Instituto Politécnico Nacional

Omar Montoya Romero

7CM1

WEB CLIENT AND BACKEND DEVELOPMENT FRAMEWORKS

Sistemas Computacionales

Investigación JWT, Sesiones y CORS

Mtro. Efraín Arredondo Morales

19/10/2023

JSON Web Tokens (JWT)

- ¿Qué es un JSON Web Token (JWT)?

  JSON Web Token (JWT) es un estándar abierto (RFC 7519) que define una forma compacta y autónoma de transmitir información de forma segura entre partes como un objeto JSON. Esta información se puede verificar y confiar porque está firmada digitalmente. Los JWT se pueden firmar mediante un secreto (con el algoritmo HMAC) o un par de claves pública/privada mediante RSA o ECDSA. [1]

- Estructura de un JWT.

  En su forma compacta, los JSON Web Tokens constan de tres partes separadas por puntos (.), que son:
  - Encabezamiento
  - Carga útil
  - Firma

  Encabezamiento
  El encabezado normalmente consta de dos partes: el tipo de token, que es JWT, y el algoritmo de firma que se utiliza, como HMAC SHA256 o RSA. Por ejemplo:

  {
  "alg": "HS256",
  "tipo": "JWT"
  }
  Carga útil

  La segunda parte del token es la carga útil, que contiene los reclamos. Los reclamos son declaraciones sobre una entidad (normalmente, el usuario) y datos adicionales. Hay tres tipos de reclamaciones: reclamaciones registradas, públicas y privadas.

  Reclamaciones registradas: son un conjunto de reclamaciones predefinidas que no son obligatorias pero sí recomendadas, para proporcionar un conjunto de reclamaciones útiles e interoperables. Algunos de ellos son: iss (emisor), exp (tiempo de vencimiento), sub (asunto), aud (audiencia) y otros.

  Reclamaciones públicas: aquellas que utilizan JWT pueden definirlas a voluntad. Pero para evitar colisiones, deben definirse en el Registro de tokens web JSON de IANA o definirse como un URI que contenga un espacio de nombres resistente a colisiones.

  Reclamos privados: estos son reclamos personalizados creados para compartir información entre partes que acuerdan usarla y no son reclamos registrados ni públicos.

  Un ejemplo de carga útil podría ser:

  {
    "sub": "1234567890",
    "nombre": "John Doe",
    "administrador": verdadero
  }

Firma

Para crear la parte de la firma, debe tomar el encabezado codificado, la carga útil codificada, un secreto, el algoritmo especificado en el encabezado y firmarlo.

Por ejemplo si desea utilizar el algoritmo HMAC SHA256, la firma se creará de la siguiente manera:

HMACSHA256(
  base64UrlEncode(encabezado) + "." +
  base64UrlEncode (carga útil),
  secreto)
La firma se utiliza para verificar que el mensaje no se modificó en el camino y, en el caso de tokens firmados con una clave privada, también puede verificar que el remitente del JWT es quien dice ser. [1]

- Proceso de creación y verificación de JWT en Node.js.

Las herramientas que vamos a utilizar en este tutorial son:

- Expressjs
- Base de datos MongoDB
- Mongoose
- Dotenv
- Bcryptjs
- Jsonwebtoken

Paso 1

Como primer paso debemos asegurarnos de que se ha instalado **node** y **npm** en su máquina local. Para comprobar si **node** y **npm** están instalados en su ordenador, abra el símbolo del sistema y escriba **node -v** y **npm -v**.

Paso 2

Inicializamos el proyecto con el siguiente comando para crear el archivo JSON

npm init -y

Paso 3

A continuación descargamos los paquetes anteriormente mencionados

npm install express dotenv jsonwebtoken mongoose bcryptjs

Paso 4

Una vez instalados procedemos a la Creación del servidor y conexión de la base de datos

Creamos un archivo index.js el cual debe contener lo siguiente:

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

//Configurar los archivos dotenv de arriba usando cualquier otra librería y archivos
```

```
dotenv.config({path:'./config/config.env'});

//Crear una app desde express
const app = express();

//Usar express.json para obtener la petición de datos json
app.use(express.json());



