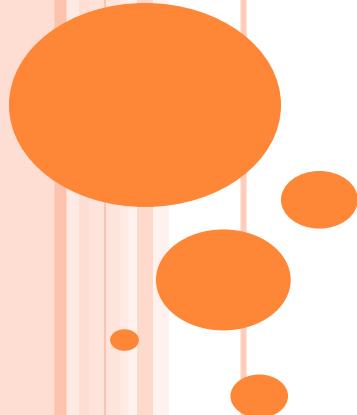


PROGRAMACIÓN EN EL SERVIDOR CON NODEJS/NEST



FUNDAMENTOS NEST

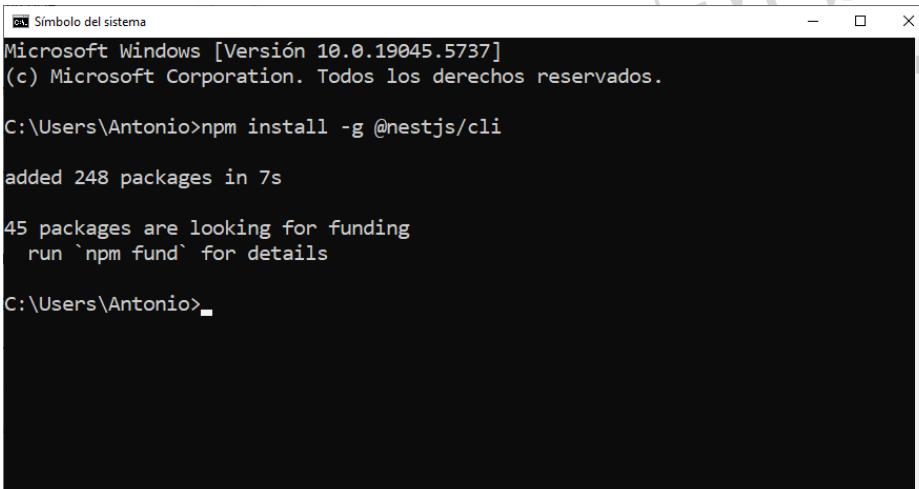
- Framework para la creación de aplicaciones de lado de servidor (backend), basado en Node.js.
- Utiliza TypeScript como lenguaje de programación.
- Inspirado en la filosofía de Angular para el desarrollo de aplicaciones MVC
- Soporte para HTTP
- Integración con bases de datos



INSTALACIÓN

- Se requiere tener instalado Node.js con npm.
- Para instalar Nest:

```
>npm install -g @nestjs/cli
```



The screenshot shows a Windows Command Prompt window titled "Símbolo del sistema". The window displays the following text:

```
Microsoft Windows [Versión 10.0.19045.5737]
(c) Microsoft Corporation. Todos los derechos reservados.

C:\Users\Antonio>npm install -g @nestjs/cli

added 248 packages in 7s

45 packages are looking for funding
  run `npm fund` for details

C:\Users\Antonio>
```

CREACIÓN DE UN PROYECTO NEST

- Para crear un proyecto Nest, nos colocaremos sobre la carpeta en la que queremos que esté y escribimos:

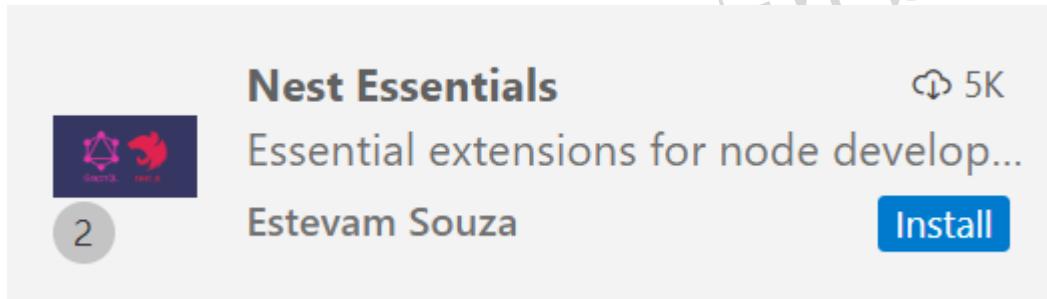
```
>nest new primer_proyecto
```

- Si nos pregunta el tipo de gestor de paquetes a utilizar, elegiremos npm
- Se creará un proyecto básico con esta composición de archivos en su carpeta src:

nest > primer_proyecto > src		
Nombre	^	Fecha
app.controller.spec.ts		02/05/2025 11:45
app.controller.ts		02/05/2025 11:45
app.module.ts		02/05/2025 11:45
app.service.ts		02/05/2025 11:45
main.ts		02/05/2025 11:45

