NestJS

Nest CLI

Generar una nueva aplicación de Nest

```
nest new aplication-sample
```

Generar archivos de distribución

```
nest build
```

Ejecutar la aplicación (-w for watch changes)

```
nest start
```

Generar elemento de Nest

```
nest generate ELEMENT
```

Elementos que se pueden generar:

- *Controller (co)*
- *Service (s)*
- *Module (mo)*
- *Middleware (m)*
- *Guard (gu)*
- *Interceptor (in)*
- *Pipe (pi)*
- *Decorator (d)*
- *Filter (f)*
- *Resolver (r)*
- *Gateway (g)*
- *Schema (sc)*
- *Resource (res)*

Estructura de Carpetas

(Recomendada)

```
src/
|
├── users/
|   ├── dto/
|   |   └── create-user.dto.ts
|   ├── entities/
|   |   └── user.entity.ts
|   ├── users.controller.ts
|   ├── users.service.ts
|   └── users.module.ts
|
├── common o shared/
|   ├── guards/
|   ├── pipes/
|   ├── filters/
|   └── decorators/
|
├── config
|   ├── config.module.ts
|   └── config.service.ts
|
├── app.module.ts
└── main.ts
```

Módulos

Un módulo es una agrupación de una funcionalidad de la aplicación, para separar toda la lógica en partes más pequeñas, mantenibles y autónomas.

```
@Module({
  imports: [],
  controllers: [UserController],
  providers: [UserService],
  exports: [],
})
export class UserModule {}
```

- ***Imports***: Especifica un arreglo de dependencias (otros modulos) que todos los miembros del módulo conocerán.
- ***Controllers***: Controladores que pertenecen al módulo.
- ***Providers***: Providers y servicios que usaran los miembros del módulo (controllers)
- ***Exports***: En debido caso que otro modulo necesite usar algún miembro del módulo local.

Controllers

Un controlador se encargará de administrar las rutas del módulo perteneciente y la conexión a la lógica de negocios / procesos (servicios).

Se encarga, mediante los métodos declarados, de procesar las solicitudes HTTP.

```
@Controller('user')
export class UserController {
    // Endpoints - methods / user
}
```

Endpoints

Los endpoints de la API que manejara el controlador, serán dados por los métodos de la clase que tenga, cada método se identificara con su respectivo decorador. Cada método retornará un objeto JSON que será la respuesta final de la petición.

```
@Controller('user')
export class UserController {
    @Get()
    public getAllUsers(){}

    @Get(':id')
    public getUserById(){}

    @Post()
    public postUser(){}

    @Put(':id')
    public putUser(){}

    @Delete(':id')
    public deleteUser(){}
}
```

Services

Un servicio es un tipo de provider que nos servirá como un Singleton (una única instancia en la aplicación) para poder compartir datos entre varios miembros o generalmente, usados para realizar la lógica de la aplicación y la conexión a la base de datos.

NestJS

```
@Injectable()
export class UserService {}
```

El decorador @Injectable, generalmente declara un provider.

Inyección de dependencias.

Para poder usar un servicio en un controlador, al ser un objeto Singleton, debe inyectarse en el constructor del controlador donde se usará, además de asegurarse que pertenecen al mismo modulo o se está importando correctamente de un módulo externo.

```
@Controller('user')
export class UserController {
    constructor(
        private readonly userService: UserService
    ){}
}
```

Exception Filters

Para manejar los códigos de respuesta al momento de contestar a una petición, necesitaremos lanzar un error que Nest nos provee.

Al lanzar un error de estos, NestJS detiene inmediatamente la ejecución y retorna automáticamente una respuesta al cliente con el status del error.

```
@Get()
public getAllUsers(){
   throw new NotFoundException('Users not found')
}
```

Tipos de errores que Nest provee:

- *BadRequestException*
- *NotFoundException*
- *RequestTimeoutException*
- *PayloadTooLargeException*
- *ForbiddenException*
- *UnauthorizedException*
- *InternalServerErrorException*
- *GoneException*

Endpoints Data

Recibir parámetro de ruta - @Param

https:localhost/user/2

```
@Get(':id')
public getUser( @Param('id') id: number ) {
    return this.user[id]
}
```

Recibir Query String - @Query

https:localhost/user?verbose=true

```
@Get()
existUser( @Query('verbose') id: boolean ) {
return [] }
```

Recibir cuerpo - @Body

```
@Post()
addUser( @Body() body: User ) {
    return []
}
```

Pipes

Los pipes son capaces de transformar los datos recibidos de la petición antes de tenerlos para asegurar su tipado, valor o que sea instancia de un objeto. Por defecto, si no se usa un Pipe para transformar será string.

