Configuració de l'autenticació amb tokens


En aquest document veurem com s'aplica l'autenticació basada en tokens en aplicacions REST amb Node.js. Començarem veient un exemple senzill, per a després aplicar els mateixos criteris en les aplicacions que hem vingut desenvolupant en sessions anteriors.

1. Un exemple senzill

Per a provar com funciona l'autenticació basada en tokens, implementarem una xicoteta API REST d'exemple, que definisca un parell de rutes (una pública i una altra protegida), que retornen una certa informació en format JSON. Crearem per a això un projecte *AutenticacioTokens* en la nostra carpeta *ProjectesNode/Proves*.

1.1. El servidor principal

En el servidor principal `index.js` usarem Express i definirem una ruta principal d'accés públic, i una altra a la URI `/protegit` que només serà accessible per usuaris registrats. Per a simplificar la gestió d'usuaris, hem optat per emmagatzemar-los en un vector, simulant que ja els tenim carregats de la base de dades:

```
const express = require('express');

const usuaris = [
    { usuari: 'nacho', password: '12345' },
    { usuari: 'pepe', password: 'pepe111' }
];

let app = express();

app.get('/', (req, res) => {
    res.send({ok: true, resultat: "Benvingut a la ruta d'inici"});
});

app.get('/protegit', (req, res) => {
    res.send({ok: true, resultat: "Benvingut a la zona protegida"});
});

app.listen(8080);
```

Per a poder generar un token utilitzarem la llibreria *jsonwebtoken*, que es basa en l'estàndard JWT comentat abans. El primer que farem serà instal·lar-la en el projecte que la necessite (a més d'instal·lar Express, en aquest cas):

```
npm install jsonwebtoken express
```

Després, la incorporem al nostre servidor Express amb la resta de mòduls:

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

1.2. Validant al client

El procés de validació comprén dos passos bàsics:

1. Recollir les credencials de la petició del client i comprovar si són correctes
2. Si ho són, generar un token i enviar-li-ho de tornada al client

Comencem pel segon pas: definim una funció que, utilitzant la llibreria *jsonwebtoken* instal·lada anteriorment, genere un token signat, que emmagatzeme una certa informació que ens puga ser útil (per exemple, el *login* de l'usuari validat).

```
const jwt = require('jsonwebtoken');
const secret = "secretNode";

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

El mètode `sign` rep tres paràmetres: l'objecte JavaScript amb les dades que vulguem emmagatzemar en el token (en aquest cas, el login de l'usuari validat, que rebem com a paràmetre del mètode), una paraula secreta per a xifrar-lo, i alguns paràmetres addicionals, com per exemple el temps d'expiració.

Notar que necessitem una paraula secreta per a xifrar el contingut del token. Aquesta paraula secreta l'hem definida en una constant en el codi, encara que normalment es recomana que se situe en un arxiu extern a l'aplicació, o com una variable d'entorn del sistema, per a evitar que es puga accedir a ella fàcilment. En aquest últim cas, suposant que hem anomenat `SECRET` a aquesta variable d'entorn, podem accedir a ella així:

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

Aquesta funció `generarToken` l'emprarem en la ruta de *login*, que recollirà les credencials del client per POST i les acararà contra alguna base de dades o similar. Si són correctes, cridarem a la funció anterior perquè

genere el token, i li ho enviarem al client com a part de la resposta JSON:

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

    let existeixUsuari = usuaris.filter(u =>
        u.usuari == usuari && u.password == password);

    if (existeixUsuari.length == 1)
        res.send({ok: true, token: generarToken(usuari)});
    else
        res.send({ok: false});
});
```

1.3. Autenticant al client validat

El client rebrà el token d'accés la primera vegada que es valide correctament, i dit token s'ha d'emmagatzemar en algun lloc de l'aplicació. Podem emprar mecanismes com la variable `localStorage` per a aplicacions basades en JavaScript i navegadors, o altres mètodes en el cas de treballar amb altres tecnologies i llenguatges.

