Nest

Autenticación con JWT

CertiDevs

Índice de contenidos

1. Introducción a JWT

JWT (JSON Web Token) es un estándar abierto (RFC 7519) para la creación de tokens de acceso que permiten la **autenticación** y **autorización** seguras entre dos partes, como un cliente y un servidor.

Sitio web oficial de JWT

JWT se utiliza comúnmente para transmitir información de **identidad** y **privilegios** en aplicaciones web, APIs y servicios.

JWT es un método compacto y seguro para transmitir información entre dos partes como un **objeto JSON**, que se firma **digitalmente** y, opcionalmente, se cifra. La **firma digital** garantiza que el **emisor del token** (generalmente, un servidor de autenticación) pueda **verificar la integridad** de los datos y asegurarse de que no se hayan modificado durante la transmisión.

2. Estructura JWT

La **estructura** de un **token JWT** consta de **tres partes**, separadas por puntos (.):

2.1. Encabezado

Encabezado (Header): El encabezado es un objeto JSON que contiene información sobre el tipo de token (generalmente JWT) y el algoritmo de firma utilizado (como HS256 o RS256). El encabezado se codifica en base64 y se utiliza para verificar la firma del token.

Ejemplo de encabezado:

```
{
"alg": "HS256",
"typ": "JWT"
}
```

2.2. Payload

Carga útil (Payload): La carga útil es un objeto JSON que contiene los llamados "claims" (reclamaciones), que son declaraciones sobre el sujeto (usuario) y otros metadatos.

Los claims pueden ser de **tres tipos**: claims registrados, públicos y privados.

Los claims registrados son un conjunto de claims predefinidos por el estándar JWT, como "iss" (emisor), "exp" (tiempo de expiración) y "sub" (sujeto).

Los claims públicos y privados son específicos de la aplicación y pueden contener información como roles de usuario, permisos, etc.

Al igual que el encabezado, la carga útil se codifica en **base64**.

Ejemplo de carga útil:

```
{
"sub": "1234567890",
"name": "John Doe",
"iat": 1516239022,
"exp": 1516242622,
"role": "admin"
}
```

2.3. Firma

Firma (Signature): La firma se crea utilizando el encabezado codificado en base64, la carga útil codificada en base64 y una clave secreta o un par de claves pública/privada, dependiendo del algoritmo de firma utilizado.

La firma es esencial para verificar la integridad y autenticidad del token JWT.

Ejemplo de cómo calcular la firma (usando el algoritmo HS256 y una clave secreta):

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret_key
)
```

Cuando se combinan, estas tres partes forman el token JWT completo, que se ve así:

```
[header_base64].[payload_base64].[signature]
```

3. Cómo funciona JWT

Para utilizar **JWT** en una aplicación, un **cliente** (por ejemplo, un navegador web o una aplicación móvil) solicita un **token** a un **servidor** de autenticación proporcionando sus credenciales (como un nombre de usuario y contraseña).

Si las **credenciales** son válidas, el **servidor** crea y firma un token JWT que contiene información sobre el usuario y lo envía al **cliente**.

El **cliente** almacena el **token** y lo utiliza en solicitudes posteriores al **servidor**, que valida el **token** antes de procesar la solicitud.

Si el **token** es válido, el **servidor** procesa la solicitud según los privilegios del usuario representados por el token.

Si el token no es válido o ha expirado, el **servidor** puede denegar el acceso y solicitar al **cliente** que

vuelva a autenticarse.

Aquí hay un ejemplo de flujo de autenticación y autorización utilizando JWT:

1. El cliente envía sus credenciales (nombre de usuario y contraseña) al servidor de autenticación en una solicitud de inicio de sesión.
2. El servidor de autenticación verifica las credenciales y, si son válidas, crea un token JWT que contiene información sobre el usuario (claims) y firma el token con una clave secreta o un par de claves pública/privada.
3. El servidor devuelve el token JWT al cliente.
4. El cliente almacena el token JWT y lo incluye en el encabezado Authorization de las solicitudes posteriores al servidor, generalmente como un "Bearer Token".
5. El servidor verifica la firma del token JWT en cada solicitud y comprueba si el token ha expirado. Si el token es válido, el servidor procesa la solicitud según los privilegios del usuario representados por el token. Si el token no es válido o ha expirado, el servidor deniega el acceso y puede solicitar al cliente que vuelva a autenticarse.

Algunas ventajas de utilizar JWT para la autenticación y autorización incluyen:

- Autosuficiencia: Los tokens JWT contienen toda la información necesaria para autenticar y autorizar a un usuario, lo que elimina la necesidad de consultar una base de datos o un sistema de almacenamiento externo en cada solicitud.
- Escalabilidad: Dado que los tokens JWT son autosuficientes y no requieren acceso a un sistema de almacenamiento centralizado, son ideales para aplicaciones distribuidas y escalables.
- Compatibilidad entre dominios: A diferencia de las cookies, los tokens JWT pueden transmitirse fácilmente entre diferentes dominios, lo que facilita la implementación de la autenticación y autorización en aplicaciones con múltiples servicios y dominios.

Es importante tener en cuenta que, aunque JWT proporciona un mecanismo seguro para transmitir información entre dos partes, los tokens deben tratarse como datos sensibles y protegerse adecuadamente. Por ejemplo, se debe utilizar HTTPS para transmitir tokens JWT en aplicaciones web y no se debe almacenar información confidencial en la carga útil del token, ya que los datos codificados en base64 pueden decodificarse fácilmente.

4. JWT en NestJS

La **autenticación** con **JWT** (JSON Web Token) es un enfoque común para proteger rutas y recursos en aplicaciones web.

En este caso, vamos a implementar la **autenticación JWT** en NestJS para un backend que será consumido por una aplicación Angular.

4.1. Instala las dependencias necesarias:

```
npm install @nestjs/jwt @nestjs/passport passport passport-jwt
```

4.2. Crea un módulo

Crea un módulo `AuthModule` para manejar la autenticación:

```
nest generate module auth
```

4.3. Crea un servicio

Crea un servicio `AuthService` para manejar la lógica de autenticación:

```
nest generate service auth
```

Implementa el servicio AuthService.

En este ejemplo, utilizaremos un método para validar las credenciales del usuario y otro para generar un JWT:

Archivo `auth.service.ts`:

```
import { Injectable } from '@nestjs/common';
import { JwtService } from '@nestjs/jwt';
import { UserService } from '../user/user.service';

@Injectable()
export class AuthService {
constructor(
private readonly userService: UserService,
private readonly jwtService: JwtService,
) {}

  async validateUser(username: string, password: string): Promise<any> {
    const user = await this.userService.findOne(username);
    if (user && user.password === password) {
      const { password, ...result } = user;
      return result;
    }
    return null;
  }

  async login(user: any) {
    const payload = { username: user.username, sub: user.id };
    return {
      access_token: this.jwtService.sign(payload),
```

```
    };
  }
}
```

Aquí, `validateUser` es un método que valida las credenciales del usuario.

Si las credenciales son correctas, devuelve el objeto user sin la contraseña. El método login genera un JWT utilizando el módulo JwtService.

4.4. Configura Passport y la estrategia JWT:

Crea un archivo `jwt.strategy.ts` en la carpeta auth:

Archivo `jwt.strategy.ts`:

```
import { Injectable } from '@nestjs/common';
import { PassportStrategy } from '@nestjs/passport';
import { ExtractJwt, Strategy } from 'passport-jwt';
import { AuthService } from './auth.service';
import { jwtConstants } from './constants';

@Injectable()
export class JwtStrategy extends PassportStrategy(Strategy) {
    constructor(private readonly authService: AuthService) {
        super({
        jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
        ignoreExpiration: false,
        secretOrKey: jwtConstants.secret,
        });
    }

    async validate(payload: any) {
        return { id: payload.sub, username: payload.username };
    }
}
```

En este archivo, configuramos la estrategia JWT de Passport para extraer el token JWT del encabezado Authorization y validarlo utilizando la clave secreta.

Crea un archivo `constants.ts` en la carpeta auth y define una constante para el secreto JWT:

Archivo `constants.ts`:

```
export const jwtConstants = {
    secret: 'your-secret-key',
};
```

Configura `AuthModule`:

Archivo `auth.module.ts`:

```
import { Module } from '@nestjs/common';
import { AuthService } from './auth.service';
import { JwtModule } from '@nestjs/jwt';
import { jwtConstants } from './constants';
import { JwtStrategy } from './jwt.strategy';
import { UserModule } from '../user/user.module';

