Configuración de la autenticación por tokens

En este documento veremos cómo se aplica la autenticación basada en tokens en aplicaciones REST con Node.js. Comenzaremos viendo un ejemplo sencillo, para luego aplicar los mismos criterios en las aplicaciones que hemos venido desarrollando en sesiones anteriores.

1. Un ejemplo sencillo

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. Crearemos para ello un proyecto *AutenticacionTokens* en nuestra carpeta *ProyectosNode/Pruebas*.

1.1. El servidor principal

En el servidor principal `index.js` usaremos Express y definiremos 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 express = require('express');

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"});
});

app.listen(8080);
```

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 express
```

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

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

1.2. 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 jwt = require('jsonwebtoken');
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, o como una variable de entorno del sistema, para evitar que se pueda acceder a ella fácilmente. En este último caso, suponiendo que hemos llamado `SECRETO` a dicha variable de entorno, podemos acceder a ella así:

```
return jwt.sign({...}, process.env.SECRETO, {...});
```

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});
});
```

1.3. 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.

```
let validarToken = (token) => {
    try {
        let resultado = jwt.verify(token.substring(7), secreto);
        return resultado;
    } catch (e) {}
};
```

2. Autenticación en proyectos complejos

El ejemplo anterior nos ha servido para conocer los principios básicos de la autenticación basada en tokens. Pero... ¿qué ocurre cuando queremos proteger varias rutas dispuestas en distintos enrutadores? Una opción sería replicar las funciones de *validarToken* o *protegerRuta* en cada enrutador, con el consiguiente problema de la duplicidad y mantenimiento de ese código.

Lo que se suele hacer en estos casos es sacar todo el proceso de autenticación basada en tokens a un módulo aparte e incluir dicho módulo en los enrutadores o ficheros que lo requieran. Vamos a seguir estos pasos en el proyecto de contactos. Para ello, tomaremos el proyecto *ContactosREST_v2* de sesiones anteriores y lo vamos a copiar en otro llamado *ContactosRESTToken*.

2.1. Nuevos módulos y enrutadores

Comenzaremos instalando la librería *jsonwebtoken* junto a las que ya tiene el proyecto (Express y Mongoose), y después crearemos dos ficheros en el proyecto:

- `utils/auth.js` : donde incluiremos todas las funciones y mecanismos para la autenticación por token

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

const secreto = "secretoNode";

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

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

let protegerRuta = (req, res, next) => {
    let token = req.headers['authorization'];
    if (token && token.startsWith("Bearer "))
        token = token.slice(7);

    if (validarToken(token))
        next();
    else
        res.send({ok: false, error: "Usuario no autorizado"});
}

module.exports = {
    generarToken: generarToken,
    validarToken: validarToken,
    protegerRuta: protegerRuta
};
```

- `routes/auth.js` : donde incluiremos el servicio POST para hacer el *login* correspondiente.

```
const express = require('express');
const auth = require(__dirname + '/../utils/auth');

let router = express.Router();

// Simulamos la base de datos así
const usuarios = [
    { usuario: 'nacho', password: '12345' },
    { usuario: 'alex', password: 'alex111' }
];

router.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: auth.generarToken(usuario)});
    else
        res.send({ok: false});
});

module.exports = router;
```

2.2. El programa principal

En el programa principal `index.js` deberemos cargar el nuevo enrutador y asociarlo a alguna ruta (por ejemplo, `/auth`):

```
...

// Enrutadores
const mascotas = require(__dirname + '/routes/mascotas');
const restaurantes = require(__dirname + '/routes/restaurantes');
const contactos = require(__dirname + '/routes/contactos');
const auth = require(__dirname + '/routes/auth');

...

// Middleware para peticiones POST y PUT
// Enrutadores para cada grupo de rutas
app.use(express.json());
app.use('/mascotas', mascotas);
app.use('/restaurantes', restaurantes);
app.use('/contactos', contactos);
app.use('/auth', auth);

// Puesta en marcha del servidor
app.listen(8080);
```

2.3. Proteger los servicios deseados

Si queremos proteger cualquier servicio de cualquier enrutador, basta con que carguemos el módulo `utils/auth` y añadamos el *middleware* `protegerRuta` en dicho servicio. Por ejemplo, podemos proteger los servicios POST, PUT y DELETE de todos los enrutadores:

Ficheros routes/restaurantes.js y routes/mascotas.js

```
const express = require('express');
const auth = require(__dirname + '/../utils/auth.js');
...

// Servicio de inserción
router.post('/', auth.protegerRuta, (req, res) => {
    ...
});

// Servicio de borrado
router.delete('/:id', auth.protegerRuta, (req,res) => {
    ...
});

module.exports = router;
```