Tipos de pipes integrados:

- *ValidationPipe*
- *ParseBoolPipe*
- *ParseIntPipe*
- *ParseFloatPipe*
- *ParseArrayPipe*
- *ParseUUIDPipe*

Ejemplo de uso (ParseIntPipe):

```
@Get(':id')
public addUser(
   @Param('id', ParseIntPipe) id: number
) {
   return this.user[id]
}
```

Si un pipe falla, lanzara un error 400 por defecto (error de petición)

Data Transfer Object (DTOs)

```
npm install class-validator class-transformer
```

Un DTO es un objeto tipado que define como vas a recibir la información de la petición y como la vas a responder.

También es útil para validar como será el JSON que vas a recibir de la petición, para ello Nest trabaja con estas dependencias para validarlo.

Crear una clase para un DTO

```
export class CreateUserDto {
    @IsString()
    readonly name: string;

    @IsEmail()
    readonly email: string;
}
```

Usar el DTO para validar un Body con ValidationPipe

```
@Post()
@UsePipes( ValidationPipe )
public addUser(
    @Body() user: CreateUserDto
) { // logic to add user }
```

NestJS

Uso de Pipes Globales

El uso, por ejemplo, de Validation Pipe, por lo general lo queremos en todos los endpoints de tipo Post, Put, para no estar necesitando del decorador en todos los métodos o controladores que requiramos, podemos usarlo de forma global.

Main.ts (useGlobalPipes)

```
async function bootstrap() {
  const app =await NestFactory.create(AppModule);
  app.useGlobalPipes( new ValidationPipe({
    whitelist: true,
    forbidNonWhitelisted: true,
    transform: true,
    transformOptions: { enableImplicitConversion:
true }}) )
  await app.listen(process.env.PORT ?? 3000);
}
bootstrap();
```

Generar recursos completos

```
nest generate resource name
```

Nest nos apoyara para la creación de controladores, servicios, módulos, DTOs, entities para base de datos con un solo comando al mismo tiempo. Capaz de generar Rest API, Websockets, GraphQL Cruds.

```
✓ What transport layer do you use? REST API
✓ Would you like to generate CRUD entry points? Yes
CREATE src/group/group.controller.ts (938 bytes)
CREATE src/group/group.controller.spec.ts (586 bytes)
CREATE src/group/group.module.ts (257 bytes)
CREATE src/group/group.service.ts (647 bytes)
CREATE src/group/group.service.spec.ts (471 bytes)
CREATE src/group/dto/create-group.dto.ts (32 bytes)
CREATE src/group/dto/update-group.dto.ts (177 bytes)
CREATE src/group/entities/group.entity.ts (23 bytes)
UPDATE package.json (2215 bytes)
UPDATE src/app.module.ts (445 bytes)
✓ Packages installed successfully.
```

DTO Update

```
export class UpdateGroupDto extends
PartialType(CreateGroupDto) {}
```

El DTO de update generado por Nest, usa PartialType, lo que hace que los valores del DTO de creación se extiendan, pero siendo opcionales todos los atributos.

Servir Estaticos

```
npm install @nestjs/serve-static
```

Para servir contenido estatico se añade esta configuración al appModule.

```
@Module({
  imports: [
    ServeStaticModule.forRoot({
      rootPath: join(__dirname,'..','public')
    })
  ],
  controllers: [ AppController ]
)
```

Global Prefix

Colocar un prefijo como ‘api/route’ puedes hacerlo en el main.ts

```
app.setGlobalPrefix('api')
```

Custom Pipes

Para crear un pipe nuestro, personalizado por nosotros para hacer validaciones o cambios a los datos recibitos, el dato retornado, es el nuevo que alterara en el controller.

```
nest generate pipe name
```

```
@Injectable()
export class MongoIdPipe implements PipeTransform {
  transform(value: string, metadata: ArgumentMetadata)
{
    if( value.length == 0 ){
      throw new BadRequestException('mal')
    }
    return value.toLowerCase();
  }
}
```