//Escuchar al servidor
app.listen(process.env.PORT,()=>{
 console.log(`El servidor está escuchando en ${process.env.PORT}`);
})
```

Creamos una carpeta con 2 archivos llamados conn.js y config.env

Los cuales deben de contener

conn.js

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

mongoose.connect(process.env.URI,
 { useNewUrlParser: true,
 useUnifiedTopology: true })
.then((data) => {
 console.log(`Base de datos conectada a ${data.connection.host}`)
})
```

config.env

```
URI = 'mongodb
srv://ghulamrabbani883:rabbani@nodeapi.ulxmv.mongodb.net/?retryWrites=true&w=majority'
PORT = 5000
```

Paso 5

Creamos los Modelos y rutas, donde se crean dos carpetas una de rutas y la otra de modelos, dentro de rutas se crea el archivo userRoute.js y dentro de modelos se crea el archivo userModel.js

userModel:

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

//Crear esquema usando mongoose
const userSchema = new mongoose.Schema({
 name: {
 type:String,
 required:true,
 minLength:[4,'El nombre debe tener un mínimo de 4 caracteres']
 },
 email:{
```

```
type:String,
required:true,
unique:true,
},
password:{
type:String,
required:true,
minLength:[8,'La contraseña debe tener un mínimo de 8 caracteres']
},
token:{
type:String
}
})

//Creación de modelos
const userModel = mongoose.model('user',userSchema);
module.exports = userModel;
```

userRoute.js

```
const express = require('express');
//Creación del enrutador express
const route = express.Router();
//Importación de userModel
const userModel = require('../models/userModel');

//Creación de la ruta de registro
route.post('/register',(req,res)=>{

})
//Creación de rutas de inicio de sesión
route.post('/login',(req,res)=>{

})

//Creación de rutas de usuario para obtener los datos de los usuarios
route.get('/user',(req,res)=>{

})
```

Paso 6
Implementamos la funcionalidad de las rutas y tokens

```
//Requiriendo todos los archivos y librerías necesarios
const express = require('express');
const bcrypt = require('bcryptjs');
const jwt = require('jsonwebtoken');

//Creando el enrutador express
const route = express.Router();
//Importar userModel
const userModel = require('../models/userModel');
```

```
//Crear ruta de registro
route.post("/register", async (req, res) => {

 try {
 const { name, email, password } = req.body;
 //Comprobar si los datos entrantes están vacíos
 if (!name || !email || !password) {
 return res.json({ message: 'Please enter the details' })
 }

      //Comprobar si el usuario ya existe o no
 const userExist = await userModel.findOne({ email: req.body.email });
 if (userExist) {
 return res.json({ message: 'El usuario ya existe con el emailId dado' })
 }
      //Hash de la contraseña
 const salt = await bcrypt.genSalt(10);
 const hashPassword = await bcrypt.hash(req.body.password, salt);
 req.body.password = hashPassword;
 const user = new userModel(req.body);
 await user.save();
 const token = await jwt.sign({ id: user._id }, process.env.SECRET_KEY, {
 expiresIn: process.env.JWT_EXPIRE,
 });
 return res.cookie({ 'token': token }).json({ success: true, message: 'Usuario registrado con éxito', data:
user })
 } catch (error) {
 return res.json({ error: error });
 }

})
//Creando rutas de inicio de sesión
route.post('/login', async (req, res) => {
 try {
 const { email, password } = req.body;
 //Comprobar si los datos entrantes están vacíos
 if (!email || !password) {
 return res.json({ message: 'Por favor, introduzca todos los datos' })
 }
      //Comprobar si el usuario ya existe o no
 const userExist = await userModel.findOne({email:req.body.email});
 if(!userExist){
 return res.json({message:'Credenciales erróneas'})
 }
      //Comprobar coincidencia de contraseña
 const isPasswordMatched = await bcrypt.compare(password,userExist.password);
 if(!isPasswordMatched){
 return res.json({message:'Credenciales incorrectas pasan'});
 }
 const token = await jwt.sign({ id: userExist._id }, process.env.SECRET_KEY, {
 expiresIn: process.env.JWT_EXPIRE,
```

```
});
return res.cookie({"token":token}).json({success:true,message:'LoggedIn Successfully'})
} catch (error) {
return res.json({ error: error });
}