NEST EN VISUAL STUDIO CODE

- Si queremos desarrollar proyectos Nest en Visual Studio Code, hay diversos plugins que nos pueden ayudar.
- El Nest Essentials agrupa varios de esos plugins que nos facilitan la programación de aplicaciones Nest:



EJECUCIÓN

- Para ejecutar una aplicación Nest, nos situamos en la carpeta raíz del proyecto y escribimos:

```
>npm run start:dev
```

- Se iniciará el servidor de aplicaciones en el puerto 3000 y podremos acceder a la dirección raiz:

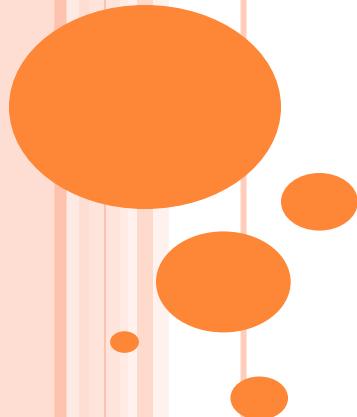
```
http://localhost:3000
```

- Se visualizará la respuesta generada por el controlador, que no es más que la página con un mensaje "Hello World!"

- Se puede modificar el puerto en main.js



CONTROLADOR



ESTRUCTURA Y FUNCIÓN

- El controlador se encarga de manejar las peticiones HTTP y generar respuestas
- Se define en una clase con la siguiente estructura:

Define la clase como un controlador y le asocia una dirección base

```
@Controller('libros')
export class LibrosController {
  @Post("alta")
  create(@Body() libro:LibroModel) {
    }
  @Get("catalogo")
  findAll() {
  }
  @Get("buscar/:id")
  findOne(@Param('id') id: string) {
  }
}
```

Rutas específicas de cada recurso

Mapea el cuerpo JSON a un objeto

PathVariable

RECOGIDA DE DATOS DE PETICIÓN

- En los métodos del controlador se recogen los datos enviados en la petición a través de parámetros
- Estos datos pueden venir:
 - Como Path variables (url/variable):

```
@Get("buscar/:id")
findOne(@Param('id') id: string) {...}
```
 - Como parámetros en querystring(url?param=value):

```
@delete("eliminar")
deleteOne(@Query('id') id: string) {...}
```
 - Como JSON o Form url-endcoded en el body:

```
@post ("agregar")
create(@Body() data: Persona) {...}
```

POLITICA CORS

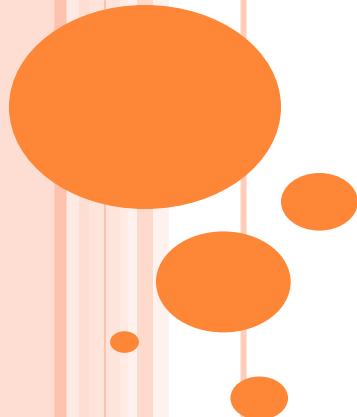
- La política CORS, o política de origen cruzado, se basa en bloquear peticiones que llegan a un servicio, procedentes de un origen diferente (por ejemplo, el front)
- Para eliminar esta restricción se añade lo siguiente en el archivo main.ts:

```
async function bootstrap() {
  const app = await NestFactory.create(AppModule);

  // Habilita CORS para todos los orígenes
  app.enableCors();

  await app.listen(3000);
}
```

SERVICIO



ESTRUCTURA Y FUNCIÓN

- El servicio encapsula la lógica de negocio de la aplicación.
- Se define de forma similar a los servicios Angular:

```
@Injectable()
export class LibrosService {
    libros:LibroModel[]=[];
    create(libro: LibroModel):void {
        this.libros.push(libro);
    }

    findAll():LibroModel[] {
        return this.libros;
    }
    :
}
```

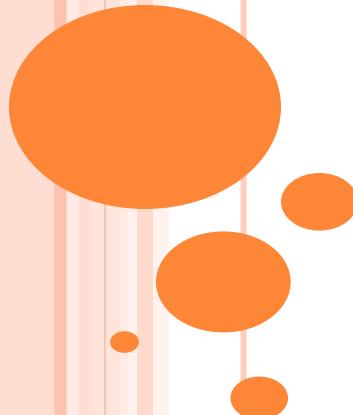
- Se inyecta en el controller a través del constructor.