Uso en controlador:

```
@Get(':id')
  findAll( @Param( 'id', MongoIdPipe ) id : string ) {
  }
```

MongoDB Mongoose

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

Configuración de URI de conexión en AppModule

```
@Module({
  imports: [
MongooseModule.forRoot('mongodb:localhost:27017/dbnm'),
  ],
  controllers: [ AppController ],
  providers: [ AppService ],
})
export class AppModule {}
```

Creación de un schema (entity) que será nuestra table en la base de datos

```
import { Prop, Schema, SchemaFactory } from
"@nestjs/mongoose";
import { Document } from "mongoose";
@Schema()
export class Pokemon extends Document{

    @Prop({
        unique: true,
        index:true
    })
    nombre: string;
   @Prop({
```

NestJS

```
        unique: true,
        index:true
    })
    poder: number;
}
export const PokemonSchema =
SchemaFactory.createForClass( Pokemon );
```

Registrar el Schema en el submodulo de Nest perteneciente a esa tabla

```
@Module({
  imports: [
    MongooseModule.forFeature( [
      {
        name: Pokemon.name,
        schema: PokemonSchema
      }
    ] )
  ],
  controllers: [PokemonController],
  providers: [PokemonService],
})
export class PokemonModule {}
```

Usar y alterar el modelo

Para usar y hacer inserciones del modelo creado en mongoose, se tiene que hacer en un servicio (recomendad) donde haremos la inyección del modelo (que haremos Injectable) con el nombre del modelo y su tipo

En el servicio:

```
import { Model } from 'mongoose';
import { Pokemon } from './entities/pokemon.entity';
import { InjectModel } from '@nestjs/mongoose';
@Injectable()
export class PokemonService {
  constructor(
    @InjectModel( Pokemon.name )
    private readonly pokemonModel: Model<Pokemon>
  ){}
```

CRUD Mongoose

Create

```
async create(createPokemonDto: CreatePokemonDto) {
   this.pokemonModel.create( createPokemonDto )
}
```

FindById

```
async findById(id: number) {
    await this.pokemonModel.findById(1);
}
```

FindAll

```
async findAll(id: number) {
   await this.pokemonModel.find();
}
```

Update

```
async update
(id: string, updatePokemonDto: Partial<CreatePokemonDto>) {
  await this.pokemonModel.findByIdAndUpdate(id,
updatePokemonDto,
  {  new: true, // ← devuelve el documento actualizado
     runValidators: true, // ← valida según el schema
  });
  return updated;
}
```

Delete

```
async remove(id: string) {
    await this.pokemonModel.findByIdAndDelete(id);
}
async remove(id: string) {
    await this.pokemonModel.deleteOne({_id: id});
}
```

Patrón Adapter

Un patrón adaptador nos permite centralizar las dependencias externas de nuestro proyecto, para en debido caso que necesiten cambiarse, se pueda cambiar en un solo archivo y no en todos donde se usa.

Centralizar, por ejemplo, la dependencia de axios en un servicio, y que demás servicios a la hora de realizar una petición, usen nuestro adaptador, nos dará la tolerancia fácil a cambios, por ejemplo, dejar de usar axios y usar fetch, será un cambio fácil de implementar en la aplicación. Centralizamos la dependencia en un provider y ese provider lo usamos en demás app.

```
export interface IHttpAdapter{
    get<T>( url:string ): Promise<T>
}
@Injectable()
export class HttpAdapter implements IHttpAdapter {
    private axios: AxiosInstance = axios
    async get<T>(url: string): Promise<T> {
        try{
            const res = await this.axios.get<T>(url);
            return res.data;
        }catch(e){
            throw new Error(e)
        }
    }
}
```

Si usamos este provider en demás servicios para realizar peticiones, solo necesitaremos alterar este provider.