Fichero routes/contactos.js

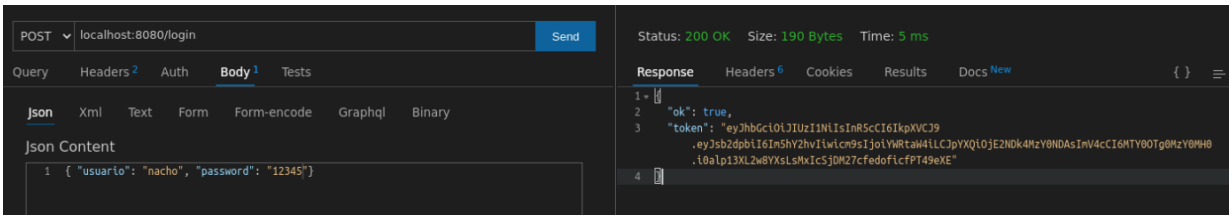
```
const express = require('express');
const auth = require(__dirname + '/../utils/auth.js');
...

// Servicio de inserción
router.post('/', auth.protegerRuta, (req, res) => {
    ...
});

// Servicio de modificación
router.put('/:id', auth.protegerRuta, (req, res) => {
    ...
});

// Servicio de borrado
router.delete('/:id', auth.protegerRuta, (req,res) => {
    ...
});

module.exports = router;
```

3. Prueba de autenticación y otras opciones

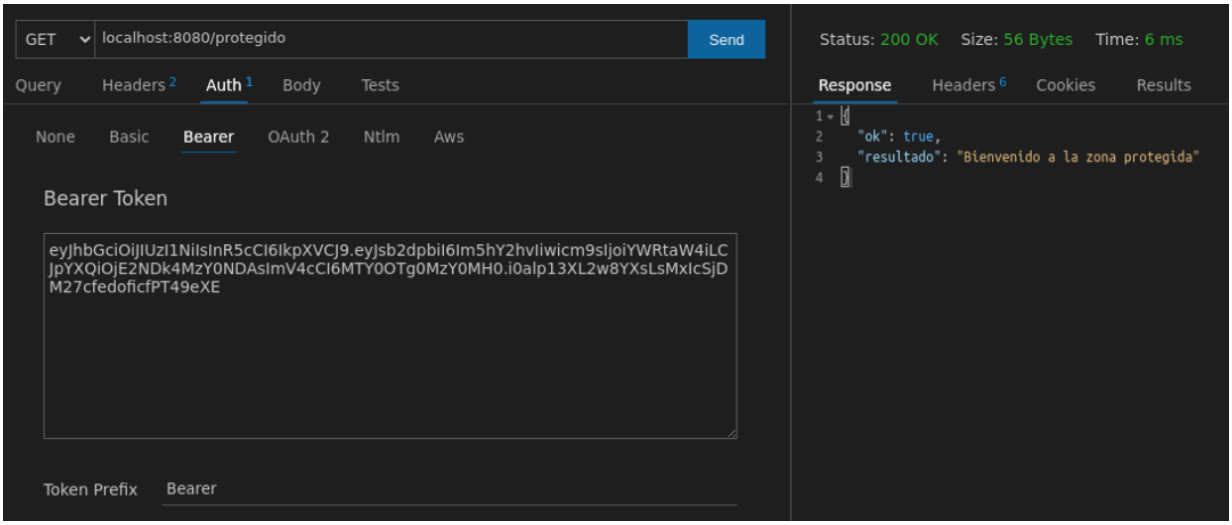
Ahora que ya hemos visto cómo incorporar los mecanismos de autenticación basada en tokens en nuestras aplicaciones REST, veremos algunas cosas que se han quedado en el tintero en este documento.

3.1. Pruebas de autenticación con ThunderClient

Vamos a probar la aplicación de ejemplo con *ThunderClient* desde Visual Studio Code, y veremos cómo obtener y enviar el token de acceso desde esta herramienta. Lo primero que deberemos hacer es una petición POST para loguearnos. 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:



3.2. 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.

3.3. 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});
});
```

Ejercicio 1:

Crea una copia del ejercicio *LibrosREST_v2* previo, y llámala **LibrosRESTToken**. 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 con *login*, *password* y *rol*, que podrá ser *editor* o *admin*, y añadiremos la librería y los métodos para generar y validar el token, como en el ejemplo de contactos. Los utilizaremos para proteger el acceso a los servicios que impliquen modificación de datos (POST, PUT y DELETE sobre la colección de libros y/o autores), de modo que los usuarios *admin* podrán acceder a todos estos servicios, y los de tipo *editor* no podrán acceder a los servicios de modificación de

> autores. Por tanto, el método `protegerRuta` deberá tener en cuenta el rol del usuario en algunos casos, como se ha explicado antes.
>
> 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 con la información útil guardada (login y rol del usuario validado) como en el ejemplo de la sesión. Crea una nueva colección en *ThunderClient* 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*.