REGISTRO EN MÓDULO

- Para que puedan ser reconocidos por la aplicación, los servicios y controladores deben registrarse en el archivo de módulo AppModule:

```
@Module({  
    imports: [],  
    controllers: [LibrosController],  
    providers: [LibrosService],  
})  
export class AppModule {}
```

ACCESO A BASES DE DATOS



TYPEORM

- **Librería para acceder a bases de datos relaciones desde TypeScript.**
- **Está basada en ORM.**
- **Para instalar TypeORM, junto con el controlador de MySQL:**

¡Para cada proyecto!

```
>npm install @nestjs/typeorm@11 typeorm@0.3 mysql2
```

- **Para MySQL, es necesario crear un usuario que use el plugin mysql_native_password, que es el compatible con TypeORM:**

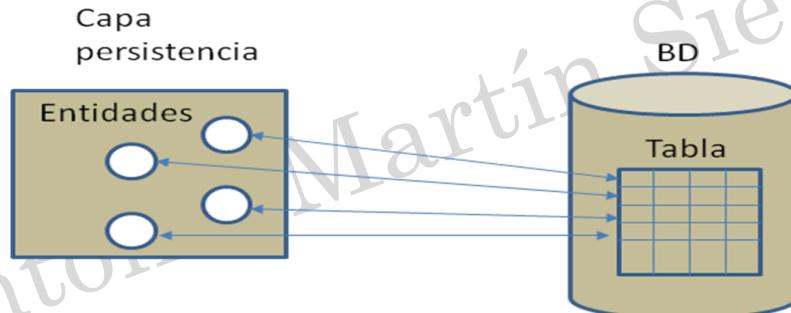
```
CREATE USER 'nestuser'@'localhost' IDENTIFIED WITH mysql_native_password BY 'nestpass';
GRANT ALL PRIVILEGES ON libros.* TO 'nestuser'@'localhost';
FLUSH PRIVILEGES;
```

- **Si ya existe el usuario:**

```
GRANT ALL PRIVILEGES ON formacion.* TO 'nestuser'@'localhost';
FLUSH PRIVILEGES;
```

ENTIDADES

- El acceso a los datos se realiza a través de entidades.
- Una entidad es un objeto que representa una fila de una tabla de la base de datos:



CREACIÓN DE UNA ENTIDAD

- Se definen a través de una clase que encapsula los datos de la entidad.
- Se configura a través de una serie de decoradores:

campo clave
primaria

```
import {Entity, PrimaryColumn, Column,} from 'typeorm';
@Entity("libros")
export class LibroModel {
    @PrimaryColumn()
    isbn:string;
    @Column()
    titulo:string;
    @Column()
    precio:number;
```



CONFIGURACIÓN DE LA CONEXIÓN

➤ Los datos de conexión se indican en el módulo de la aplicación en la sección de importaciones:

```
@Module({
    imports: [TypeOrmModule.forRoot({
        type: 'mysql',
        host: 'localhost',
        port: 3307,
        username: 'nestuser',
        password: 'nestpass',
        database: 'libros',
        entities: [LibroModel],
        synchronize: false,
    }), TypeOrmModule.forFeature([LibroModel])],
    controllers: [LibrosController],
    providers: [AppService],
})
```

También se
importa la
entidad

REPOSITORIO

- Para acceder a datos mediante ORM, el módulo TypeORM dispone del objeto Repository.
- Se debe inyectar en la capa service utilizando `@InjectRepository`:

```
constructor(@InjectRepository(LibroModel)
            private readonly librosRepository: Repository<LibroModel>){  
}
```

- El objeto proporciona una serie de métodos para operar con entidades.

MÉTODOS DE REPOSITORY

- **save(entidad):Promise<Entidad>.** Guarda o actualiza la entidad en la base de datos
- **find():Promise<Entidad[]>.** Devuelve todas las entidades
- **findBy(where):Promise<Entidad[]>.** Recupera todas las entidades en función de una condición que se establece a través de un JSON.
- **findOneBy(where):Promise<Entidad>.** Igual que el anterior, pero devolviendo solo una entidad
- **remove(Entidad):Promise<Entidad>.** Elimina la entidad
- **delete(where):Promise<DeleteResult>.** Elimina las entidades que cumplen la condición
- **update(codicion,valores).** Actualiza entidades

EJEMPLO SERVICE CON REPOSITORY

```
@Injectable()
export class AppService {
    constructor(@InjectRepository(LibroModel)
                private readonly librosRepository: Repository<LibroModel>){
    }
    create(libro: LibroModel):Promise<LibroModel> {
        return this.librosRepository.save(libro)
    }
    findAll():Promise<LibroModel[]> {
        return this.librosRepository.find();
    }
    findByIsbn(isbn: string) :Promise<LibroModel>{
        return this.librosRepository.findOneBy({isbn:isbn});
    }
    findByPrecioAndPaginas(precio:number, paginas:number) :Promise<LibroModel>{
        return this.librosRepository.findOneBy({where: {
            precio: LessThan(30),
            paginas: MoreThan(100),
        },});
    }
    deleteByPrecioMax(precio:number):Promise<DeleteResult>{
        return this.repository.delete({precio: MoreThan(30)});
    }
}
```

RELACIONES ENTRE ENTIDADES

- Se pueden relacionar las entidades para poder operar sobre entidades en función de condiciones que afecten a entidad relacionada.
- Las relaciones pueden ser uno a muchos/muchos a uno
- Muchos a muchos
- Para relacionar entidades, cada entidad contiene una propiedad con el objeto/objetos de las entidades relacionadas.