A partir d'aquest punt, cada vegada que vulguem sol·licitar algun recurs protegit del servidor, haurem d'adjuntar nostre token per a mostrar-li que ja estem validats. Per a això, el token sol enviar-se en la capçalera de petició *Authorization*. Des del punt de vista del servidor no hem de fer res sobre aquest tema en aquest apartat, excepte llegir el token d'aquesta capçalera quan ens arribe la petició, i validar-ho. Per exemple, el següent *middleware* obté el token de la capçalera, i crida a un mètode `validarToken` que veurem després per a la seua validació:

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

La funció `validarToken` s'encarrega de cridar al mètode `verify` de *jsonwebtoken* per a comprovar si el token és correcte, d'acord amb la paraula secreta de codificació.

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

La funció obté l'objecte emmagatzemat en el token (amb el login de l'usuari, en aquest cas) i retornarà `null` si alguna cosa falla.

En cas que alguna cosa falle, el propi *middleware* envia un missatge d'error en aquest cas. Ens falta aplicar aquest *middleware* a les rutes protegides, i per a això ho afegim en la capçalera de la pròpia ruta, com a segon paràmetre:

```
app.get('/protegit', protegirRuta, (req, res) => {
    res.send({ok: true, resultat: "Benvingut a la zona protegida"});
});
```

> **NOTA:** segons els estàndards, s'indica que la capçalera "Authorization" que envia el token tinga un prefix "Bearer ", per la qual cosa el contingut d'aqueixa capçalera normalment serà "Bearer .....token.....", i per tant per a obtindre el token caldria processar el valor de la capçalera i tallar els seus primers caràcters.

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

2. Autenticació en projectes complexos

L'exemple anterior ens ha servit per a conéixer els principis bàsics de l'autenticació basada en tokens. Però... què ocorre quan volem protegir diverses rutes disposades en diferents encaminadors? Una opció seria replicar les funcions de *validarToken* o *protegirRuta* en cada encaminador, amb el conseqüent problema de la duplicitat i manteniment d'aqueix codi.

El que se sol fer en aquests casos és traure tot el procés d'autenticació basada en tokens a un mòdul separat i incloure aquest mòdul en els encaminadors o fitxers que ho requerisquen. Seguirem aquests passos en el projecte de contactes. Per a això, prendrem el projecte *ContactesREST_v2* de sessions anteriors i ho copiarem en un altre anomenat *ContactesRESTToken*.

2.1. Nous mòduls i encaminadors

Començarem instal·lant la llibreria *jsonwebtoken* juntament amb les que ja té el projecte (Express i Mongoose), i després crearem dos fitxers en el projecte:

- `utils/auth.js` : on inclourem totes les funcions i mecanismes per a l'autenticació per token

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

const secret = "secretNode";

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

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

let protegirRuta = (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: "Usuari no autoritzat"});
}

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

- `routes/auth.js` : on inclourem el servei POST per a fer el *login* corresponent.

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

let router = express.Router();

// Simulem la base de dades així
const usuaris = [
    { usuari: 'nacho', password: '12345' },
    { usuari: 'alex', password: 'alex111' }
];

router.post('/login', (req, res) => {
    let usuari = req.body.usuari;
    let password = req.body.password;
    let existeixUsuari = usuaris.filter(u =>
        u.usuari == usuari && u.password == password);

    if (existeixUsuari.length == 1)
        res.send({ok: true, token: auth.generarToken(usuari)});
    else
        res.send({ok: false});
});

module.exports = router;
```

2.2. El programa principal

En el programa principal `index.js` haurem de carregar el nou encaminador i associar-lo a alguna ruta (per exemple, `/auth`):

```
...

// Encaminadors
const mascotes = require(__dirname + '/routes/mascotes');
const restaurants = require(__dirname + '/routes/restaurants');
const contactes = require(__dirname + '/routes/contactes');
const auth = require(__dirname + '/routes/auth');

...