})
```

Config.env

```
URI = 'mongodb
srv://ghulamrabbani883:rabbani@nodeapi.ulxmv.mongodb.net/?retryWrites=true&w=majority'
PORT = 5000
SECRET_KEY = KGGK>HKHVHJVKBKJKJBKBKHKBMKHB
JWT_EXPIRE = 2d
```

Index.js

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

//Configurar los archivos dotenv anteriores utilizando cualquier otra biblioteca y archivos
dotenv.config({path:'./config/config.env'});
require('./config/conn');
//Crear una aplicación desde express
const app = express();
const route = require('./routes/userRoute');

//Usar express.json para obtener la petición de datos json
app.use(express.json());
//Usar rutas

app.use('/api', route);

//escuchar al servidor
app.listen(process.env.PORT,()=>{
 console.log(`El servidor está escuchando en ${process.env.PORT}`);
})
```

Paso 7

Creamos el middleware para la autenticación, donde se va a crear una carpeta llamada middleware y dentro de ella se crea el archivo auth.js

```
const userModel = require('../models/userModel');
const jwt = require('jsonwebtoken');
const isAuthenticated = async (req,res,next)=>{
 try {
 const {token} = req.cookies;
 if(!token){
 return next('Por favor, identifíquese para acceder a los datos');
 }
 const verify = await jwt.verify(token,process.env.SECRET_KEY);
 req.user = await userModel.findById(verify.id);
```

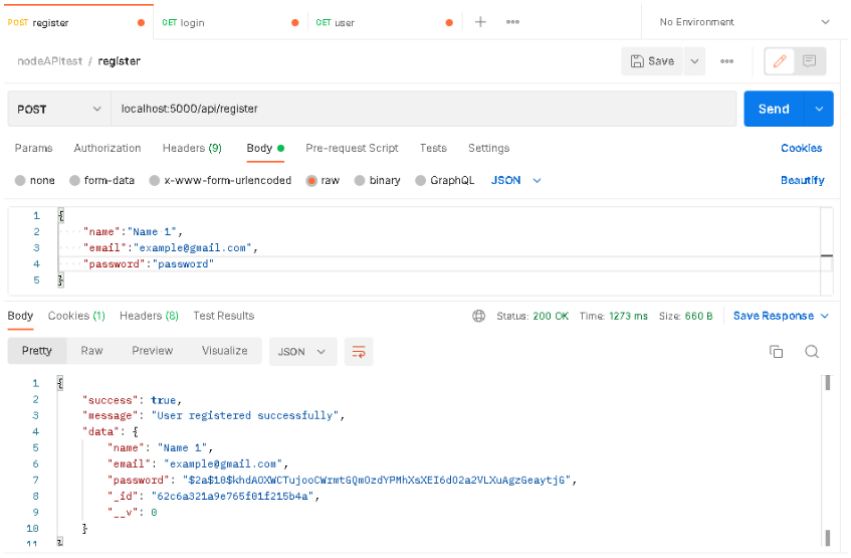
```
next();
} catch (error) {
return next(error);
}
}

module.exports = isAuthenticated;
```

procedemos a instalar cookieParse para la configuración de obtención de tokens

```
npm i cookie-parser
```

Una vez instalado lo implementamos en el usertoute e index.

Paso 8
Comprobamos que funcione en postman, pero para eso ponemos el comando npm start para prenderlo y poder hacer consultas.



[2]

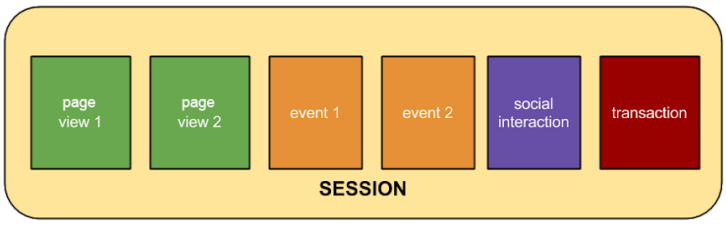
- Uso de JWT para autenticación y autorización.

  Para autenticar un usuario, una aplicación cliente debe enviar un token web JSON (JWT) en el encabezado de autorización de la solicitud HTTP a tu API de backend. API Gateway valida el token en nombre de la API, por lo que no es necesario agregar ningún código a la API para procesar la autenticación. Sin embargo, debes ajustar la configuración de la API para que la puerta de enlace sea compatible con los métodos de autenticación que elegiste. [3]

Sesiones en Node.js

- ¿Qué son las sesiones en aplicaciones web?

