# Autenticación basada en sesiones




Como ya comentamos anteriormente, la seguridad basada en sesiones es, quizá, el mecanismo más utilizado para definir autenticación en aplicaciones web tradicionales, basadas en navegador.

Hay que tener en cuenta, antes de nada, que el protocolo HTTP que se emplea en las comunicaciones cliente-servidor es un protocolo *sin estado*, es decir, no se guarda ninguna información, ni existe ninguna relación, entre dos peticiones consecutivas al mismo servidor. Esto dificulta, en principio, el hecho de que el servidor pueda "recordar" quién ha accedido a la web, para dejarle seguir haciéndolo. Los mecanismos de seguridad basados en sesiones añaden un elemento llamado *sesión* a la comunicación cliente-servidor, que permite almacenar información sobre el cliente que accede, de forma que el servidor almacena esa información, y cuando el cliente vuelve a acceder le recuerda, y le da acceso.

Veremos en esta sesión cómo configurar las sesiones en Express y definir mecanismos de autenticación y validación de usuarios.

## 1. Fundamentos de la 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 en la sesión 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.

## 2. 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. Es algo similar a la palabra secreta para cifrar un token, en la autenticación basada en tokens.
- `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.

> **NOTA**: la configuración de la sesión deberá hacerse ANTES de definir los enrutadores, ya que de lo contrario este *middleware* se aplicará después de procesar las rutas, y no tendrá efecto.

## 2.1. 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"});
    }
});
```

## 2.2. 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. 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');
})
```

# 4. 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.

## 4.1. Cierre de sesión o *logout*

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('/');
});
```

## 4.2. Acceder a la sesión desde las vistas

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();
});
```

> **NOTA**: este middleware debe definirse después del middleware que configura la sesión y antes de los enrutadores, para que tenga efecto al renderizar las vistas.

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 %}
```

## 4.3. Tiempo de vida de la sesión

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))
}));
```

Aquí puedes descargar 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*.