemos definido una ruta principal de acceso público, y otra a la URI `/protegido` que sólo será accesible por usuarios registrados. Igual que en el caso anterior, hemos optado por almacenar los usuarios previamente en un vector, simulando que ya los tenemos cargados de la base de datos:

```
const usuarios = [
  { usuario: 'nacho', password: '12345' },
  { usuario: 'pepe', password: 'pepe111' }
];

let app = express();

app.get('/', (req, res) => {
  res.send({ok: true, resultado: "Bienvenido a la ruta de inicio"});
});

app.get('/protegido', (req, res) => {
  res.send({ok: true, resultado: "Bienvenido a la zona protegida"});
});
```

Para poder generar un token utilizaremos la librería *jsonwebtoken*, que se basa en el estándar JWT comentado antes para generar dicho token cuando corresponda. Lo primero que haremos será instalarla en el proyecto que la necesite:

```
npm install jsonwebtoken
```

Después, la incorporamos a nuestro servidor Express con el resto de módulos:

```
const express = require('express');
const jwt = require('jsonwebtoken');
...
```

3.2.2.3. Validando al cliente

El proceso de validación comprende dos pasos básicos:

1. Recoger las credenciales de la petición del cliente y comprobar si son correctas
2. Si lo son, generar un token y enviárselo de vuelta al cliente

Comencemos por el segundo paso: definimos una función que, utilizando la librería *jsonwebtoken* instalada anteriormente, genere un token firmado, que almacene cierta información que nos pueda ser útil (por ejemplo, el *login* del usuario validado).

```
const secreto = "secretoNode";

let generarToken = login => {
  return jwt.sign({login: login}, secreto,
    {expiresIn: "2 hours"});
};
```

El método `sign` recibe tres parámetros: el objeto JavaScript con los datos que queramos almacenar en el token (en este caso, el login del usuario validado, que recibimos como parámetro del método), una palabra secreta para cifrarlo, y algunos parámetros adicionales, como por ejemplo el tiempo de expiración.

Notar que, igual que ocurría con la autenticación basada en sesiones, necesitamos una palabra secreta para, en este caso, cifrar el contenido del token. Esta palabra secreta la hemos definido en una constante en el código, aunque normalmente se recomienda que se ubique en un archivo externo a la aplicación, para evitar que se pueda acceder a ella fácilmente.

Esta función `generarToken` la emplearemos en la ruta de *login*, que recogerá las credenciales del cliente por POST y las cotejará contra alguna base de datos o similar. Si son correctas, llamaremos a la función anterior para que genere el token, y se lo enviaremos al cliente como parte de la respuesta JSON:

```
app.post('/login', (req, res) => {  
  let usuario = req.body.usuario;  
  let password = req.body.password;  
  
  let existeUsuario = usuarios.filter(u =>  
    u.usuario == usuario && u.password == password);  
  
  if (existeUsuario.length == 1)  
    res.send({ok: true, token: generarToken(usuario)});  
  else  
    res.send({ok: false});  
});
```

3.2.2.4. Autenticando al cliente validado

El cliente recibirá el token de acceso la primera vez que se valide correctamente, y dicho token se debe almacenar en algún lugar de la aplicación. Podemos emplear mecanismos como la variable `localStorage` para aplicaciones basadas en JavaScript y navegadores, u otros métodos en el caso de trabajar con otras tecnologías y lenguajes.