  Se puede decir que una sesión es el elemento que engloba las acciones del usuario en su sitio web.



Un único usuario puede abrir varias sesiones, ya sea el mismo día o en varios días, semanas o meses. En cuanto finaliza una sesión, es posible empezar una sesión nueva. El vencimiento de las sesiones puede basarse en:

- El tiempo:
  - Después de 30 minutos de inactividad
  - A medianoche
- La campaña:
  - Si el usuario llega al sitio a través de una campaña, lo abandona y, a continuación, regresa al sitio a través de otra campaña. [4]

- Implementación de sesiones en Node.js.

  Paso 1

  Instalamos la dependencia express-session

  ```
  npm install --save express-session
  ```

  Una vez instalamos lo importamos

  ```
  const session = require("express-session");
  ```

  Paso 2

  Instanciamos el middleware que se va a usar en la sesión es decir una clave secreta

  ```
  // Primero usar la sesión
  app.use(session({
    // Se recomienda cambiar en cada entorno
    // https://parzibyte.me/blog/2020/06/02/sesiones-node-express-js/#Configurar_uso_de_sesion
    secret: "123",
    saveUninitialized: true,
    resave: true,
    cookie:{
      maxAge: 60000,
    }
  }));
  // Y después definir las rutas!
  app.use('/', indexRouter);
  app.use('/productos', productosRouter);
  app.use('/usuarios', usuariosRouter);
  ```

  Paso 3

Creamos el método de inicio de sesión para hacer la consulta de la siguiente manera

```
app.get("/guardar_en_sesión", (req, res) => {
  req.session.nombre = "Luis";
  req.session.edad = 23;
  req.session.web = "https://parzibyte.me";
});
```

Paso 4

Creamos la función de leer los datos de la sesión

```
app.get("/ruta_solo_logueados", (req, res) => {
  // Si, por ejemplo, no hay nombre
  if(!req.session.nombre){
    res.end("No tienes permiso. Fuera de aquí");
  }else{
    // Ok, el usuario tiene permiso
    res.end("Hola " + req.session.nombre);
  }
}); [5]
```

- Uso de cookies y almacenamiento de sesiones.

  Las cookies y sesiones son mecanismos a través de los cuales podemos identificar una petición, con estas estrategias podemos almacenar información de nuestros usuarios que pueda ayudarnos como los productos que ha agregado a un carrito de compras, sus preferencias, si inició sesión o no, y mucho más.

  Las cookies son datos que se almacenan en tu navegador, y son enviados al servidor en cada petición que haces del cliente al servidor mismo. Esto significaría, por ejemplo, que cada que llegas con el servidor sin memoria te presentes y le digas toda la información que posiblemente requería de ti.

  Las sesiones por otro lado guardan la información en el servidor y no en el cliente, y lo que se envía entre cada petición es un identificador de sesión para cada usuario, a través del cual puedes obtener los datos que guardaste en el servidor. Esto significaría, por ejemplo, que cuando llegues con el servidor sin memoria le entregues una identificación tuya y él busque entre sus datos toda la información que necesitas saber, en lugar de que tú tengas que decirle todo. [6]

- Seguridad en el manejo de sesiones.

  El manejo de la sesión es uno de los aspectos críticos de la seguridad WEB. Los objetivos principales son:
  - Los usuarios autenticados tengan una asociación con sus sesiones robusta y criptográficamente segura .
  - Se hagan cumplir los controles de autorización.
  - Se prevengan los típicos ataques web, tales como la reutilización, falsificación e intercepción de sesiones.

  Se detectan las siguientes vulnerabilidades significativas.
  - **Fijación de sesión** que intenta explotar la vulnerabilidad de un sistema que permite a una persona fijar el identificador de sesión de otra persona. La mayoría de los ataques de fijación del

período de sesiones están basados en la web, y la mayoría depende de los identificadores de sesión que han sido aceptados de URL o datos POST

- **Identificador de la sesión vulnerable**, si no se reserva un tamaño adecuado el atacante, mediante técnicas de fuerza bruta atacante puede conocer el identificador de una sesión autenticada y por lo tanto hacerse con el control de la sesión.
- **Manejo de la información de sesión errónea**, ya sea por estar en un espacio compartido o mal encriptada, el atacante puede obtener datos de la sesión de otro usuario. [7]

Cross-Origin Resource Sharing (CORS) en Node.js

- ¿Que es Cross-Origin Resource Sharing (CORS)?

