7. Seguridad basada en tokens

7.1. Introducción

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 este último mecanismo, y cómo configurarlo en aplicaciones Express.

7.2. 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.

7.2.1. Estructura de un token JWT

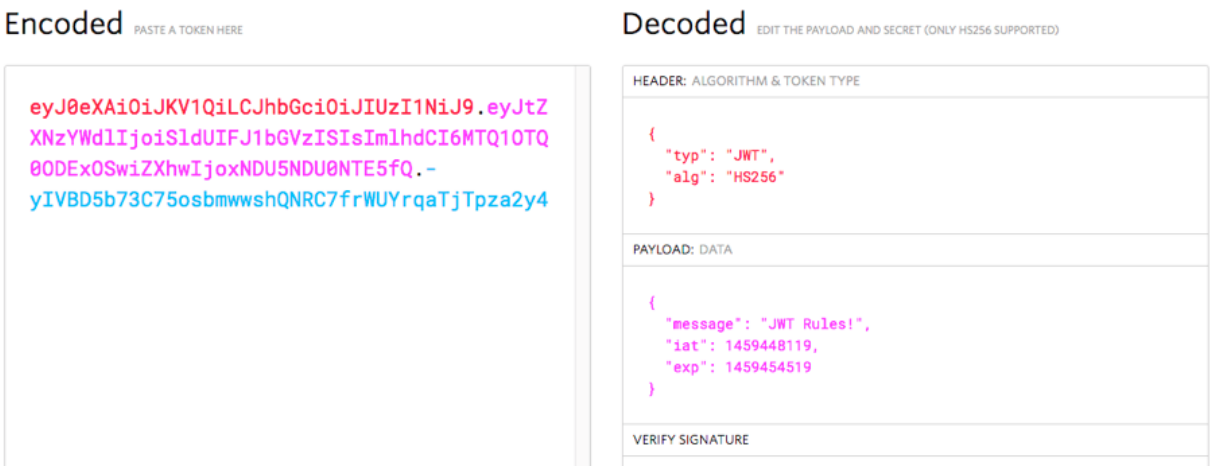
Un token JWT consiste en tres partes:

- **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.eyJtZXNzYWdlIjoiSldUIFJ1bGVzISIsImlhdCI6MTQ1
OTQ0ODExOSwiZXhwIjoxNDU5NDU0NTE5fQ.-yIVBD5b73C75osbmwwshQNRC7frWUYrqaTjTpza2y4
```

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



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.

7.2.2. Ventajas e inconvenientes 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.

Sin embargo, para poder implementar una autenticación basada en tokens en una aplicación tradicional de navegador, necesitamos que el cliente no sea un cliente "tonto", que se limite a renderizar el contenido HTML y poco más. Es necesario cierto pre-procesamiento JavaScript en la parte cliente para enviar el token al servidor en cada petición, así como para recoger el token generado por el servidor cuando nos autenticamos. Esto hace imprescindible el uso de ciertas librerías en el cliente, como jQuery, Angular, React o Vue, entre otras.

7.3. 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. Para simplificar la gestión de usuarios, hemos optado por almacenarlos 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. Lo primero que haremos será instalarla en el proyecto que la necesite (además de instalar Express, en este caso):

```
npm install jsonwebtoken
```

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

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

7.3.1. 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 necesitamos una palabra secreta para 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});
});
```

7.3.2. 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* envía un mensaje de error 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:

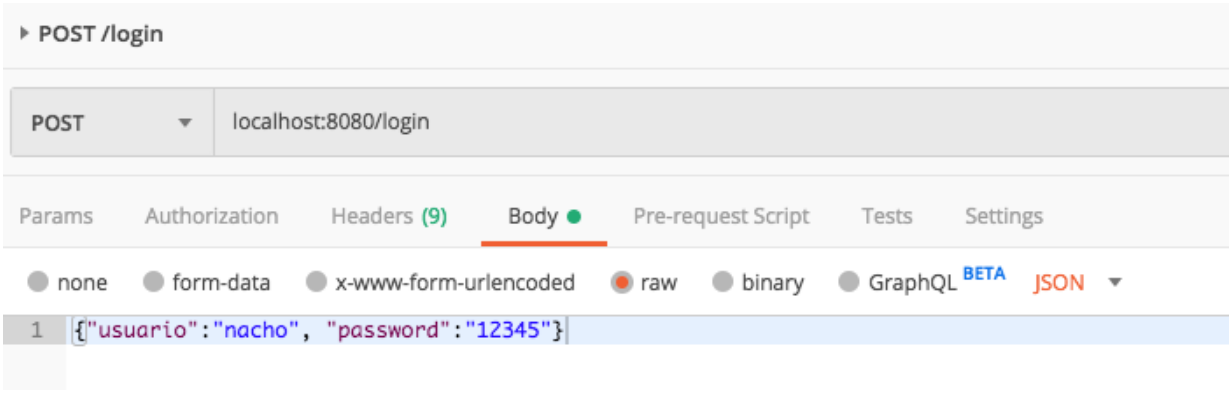
```
app.get('/protegido', protegerRuta, (req, res) => {
    res.send({ok: true, resultado: "Bienvenido a la zona protegida"});
});
```

> **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 "Bearer .....token.....", 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.