NestJS

Environments .ENV

```
npm install @nestjs/config
```

Crea el archivo env en el root del proyecto con las variables de entorno necesarias.

```
MONGO_DB=mongodb://localhost:27017/nest-pokemon
PORT=3000
```

En app.module.ts

```
@Module({
  imports: [ ConfigModule.forRoot({ isGlobal:true,
      load : [ ()=>{
          MONGO_DB: process.env.MONGO_DB || 'holi',
          PORT: parseInt(process.env.PORT) || 3000,
          defaultLimit:
parseInt(process.env.DEFAULT_LIMIT) || 10,
        }) ]
    })
 ]
})
export class AppModule {}
```

Accede con process.env.VARIABLE en archivos que no son building block de nest (main.ts por ejemplo)

Si se requiere usarse en un servicio, controller u otro (building block) las variables de entorno se usarán inyectando este servicio donde el get es definió en el root de load de ConfigModule.

```
@Injectable()
export class AppService {
  constructor(
    private configService: ConfigService
  ){}
  getVariable(){
  this.configService.getOrThrow<number>('defaultLimit')
  }
}
```

CORS

```
async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.enableCors({
    origin: ['http://localhost:4200'],
    methods: 'GET,HEAD,PUT,PATCH,POST,DELETE',
    credentials: true, // Cookies o auth
  });
  await app.listen(process.env.PORT ?? 3000);
}
```

TypeORM

```
npm install --save @nestjs/typeorm typeorm pg
```

```
npm install --save @nestjs/typeorm typeorm mysql2
```

Importar el módulo de conexión en el app.module.ts

```
@Module({
  imports: [
    ConfigModule.forRoot(),
    TypeOrmModule.forRoot({
      type: 'postgres',
      host: 'localhost',
      database: process.env.DB,
      username: process.env.DB_USER,
      password: process.env.DB_PASS,
      port: 5432,
      autoLoadEntities: true,
      synchronize: true
    })
  ],
})
export class AppModule {}
```

(Process.env como ya se explicó, los valores se están tomando del archivo .env)

Entity ORM

Para crear una entidad o tabla simple (sin relacion) en una base de datos simple con TypeORM, primero definiremos nuestro archivo de entidad en un modulo.

Product.entity.ts

```
import { Column, Entity, PrimaryGeneratedColumn } from
"typeorm";
@Entity()
export class Product {
    @PrimaryGeneratedColumn('increment')
    id: number;

    @Column('text',{ unique: true })
    title:string

    @Column('numeric',{ default: 0, nullable: false })
    stock:string

    @Column('text',{ array:true })
    sizes:string
}
```

Usar la entidad o servicio en otros modulos para hacer un CRUD en otro modulo (siempre hay que exportar el TypeORM)

```
import { TypeOrmModule } from '@nestjs/typeorm';
import { Product } from './entities/product.entity';
@Module({
  imports: [
    TypeOrmModule.forFeature([Product])
  ],
  exports: [TypeOrmModule, Service)]})
export class ProductsModule {}
```

NestJS

Entity Repository

Nest con TypeORM nos provee de un repositorio automático para no crear la clase nosotros (como lo hacía .NET), para inyectarlo y usarlo como conexión esperara la entidad que usara dicho repositorio.

Inyectar en el servicio de la entidad

```
constructor(
    @InjectRepository( Product )
    private readonly productRepository:
Repository<Product>
  ){}
```

CRUD a entidad en Postgres

Create

```
async create(createProductDto: CreateProductDto) {
  try{
    const product = await
this.productRepository.create( createProductDto )
    await this.productRepository.save( product )
    return product;
  }catch( e ){
    throw new InternalServerErrorException( e )
  }
}
```

GetAll

```
async getAll(){
    try{
      return await this.productRepository.find();
    }catch( e ){
      throw new InternalServerErrorException( e )
    }
  }
```

GetById

```
async getById( id :number ){
    try{
      return await this.productRepository.findOne( {
where: { id } } );
    }catch( e ){
      throw new InternalServerErrorException( e )
    }
  }
```

Update

```
async update(id: number, updateProductDto:
UpdateProductDto) {
    try {
      const product = await
this.productRepository.findOne({ where: { id } });
      const updated = Object.assign(product,
updateProductDto);
      await this.productRepository.save(updated);
      return updated;
    } catch (e) {
      throw new InternalServerErrorException(e);
    }
}
```

Delete

```
async delete(id: number) {
    try {
      const product = await
this.productRepository.findOne({ where: { id } });
      await this.productRepository.remove(product!);
      return { message: 'Producto eliminado
correctamente' };
    } catch (e) {
      throw new InternalServerErrorException(e);
    }
  }
```

Entity Listeners

Son decoradores que se colocan directamente en las entidades. Sirven para ejecutar código automáticamente **antes o después** de ciertas operaciones como insertar, actualizar o eliminar.