UNO-MUCHOS MUCHOS-UNO

```
@Entity()
export class Departamento {
  @PrimaryGeneratedColumn()
  id: number;

  @Column()
  nombre: string;

  @OneToMany(() => Empleado,
            empleado => empleado.departamento)
  empleados: Empleado[];
}
```

clase con la que se relaciona

propiedad de esa clase que contiene un objeto/objetos de esta entidad

```
@Entity()
export class Empleado {
  @PrimaryGeneratedColumn()
  id: number;

  @Column()
  nombre: string;

  @Column()
  puesto: string;

  @ManyToOne(() => Departamento,
             departamento => departamento.empleados, {
               eager: true, // opcional
             })
  departamento: Departamento;
}
```

ACTUALIZACIONES EN CASCADA

- Se utiliza para propagar una operación sobre una entidad a las entidades relacionadas

```
@Entity()  
export class Departamento {  
    @PrimaryGeneratedColumn()  
    id: number;  
  
    @Column()  
    nombre: string;  
  
    @OneToMany(() => Empleado,  
        empleado => empleado.departamento,{  
            cascade:["insert","remove"]  
    })  
    empleados: Empleado[];  
}
```

La inserción y eliminación de departamentos, provoca la inserción y eliminación de los objetos relacionados

USO DE LAS RELACIONES

➤ Obtener departamento al que pertenece un empleado:

```
async obtenerDepartamentoPorEmpleadoId(id: number): Promise<Departamento> {
    const empleado = await this.empleadoRepository.findOne({
        where: { id },
        relations: ['departamento'], // esto carga también el departamento
    });
    return empleado.departamento;
}
```

➤ Obtener empleados de un determinado departamento:

```
async obtenerEmpleadosPorDepartamento(departamentoId: number): Promise<Empleado[]> {
    return this.empleadoRepository.find({
        where: {
            departamento: {
                id: departamentoId,
            },
        },
        relations: ['departamento'], // opcional, carga también el Departamento
    });
}
```

QUERYBUILDER

- Ofrece mayor flexibilidad y eficiencia que los métodos `find()`:

```
async obtenerDepartamentoPorEmpleadoId(id: number): Promise<Departamento> {
    return await this.empleadoRepository.createQueryBuilder("departamento")
        .innerJoin("departamento.empleados","emp")
        // .innerJoinAndSelect("departamento.empleados","emp") //para obtener también campo de relación
        .where("emp.id=:id",{id:id})
        .getMany(); //getMany(), getOne(),..
}
```

- Obtener cuentas por cantidad mínima de extracción

```
async obtenerCuentasPorCantidad(cantidad: number): Promise<Cuenta[]> {
    return await this.cuentasRepository
        .createQueryBuilder("cuenta")
        .innerJoin("cuenta.movimientos","m")
        .where("m.cantidad>=:cant",{cant:cantidad})
        .andWhere("m.operacion>=:oper",{oper:"extracción"})
        .distinct(true)
        .getMany();}
```

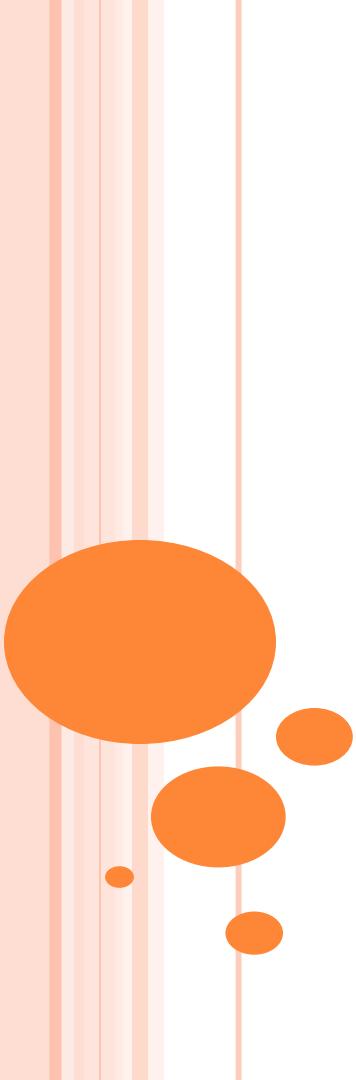
USAR QUERYS PERSONALIZADAS

- Inyectando objeto **DataSource** en lugar de **Repository**.
- Se pierde el mapeo **ORM**:

```
@Injectable()
export class LibroService {
    constructor(private readonly dataSource: DataSource) {}

    async buscarPorTitulo(titulo: string): Promise<any> {
        const resultados = await this.dataSource.query(
            'SELECT * FROM libro_model WHERE titulo = ?',
            [titulo]
        );
        return resultados;
    }
}
```