// Middleware per a peticions POST i PUT
// Encaminadors per a cada grup de rutes
app.use(express.json());
app.use('/mascotes', mascotes);
app.use('/restaurants', restaurants);
app.use('/contactes', contactes);
app.use('/auth', auth);

// Posada en marxa del servidor
app.listen(8080);
```

2.3. Protegir els serveis desitjats

Si volem protegir qualsevol servei de qualsevol encaminador, n'hi ha prou que carreguem el mòdul `utils/auth` i afegim el *middleware* `protegirRuta` en aquest servei. Per exemple, podem protegir els serveis POST, PUT i DELETE de tots els encaminadors:

Fitxers routes/restaurants.js i routes/mascotes.js

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

// Servei d'inserció
router.post('/', auth.protegirRuta, (req, res) => {
 ...
});

// Servei d'esborrat
router.delete('/:id', auth.protegirRuta, (req,res) => {
 ...
});

module.exports = router;
```

Fitxer routes/contactes.js

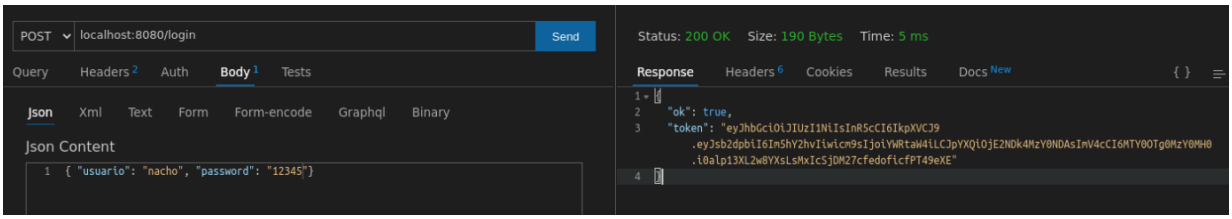
```
const express = require('express');
const auth = require(__dirname + '/../utils/auth.js');
...

// Servei d'inserció
router.post('/', auth.protegirRuta, (req, res) => {
 ...
});

// Servei de modificació
router.put('/:id', auth.protegirRuta, (req, res) => {
 ...
});

// Servei d'esborrat
router.delete('/:id', auth.protegirRuta, (req,res) => {
 ...
});

module.exports = router;
```

3. Prova d'autenticació i altres opcions

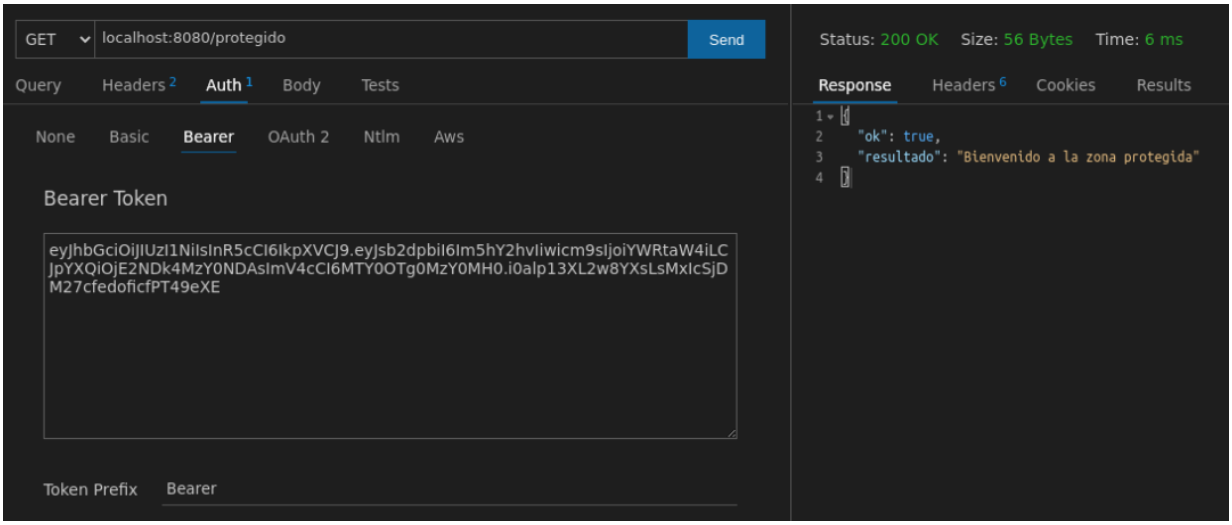
Ara que ja hem vist com incorporar els mecanismes d'autenticació basada en tokens en les nostres aplicacions REST, veurem algunes coses que s'han quedat en el tinter en aquest document.