  El uso compartido de recursos entre orígenes (CORS) es un mecanismo para integrar aplicaciones. CORS define una forma con la que las aplicaciones web clientes cargadas en un dominio pueden interactuar con los recursos de un dominio distinto. Esto resulta útil porque las aplicaciones complejas suelen hacer referencia a API y recursos de terceros en el código del cliente. [8]

- Importancia de CORS en aplicaciones web y API.

  - **Protección contra ataques de terceros:** CORS ayuda a prevenir que un sitio web malicioso obtenga información sensible de otro sitio web al evitar que las solicitudes se realicen desde un origen diferente.
  - **Seguridad en API:** Para las API, CORS permite especificar qué dominios pueden acceder a los recursos, lo que es crucial para proteger la información que se expone a través de la API.
  - **Compatibilidad con navegadores modernos:** Los navegadores modernos aplican estrictamente las políticas de CORS, lo que significa que si no se configura correctamente, ciertas funcionalidades pueden fallar. [9]

- Configuración de CORS en Node.js.

  Cuando hacemos peticiones AJAX con jQuery o Angular a un backend o un API REST es normal que tengamos problemas con el acceso CORS en NodeJS y nos fallen las peticiones.

  Para eso podemos crear un middleware como este:

```
//
Configurar
cabeceras y
cors
app.use((req,
res, next) => {
    res.header('Access-
Control-Allow-Origin', '*');
    res.header('Access-Control-
Allow-
Headers', 'Authorization, X-
API-KEY, Origin, X-
```

```
Requested-With, Content-
Type, Accept, Access-Control-
Allow-Request-Method');
   res.header('Access-
Control-Allow-
Methods', 'GET, POST,
OPTIONS, PUT,
DELETE');
   res.header('Allow', 'GET, POST,
OPTIONS, PUT, DELETE');
   next();
});
```

[10]

- Seguridad y políticas de acceso.

  Una política de control de acceso es un conjunto de condiciones que, una vez evaluadas, determinan las decisiones de acceso.
  Las condiciones son una combinación de atributos, obligaciones, políticas de autenticación y un perfil de riesgo.
  Antes de crear una política, revise los atributos disponibles, las obligaciones, las políticas de autenticación y los perfiles de riesgo en la interfaz de gestión local para determinar si satisfacen las necesidades de la política.

- Gestión de políticas de control de acceso
  - Una política de control de acceso es un conjunto de condiciones que definen si se permite o se deniega a un usuario el acceso a un recurso protegido.
- Creación de una política de control de acceso
  - Utilice el Editor de políticas en la interfaz de gestión local del dispositivo para crear y configurar una política de control de acceso.
- Gestión de conjuntos de políticas de control de acceso
  - Un conjunto de políticas es un grupo de políticas que se utilizan conjuntamente para proteger un recurso.
- Gestión de conexiones de políticas de control de acceso
  - Conectar políticas o definiciones de protección de API a recursos para que se puedan aplicar las políticas y definiciones.
- Escenarios de política
  - Se proporcionan varios escenarios de política utilizados habitualmente como ejemplos para ayudarle a crear políticas. [11]

Conclusión

Tras la investigación me doy cuenta de la importancia y el funcionamiento de las cookies y las sesiones ya que facilitan mucho la obtención de información del usuario pero tienen la diferencia de que una no es muy segura e implica que el usuario este ingresando los datos de nuevo, mientras que en las sesiones las guarda y directamente se obtienen los datos haciendo que el sitio web funcione más rápido, estas son seguras pero no quita el peligro de que se encuentre una vulnerabilidad y esos datos se vean expuestos, los tokens son un recurso interesante pero considero que de doble filo ya que son aleatorios y al crear y olvidarse de uno, en caso de ser administrador o desarrollador implica un problema ya que estos son ocultos o aleatorios.