DOCUMENTAR SERVICIOS CON SWAGGER

INSTALACIÓN

- Se puede utilizar swagger para documentar servicios REST de nestjs.
- Primeramente, se debe instalar la librería:

```
>npm install --save @nestjs/swagger swagger-ui-express
```

CONFIGURACIÓN

➤ Se debe configurar swagger en el archivo main.ts:

```
const config = new DocumentBuilder()
    .setTitle('API de ejemplo')
    .setDescription('Documentación de ejemplo de Swagger')
    .setVersion('1.0')
    .addTag(libros')
    .build();

const document = SwaggerModule.createDocument(app, config);
SwaggerModule.setup('libros/api', app, document);
```

Establece la dirección
de acceso a la página
de ayuda

➤ Utilizando el decorador @ApiOperation se pueden
documentar los endpoints

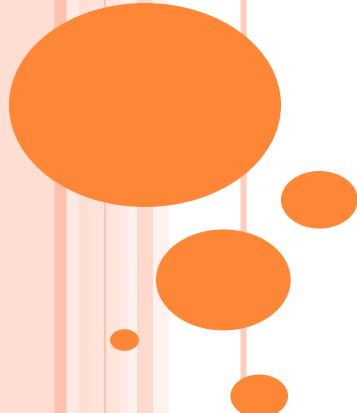
ACCESO

➤ Añadiendo a la dirección base del servidor la dirección indicada en la configuración se accede a la página de ayuda:

<http://localhost:3000/libros/api>

The screenshot shows the Swagger UI interface for a REST API. At the top, it displays the title "API de ejemplo 1.0 OAS 3.0" and the subtitle "Documentación de la API REST con Swagger". Below this, there are two main sections: "usuarios" and "Libros". The "Libros" section contains four API endpoints listed vertically: a green "POST" button for "/libros/alta", a blue "GET" button for "/libros/catalogo", a blue "GET" button for "/libros/buscar/{isbn}", and a red "DELETE" button for "/libros/eliminar/{id}". At the bottom of the interface, there is a "Schemas" section with a link to "LibroModel".

VALIDACIÓN DE DATOS EN DTOs



Fundamentos

- Se aplican a través de decoradores en los DTOs.
- Se requiere la instalación de class-validator:

>npm install class-validator class-transformer

- Y activarlo en el main.ts:

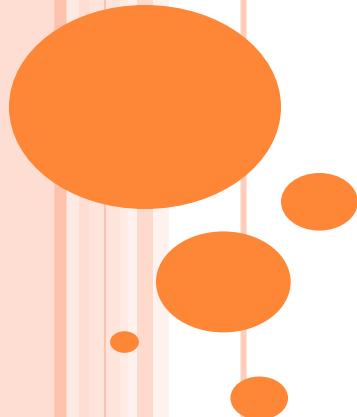
```
app.useGlobalPipes(  
  new ValidationPipe({  
    whitelist: false, //si es true, solo filtra campos validados  
    transform: true,  
    transformOptions: {  
      enableImplicitConversion: true,  
    },  
  }),  
);
```

Decoradores de validación

➤ Se colocan delante de los campos del DTO:

- `@IsString()`. Indica que el campo debe ser un string
- `@Length(min,max)`. En campos de tipo string indica la longitud mínima y máxima
- `@IsNumber()`. Tipo numérico
- `@ToInt()`. Tipo numérico entero
- `@IsEmail()`. El campo debe cumplir con las reglas de formato email
- `@Min(valor), @Max(valor)`. Para campos numéricos, indica el rango de valores entre los que debe estar
- `@IsNotEmpty()`. El campo no debe estar vacío

SECURIZACIÓN CON JWT



¿Qué es un token JWT?

- Mecanismo para comunicar de forma segura un microservicio con una aplicación cliente y proceder a su identificación.
- Se basa en el uso de una cadena JSON codificada y firmada, que incluye los datos del cliente (usuario, roles,...).
- El cliente envía el token JWT al microservicio en la cabecera de las peticiones, éste la decodifica y verifica que no ha sido alterada en tránsito, extrayendo a continuación la información del cliente para proceder a su autorización.