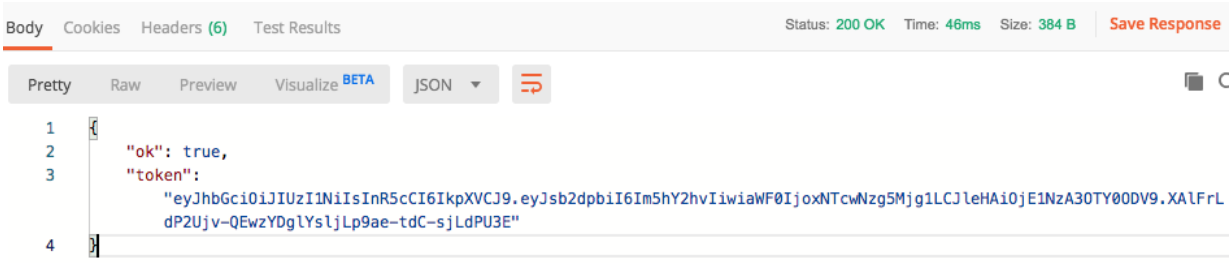
7.4. Pruebas de autenticación con Postman

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

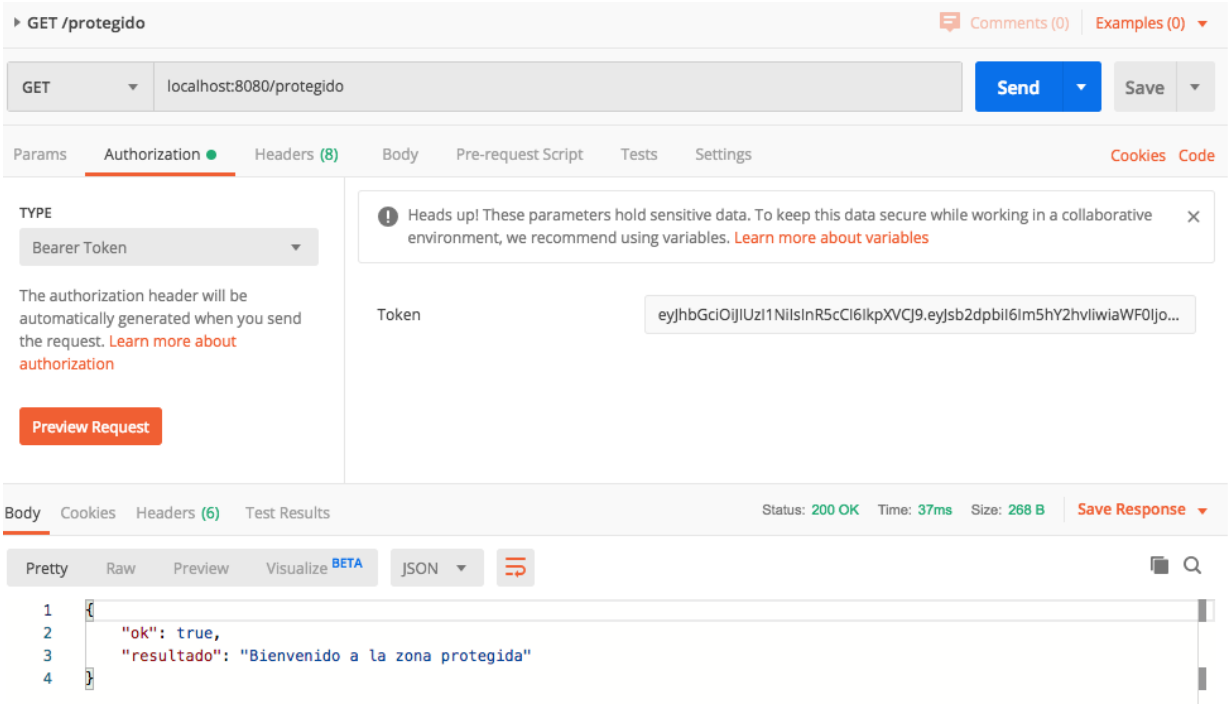
puede quedar así:



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



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:



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.

7.5. Otras opciones

Nos quedan en el tintero, como son la opción de "cerrar sesión" y la gestión de roles.

7.5.1. Cierre de sesión o *logout*

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 es responsabilidad exclusiva del cliente salir del sistema, a diferencia de la autenticación basada en sesiones, donde era el servidor quien debía destruir la información almacenada.

7.5.2. Definir roles de acceso

Para definir **roles** de acceso, podemos añadir un campo del rol que tiene cada usuario, y almacenar dicho rol en el token, junto con el login.

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

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`.

En este punto, puedes intentar realizar el Ejercicio 1 de esta sesión.

7.6. Ejercicios propuestos

Para el ejercicio de esta sesión, crea una subcarpeta llamada "**Sesion7**" en tu carpeta "**ProyectosNode/Ejercicios**", para dentro crear el proyecto del ejercicio.

Ejercicio 1

Crea una carpeta llamada "**Ejercicio_7_1**" en la carpeta de ejercicios "*ProyectosNode/Ejercicios/Sesion7*". Copia dentro el contenido del *Ejercicio_5_3* de la sesión 5, donde implementamos una API REST sobre los libros.

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, 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. Este servicio lo puedes crear en la aplicación principal *index*.

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*.

Deberás adjuntar como entrega de este ejercicio tanto el proyecto Node (sin la carpeta `node_modules`) como la colección de Postman exportada.