@Module({
    imports: [
        UserModule,
        JwtModule.register({
            secret: jwtConstants.secret,   signOptions: { expiresIn: '1h' },
        }),
    ],
    providers: [AuthService, JwtStrategy],
    exports: [AuthService],
})
export class AuthModule {}
```

En `AuthModule`, importamos `UserModule` y `JwtModule`. Configuramos `JwtModule` con la clave secreta y la duración de la sesión (1 hora en este ejemplo). También agregamos `AuthService` y `JwtStrategy` como proveedores.

4.5. Implementa el controlador de autenticación

Crea un archivo `auth.controller.ts` en la carpeta `auth`:

`auth.controller.ts`:

```
import { Controller, Request, Post, UseGuards } from '@nestjs/common';
import { LocalAuthGuard } from './local-auth.guard';
import { AuthService } from './auth.service';

@Controller('auth')
export class AuthController {
  constructor(private readonly authService: AuthService) {}

  @UseGuards(LocalAuthGuard)
  @Post('login')
  async login(@Request() req) {
    return this.authService.login(req.user);
  }
}
```

Aquí, utilizamos un **guard** personalizado `LocalAuthGuard` para proteger la ruta login.

Este **guard** se encarga de llamar al método `validate` de `AuthService` para autenticar al usuario.

4.6. Implementa el guard personalizado LocalAuthGuard

Crea un archivo `local-auth.guard.ts` en la carpeta **auth**:

Archivo `local-auth.guard.ts`:

```
import { Injectable } from '@nestjs/common';
import { AuthGuard } from '@nestjs/passport';

@Injectable()
export class LocalAuthGuard extends AuthGuard('local') {}
```

Este **guard** extiende `AuthGuard` de `Passport` y utiliza la **estrategia** `local` para autenticar al usuario.

4.7. Protege las rutas en tu aplicación NestJS

Puedes proteger las rutas que requieran autenticación utilizando el decorador `@UseGuards` y el guard `JwtAuthGuard`.

Crea un archivo `jwt-auth.guard.ts` en la carpeta **auth**:

Archivo `jwt-auth.guard.ts`:

```
import { Injectable } from '@nestjs/common';
import { AuthGuard } from '@nestjs/passport';

@Injectable()
export class JwtAuthGuard extends AuthGuard('jwt') {}
```

Luego, aplica este guard a cualquier controlador o ruta que requiera autenticación:

```
import { Controller, Get, UseGuards } from '@nestjs/common';
import { JwtAuthGuard } from '../auth/jwt-auth.guard';

@Controller('protected')
export class ProtectedController {
  @UseGuards(JwtAuthGuard)
  @Get()
  getProtectedData() {
    return { message: 'Este es un recurso protegido' };
  }
}
```

Configura la aplicación Angular para enviar el token JWT en el encabezado `Authorization`:

En tu aplicación Angular, cuando el usuario inicie sesión exitosamente y reciba un JWT, debes almacenarlo en el almacenamiento local del navegador:

```
localStorage.setItem('access_token', response.access_token);
```

Luego, crea un interceptor HTTP para agregar automáticamente el **token JWT** al encabezado `Authorization` de cada solicitud:

```
import { Injectable } from '@angular/core';
import { HttpInterceptor, HttpRequest, HttpHandler } from '@angular/common/http';

@Injectable()
export class JwtInterceptor implements HttpInterceptor {
    intercept(request: HttpRequest<any>, next: HttpHandler) {
        const token = localStorage.getItem('access_token');
        if (token) {
            request = request.clone({
                setHeaders: {
                    Authorization: `Bearer ${token}`,
                },
            });
        }
        return next.handle(request);
    }
}
```

Registra el interceptor en el módulo principal de tu aplicación Angular:

Archivo `app.module.ts`:

```
import { HttpClientModule, HTTP_INTERCEPTORS } from '@angular/common/http';
import { JwtInterceptor } from './jwt.interceptor';

@NgModule({
    // ...
    imports: [
        // ...
        HttpClientModule,
    ],
    providers: [
        // ...
        {
        provide: HTTP_INTERCEPTORS,
        useClass: JwtInterceptor,
        multi: true,
        },
```

```
    ],
    // ...
})
export class AppModule {}
```

Con estos pasos, has implementado la autenticación con JWT en NestJS y configurado tu aplicación Angular para enviar el token JWT en el encabezado Authorization de cada solicitud. Ahora, tu aplicación Angular puede consumir recursos protegidos del backend de NestJS utilizando tokens JWT para autenticarse.