3.1. Proves d'autenticació amb ThunderClient

Provarem l'aplicació d'exemple amb *ThunderClient* des de Visual Studio Code, i veurem com obtindre i enviar el token d'accés des d'aquesta eina. El primer que haurem de fer és una petició POST per a loguejarnos. Rebrem com a resposta el *token* que s'haja generat:



Ara, només ens queda adjuntar aquest token en la capçalera *Authorization* de les peticions que ho necessiten. Per a això, anem a la secció *Authorization* sota la URL de la petició, i triem que volem enviar un *Bearer token*. En el quadre inferior ens deixarà copiar aquest token:



3.2. Tancament de sessió o *logout*

Per a fer **logout**, com el token ja no s'emmagatzema en el servidor, n'hi ha prou amb eliminar-lo de l'emmagatzematge que tinguem en el client, per la qual cosa és responsabilitat exclusiva del client eixir del sistema, a diferència de l'autenticació basada en sessions, on era el servidor qui havia de destruir la informació emmagatzemada.

3.3. Definir rols d'accés

Per a definir **rols** d'accés, podem afegir un camp del rol que té cada usuari, i emmagatzemar aquest rol en el token, juntament amb el login.

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

Després, bastaria amb modificar el mètode de `protegirRuta` perquè processe el que retorna `validarToken` (l'objecte incrustat en el token) i comprove si té el rol adequat. També hauríem de modificar el mètode `generarToken` perquè reba com a paràmetre el login i rol a afegir al token, i la ruta de POST `/login` perquè li passe aquestes dues dades al mètode de `generarToken`, quan l'usuari siga correcte.

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

...

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

...

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

    let existeixUsuari = usuaris.filter(u =>
        u.usuari == usuari && u.password == password);

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

Exercici 1:

Crea una còpia de l'exercici *LlibresREST_v2* previ, i crida-la **LlibresRESTToken**. El que farem sobre aquest exercici és afegir una autenticació basada en tokens usant la llibreria *jsonwebtoken*.

Definirem un array estàtic d'usuaris registrats amb *login*, *password* i *rol*, que podrà ser *editor* o *admin*, i afegirem la llibreria i els mètodes per a generar i validar el token, com en l'exemple de contactes. Els utilitzarem per a protegir l'accés als serveis que impliquen modificació de dades (POST, PUT i DELETE sobre la col·lecció de llibres i/o autors), de manera que els usuaris *admin* podran accedir a tots aquests

> serveis, i els de tipus *editor* no podran accedir als serveis de modificació d'autors. Per tant, el mètode `protegirRuta` haurà de tindre en compte el rol de l'usuari en alguns casos, com s'ha explicat abans.
>
> Hauràs d'afegir també un servei de *login* (`POST /login`) que reba les dades de l'usuari en el cos de la petició i li retorne el token amb la informació útil guardada (login i rol de l'usuari validat) com en l'exemple de la sessió. Crea una nova col·lecció en *ThunderClient* anomenada *LlibresToken*, i adapta la col·lecció que vas fer originalment, per a utilitzar tokens en els serveis que ho requerisquen, afegint també el servei per al *login*.