Estructura de un token JWT

► Un token JWT es una cadena codificada que consta de tres partes separadas por un ":":

header . payload . signature

```
header {  
    "alg": "HS256",  
    "typ": "JWT"  
}  
  
payload {  
    "iat": 1632173019,  
    "sub": "admin",  
    "authorities": [  
        "ROLE_ADMIN",  
        "ROLE_USER"  
    ],  
    "exp": 1632259419  
}  
  
signature HMACSHA256(  
    base64UrlEncode(header) + "." +  
    base64UrlEncode(payload),  
    secret_key  
)
```

header: Incluye dos campos, el tipo de token (siempre JWT) y el mecanismo de firma

payload: Contiene la información del cliente, como el nombre, roles, etc. También el tiempo de vida del token

signature. Se construye utilizando la función especificada en la cabecera y que recibe como parámetro la cabecera más el payload codificado, además de la clave secreta preestablecida. Esto permite verificar la integridad del mensaje

eyJhbGciOiJIUzUxMiJ9
.eyJpYXQiOjE2MzQxNDI0ODgsInN1YiI6ImFkbWluIiwiYXV0aG9yaXRpZXMiOlsiUk9MRV9BRE1JTlIsIlJPTEVfVVNFUiJdLCJleHAiOjE2MzQyMjg4ODh9.JsiPFDtOjkCW5AwwmloE3cz_WTl3ZM9CC4_ZKw7XbBH-zEedc0-geuYEc3BIWM3Yf7dtZQA3c5mmToRizIJk6g

Dependencias

- Para proteger recursos de servicios REST nest mediante JWT, se deben instalar las siguientes dependencias:

```
>npm install @nestjs/jwt @nestjs/passport passport passport-jwt  
>npm install -D @types/passport-jwt
```

Strategy

```
@Injectable()
export class JwtStrategy extends PassportStrategy(Strategy, "jwt") {
  constructor() {
    super({
      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
      secretOrKey: 'mysecret', // se debería usar variable de entorno
    });
  }

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

Guard: Configuración

- Implementa la autorización y autenticación para proteger el acceso a las rutas.
- Configuración de guard para JWT:

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

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

Guard: Implementación

➤ Para roles:

roles.guard.ts

```
@Injectable()
export class RolesGuard implements CanActivate {
  constructor(private reflector: Reflector) {}

  canActivate(context: ExecutionContext): boolean {
    const requiredRoles = this.reflector.getAllAndOverride<string[]>(ROLES_KEY, [
      context.getHandler(),
      context.getClass(),
    ]);

    if (!requiredRoles || requiredRoles.length === 0) {
      return true; // No se requiere rol específico
    }

    const { user } = context.switchToHttp().getRequest();
    return requiredRoles.includes(user.role);
  }
}
```

Decorador

➤ Define el decorador `@Roles` para aplicarlo en las rutas que queramos proteger:

`roles.decorator.ts`

```
import { SetMetadata } from '@nestjs/common';

export const ROLES_KEY = 'roles';
export const Roles = (...roles: string[]) => SetMetadata(ROLES_KEY, roles);
```

genera un decorador personalizado con uno o varios argumentos

asocia los metadatos indicados el método decorado

Servicio de usuarios

- Proporciona la fuente de usuarios.
- Pueden ser usuarios definidos en memoria, o venir de una base de datos:

users.service.ts

```
@Injectable()
export class UsersService {
  private readonly users = [
    { id: 1, username: 'admin', password: 'admin', role: 'admin' },
    { id: 2, username: 'usuario', password: 'usuario', role: 'user' },
  ];

  async findByUsername(username: string) {
    return this.users.find(user => user.username === username);
  }
}
```

Servicio de autenticación

- Proporciona el método para validar usuarios y generar el token:

auth.service.ts

```
@Injectable()
export class AuthService {
  constructor(private usersService: UsersService, private jwtService: JwtService ) {}
  async validateUser(username: string, password: string) {
    const user = await this.usersService.findByUsername(username);
    const valid:boolean=password.trim()==user.password;
    console.log(valid);
    if (user && valid) {
      const { password, ...result } = user;
      return result;
    }
    return null;
  }
  async login(user: any) {
    const payload = { username: user.username, sub: user.id, role: user.role };
    return {
      access_token: this.jwtService.sign(payload),
    };
  }
}
```

desestructuración:
extrae la propiedad
password y el resto en
un objeto result

Controlador de autenticación

- Ofrece el recurso para autenticar el usuario y que pueda obtener el token:

auth.controller.ts