A partir de este punto, cada vez que queramos solicitar algún recurso protegido del servidor, deberemos adjuntar nuestro token para mostrarle que ya estamos validados. Para ello, el token suele enviarse en la cabecera de petición *Authorization*. Desde el punto de vista del servidor no tenemos que hacer nada al respecto en este apartado, salvo leer el token de dicha cabecera cuando nos llegue la petición, y validarlo. Por ejemplo, el siguiente *middleware* obtiene el token de la cabecera, y llama a un método `validarToken` que veremos después para su validación:

```
let protegerRuta = (req, res, next) => {  
  let token = req.headers['authorization'];  
  if (validarToken(token))  
    next();  
  else  
    res.send({ok: false, error: "Usuario no autorizado"});  
};
```

La función `validarToken` se encarga de llamar al método `verify` de *jsonwebtoken* para comprobar si el token es correcto, de acuerdo a la palabra secreta de codificación.

```
let validarToken = (token) => {  
  try {  
    let resultado = jwt.verify(token, secreto);  
    return resultado;  
  } catch (e) {}  
};
```

La función obtiene el objeto almacenado en el token (con el login del usuario, en este caso) y devolverá `null` si algo falla.

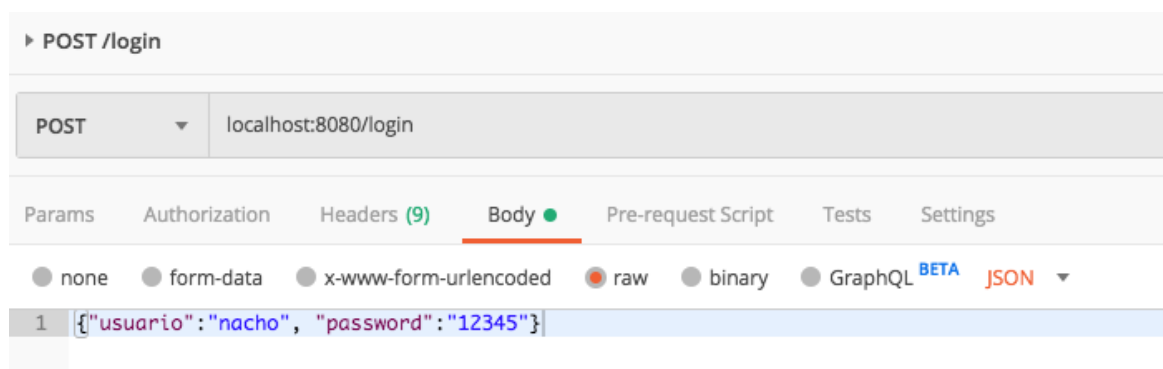
En caso de que algo falle, el propio *middleware* `protegerRuta` redirige al formulario de login en este caso. Nos falta aplicar este *middleware* a las rutas protegidas, y para eso lo añadimos en la cabecera de la propia ruta, como segundo parámetro:

```
app.get('/protegido', protegerRuta, (req, res) => {  
  res.send({ok: true,  
    resultado: "Bienvenido a la zona protegida"});  
});
```

NOTA: según los estándares, se indica que la cabecera "Authorization" que envía el token tenga un prefijo "Bearer ", por lo que el contenido de esa cabecera normalmente será "Bearertoken.....", y por tanto para obtener el token habría que procesar el valor de la cabecera y cortar sus primeros caracteres. En el ejemplo que se os proporciona se procesa y corta dicho prefijo.

3.2.2.5. Pruebas de autenticación con Postman

Vamos a probar la aplicación de ejemplo con Postman, y veremos cómo obtener y enviar el token de acceso desde esta herramienta. Lo primero que deberemos hacer es una petición POST para loguearnos. En Postman puede quedar así:



Recibiremos como respuesta el *token* que se haya generado:

```

1 {
2   "ok": true,
3   "token":
4     "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJsb2dpbiI6Im5hY2hvIiwiaWF0IjoxNTcwNzg5Mjg1LCJleHAiOiE1NzA3OTY0ODV9.XA1FrLdP2Ujv-QEwzYDgLYsljLp9ae--tdC-sjLdPU3E"
5 }

```

Ahora, sólo nos queda adjuntar este token en la cabecera *Authorization* de las peticiones que lo necesiten. Para ello, vamos a la sección *Authorization* bajo la URL de la petición, y elegimos que queremos enviar un *Bearer token*. En el cuadro inferior nos dejará copiar dicho token:

GET /protégido

Authorization: Bearer Token

Token: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJsb2dpbiI6Im5hY2hvIiwiaWF0IjoxNTcwNzg5Mjg1LCJleHAiOiE1NzA3OTY0ODV9.XA1FrLdP2Ujv-QEwzYDgLYsljLp9ae--tdC-sjLdPU3E

```

1 {
2   "ok": true,
3   "resultado": "Bienvenido a la zona protegida"
4 }

```

Disponéis en el ejemplo de una colección Postman para probar el acceso. Basta con que hagáis "login" y utilicéis el token que os proporciona para acceder a la zona protegida.

3.2.2.6. Otras opciones

Nos quedan en el tintero un par de opciones más que sí hemos probado en la autenticación por sesiones.

- Para hacer **logout**, como el token ya no se almacena en el servidor, basta con eliminarlo del almacenamiento que tengamos en el cliente, por lo que ahora es responsabilidad exclusiva del cliente salir del sistema.
- Para definir **roles** de acceso, podemos hacerlo de forma similar al caso de las sesiones: añadir el campo del rol junto a cada usuario, y almacenar dicho rol en el token, junto con el login. Después, bastaría con modificar el método de `protegerRuta` para que procese lo que devuelve `validarToken` (el objeto incrustado en el token) y compruebe si tiene el rol adecuado. También deberíamos modificar el método `generarToken` para que reciba como parámetro el login y rol a añadir al token, y la ruta de POST `/login`

para que le pase estos dos datos al método de `generarToken`, cuando el usuario sea correcto.

```
let generarToken = (login, rol) => {
  return jwt.sign({login: login, rol: rol}, secreto,
    {expiresIn: "2 hours"});
};

...

let protegerRuta = rol => {
  return (req, res, next) => {
    let token = req.headers['authorization'];
    if (token) {
      token = token.substring(7);
      let resultado = validarToken(token);
      if (resultado && (rol === "" || rol === resultado.rol))
        next();
      else
        res.send({ok: false, error: "Usuario no autorizado"});
    } else
      res.send({ok: false, error: "Usuario no autorizado"});
  };
};

...

app.post('/login', (req, res) => {
  let usuario = req.body.usuario;
  let password = req.body.password;

  let existeUsuario = usuarios.filter(u =>
    u.usuario == usuario && u.password == password);

  if (existeUsuario.length == 1)
    res.send({ok: true,
      token: generarToken(existeUsuario[0].usuario,
        existeUsuario[0].rol)});
  else
    res.send({ok: false});
});
```

Disponéis del ejemplo completo, incluyendo gestión de roles para acceder a la URI `/protegidoAdmin`.

Ejercicios propuestos:

2. Crea una carpeta llamada **"T3_LibrosToken"** en la carpeta de ejercicios *"ProyectosNode/Ejercicios"*. Copia dentro el contenido del *T2_LibrosExpress* del tema anterior, donde se estructuró una API REST completa sobre la base de datos de libros, incluyendo una carpeta de modelos, enrutadores, etc.

Lo que vamos a hacer sobre este ejercicio es añadir una autenticación basada en tokens usando la librería *jsonwebtoken*. Definiremos un array estático de usuarios registrados, como en el ejemplo, y añadiremos la librería y los métodos para generar y validar el token, como en el ejemplo. Los utilizaremos para proteger el acceso a los servicios que impliquen modificación de datos (POST, PUT y DELETE sobre la colección de libros). Deberás añadir también un servicio de *login* (**POST /login**) que reciba los datos del usuario en el cuerpo de la petición y le devuelva el token, como en el ejemplo resuelto de la sesión.

Crea una nueva colección en Postman llamada *LibrosToken*, y adapta la colección que hiciste originalmente, para utilizar tokens en los servicios que lo requieran, añadiendo también el servicio para el *login*.