```
@Controller('auths')
export class AuthController {
  constructor(private authService: AuthService) {}
  @Post('login')
  async login(@Body() { username, password }: {username: string; password: string }) {
    const user = await this.authService.validateUser(username, password);

    if (!user) throw new UnauthorizedException();
    return this.authService.login(user);
  }
}
```

Protección de recursos

```
export class LibrosController {  
    constructor(private readonly librosService: AppService) {}  
    @Roles('admin')  
    @Post("alta")  
    async create(@Body() libro:LibroModel,@Res() res:Response) {  
        const lib:LibroModel= await this.librosService.create(libro);  
        if(lib){  
            return res.status(200).json(lib);  
        }  
        return res.status(409).json(null);  
    }  
    @Roles('user', 'admin')  
    @Get("catalogo")  
    findAll() {  
        return this.librosService.findAll();  
    }  
    @Roles('admin')  
    @Delete('eliminar/:id')  
    remove(@Param('id') id: string) {  
        return this.librosService.remove(id);  
    }  
}
```

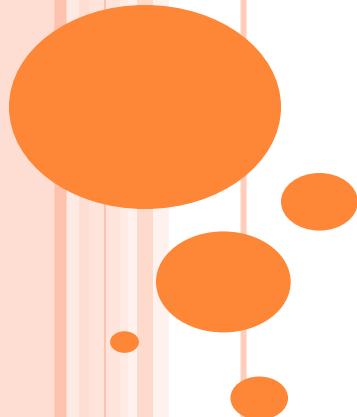


Archivo de módulo

```
@@Module({
  imports: [TypeOrmModule.forRoot({
    type: 'mysql',
    host: 'localhost',
    port: 3307,
    username: 'nestuser',
    password: 'nestpass',
    database: 'libros',
    entities: [LibroModel],
    synchronize: false,
  }),
  PassportModule,
  TypeOrmModule.forFeature([LibroModel]),
  JwtModule.register([
    {
      secret: 'mysecret',
      signOptions: { expiresIn: '1h' },
    },
  ]),
  controllers: [LibrosController, AuthController],
  providers: [AppService, UsersService, AuthService, JwtStrategy],
})
export class AppModule {}
```



COOKIES

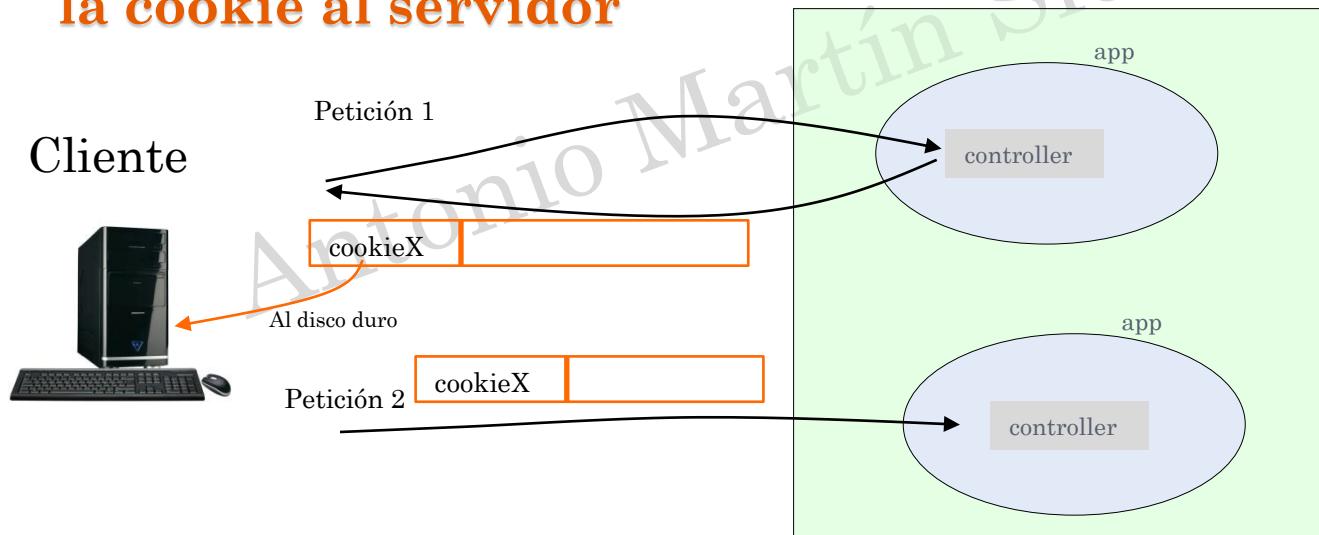


Características

- Datos creados por el backend enviados en la cabecera de la respuesta al cliente y que éste almacena en el disco duro
- Solo admite cadenas de caracteres
- El tiempo de vida es limitado
- Requiere la instalación de cookie parser:
 >npm install cookie-parser

Funcionamiento

- La cookie es creada por el servidor y enviada al cliente en la cabecera de la respuesta.
- En posteriores conexiones, el cliente envía de nuevo la cookie al servidor



Creación y envío de una cookie

➤ A través del objeto Response se crea una cookie en el controler y se envía en la cabecera de la respuesta:

```
@Controller('auth')
export class AuthController {
  @Post('login')
  login(@Res() res: Response) {
    // ejemplo de dato a enviar en la cookie
    const token = 'xyzA349';
    // Crear la cookie
    res.cookie('auth_token', token, {
      httpOnly: true, // la cookie no es accesible desde el navegador (más seguro)
      secure: false, // true si se usa HTTPS
      sameSite: 'lax', // o 'strict' o 'none' (si se usa cross-site)
      maxAge: 24 * 60 * 60 * 1000, // Tiempo de vida
    });
    //Envío de la cookie
    res.send();
  }
}
```

Recuperación de una cookie

- En el mismo u otro controler se recupera la cookie enviada por el cliente en la petición:

```
@Controller('user')
export class UserController {
  @Get('profile')
  profile(@Req() req: Request) {
    const token = req.cookies['auth_token'];
    //la cookies es analizada
    //y a partir de ella obtener información del usuario
  }
}
```

Configuración

- Para poder hacer uso de cookies en una aplicación nestjs, es necesario hacer uso de cookieParser en el archivo main.ts:

```
import * as cookieParser from 'cookie-parser';
async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.use(cookieParser());
  :
}
bootstrap();
```

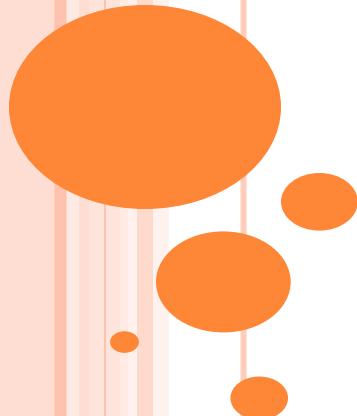
Cookies en el front

- Si una cookie está configurada como **HttpOnly** no es accesible directamente desde el front.
- Se puede configurar Angular para que la cookie sea redirigida de nuevo al realizar otra petición:

```
this.httpClient.post('https://api.midominio.com/auth/refresh', {}, {  
  withCredentials: true  
}).subscribe(response => {  
  console.log(response);  
});
```

Las cookies HttpOnly recibidas
desde esa dirección son reenviadas
de nuevo

ACCESO A MONGODB



Bases de datos MongoDB

- Se trata de una base de datos documental, donde los datos se almacenan como documentos JSON dentro de colecciones
- Se puede crear una instancia en un contenedor docker de la imagen mongo

```
>docker run --name=mongo -p 27017:27017
```

- Se puede utilizar mongo compass para gestionar las bases de datos de forma gráfica



Acceso a MongoDB en Nestjs

- En cada proyecto se debe instalar la librería mongoose:

```
>npm install @nestjs/mongoose mongoose
```

- Los modelos de datos o entidades son clases normales anotadas con `@Schema` y `@Prop`
- Se trabaja con documentos, que son la combinación de entidad y Document



Ejemplo de entidad

Documento: Entidad+Document

```
export type CursoDocument = Curso & Document;
```

```
@Schema()  
export class Curso {  
    @Prop({ required: true })  
    nombre: string;  
    @Prop()  
    duracion: number;  
    @Prop()  
    precio: number;  
}
```

```
export const CursoSchema = SchemaFactory.createForClass(Curso);
```

Esquema de la entidad

Utilización en service

- Se inyecta un objeto Model a partir del documento:

```
constructor(  
    @InjectModel(Curso.name) private cursoModel: Model<CursoDocument>  
) {}
```

- El Model proporciona métodos para operar contra la base de datos: `save()`, `find()`, `findOne()`, etc

```
crear(curso:Curso) {  
    const nuevo = new this.cursoModel(curso);  
    return nuevo.save();  
}  
async findAll() {  
    const cursos:Curso[] = await this.cursoModel.find().exec();  
    return cursos;  
}
```

creación de model a
partir de la entidad

Configuración

➤ En AppModule se configura la librería y los datos de conexión a la base de datos:

```
@Module({
  imports: [MongooseModule.forFeature([
    { name: Curso.name,
      schema: CursoSchema }]),
    MongooseModule.forRoot('mongodb://localhost:27017/academia')],
  controllers: [CursosController],
  providers: [CursosService],
})
export class AppModule {}
```