Ejemplo: (todo título va en minúsculas)

```
@Entity()
export class Product {
    @Column('text',{ unique: true })
    title:string

    @BeforeInsert()
    berofeInsert(){
        this.title = this.title.toLowerCase()
    }
}
```

Eventos disponibles en entity's:

- `@BeforeInsert`
- `@AfterInsert`
- `@BeforeUpdate`
- `@AfterUpdate`
- `@BeforeRemove`
- `@AfterRemove`
- `@AfterLoad`

QueryBuilder

Para poder usar WHERE como la usamos con SQL

```
async getByIdOrName(name: string, id: string) {
    const qb = await
this.productRepository.createQueryBuilder();
    const product = await qb
      .where(`name=:name or id=:id`, { name, id
})
      .getOne();
    return product;
  }
```

NestJS

Relaciones de Entidades

Hay una forma bien rara de hacer un 1:N, con arreglos, pero ni la voy a explicar por que ni es buena ni es mantenible, vamos al grano.

Un Usuario puede hacer muchos Post

User.entity.ts

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

  @Column('text')
  name: string;

  // Relación: Un usuario tiene muchos posts
  @OneToMany(() => Post, post => post.user)
  posts: Post[];
}
```

Importante declarar la entity en el módulo User con TypeORM.forfeature([User])

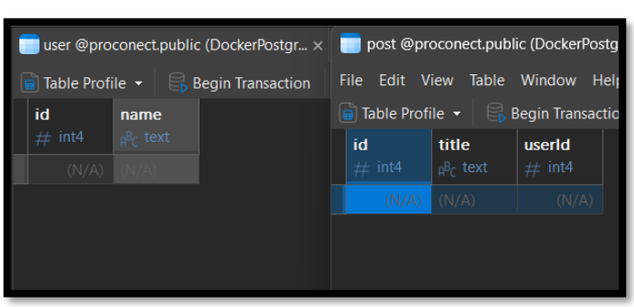
Post.entity.ts

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

  @Column('text')
  title: string;

  // Relación: Muchos posts pertenecen a un usuario
  @ManyToOne(() => User, user => user.posts, { onDelete: 'CASCADE' })
  user: User;
}
```

Al tener la conexión y las entities registradas en sus modulos correspondientes de esta forma, en nuestra base de datos ya debería de estar la relación.



Crear a Post

El usuario lo vamos a poder crear sin problema por que no depende de nadie, el problema es Post que necesita el id del usuario (o en realidad la referencia del objeto)

Forma segura validando que el usuario existe en BD.

```
async create(createPostDto: CreatePostDto) {
  const user = await this.userRepository.findOne({ where: { id: userId } });
  if (!user) throw new NotFoundException('User no encontrado');
  const post = this.postRepository.create({
    ...createPostDto,
    user: user,
  });
  return this.postRepository.save(post);
}
```

Forma en que confias que el ID que te están enviando siempre es legal (no verificas si el usuario existe, solo metes el id)

```
async create(createPostDto: CreatePostDto) {
  const post = this.postRepository.create({
    ...createPostDto,
    user: { id: createPostDto.userId },
  });
  return this.postRepository.save(post);
}
```

Consultas Join

User a su Post

```
const user = await this.userRepository.find({
  relations: ['posts'],
});
```

```
  "id": 1,
  "name": "Luis",
  "posts": [
    {
      "id": 101,
      "title": "Primer post"
    },
    {
      "id": 102,
      "title": "Segundo post"
    }
  ]
```

Post a su User

```
const posts = await this.postRepository.find({
    relations: ['user'],
});
```

```
{
  "id": 101,
  "title": "Primer post",
  "user": {
    "id": 1,
    "name": "Luis"
  }
},
```

Con queryBuilder

```
const post = await this.postRepository
    .createQueryBuilder('post')
    .leftJoinAndSelect('post.user', 'user') // relación definida en la entidad
    .where('post.id = :id', { id: 101 })
    .getOne(); // O getMany para todos los post
```

NestJS

QueryRunner

Es una herramienta que te da TypeORM para:

- Correr SQL a bajo nivel.
- Manejar transacciones manuales.
- Ejecutar múltiples queries en una misma conexión.
- Brincarte los repositorios del ORM.

Inyección del datasource de TypeORM para poder usar queryRunner en el servicio:

```
constructor(
  private readonly dataSource: DataSource
){}
```

Usar QueryRunner con transacciones:

```
const queryRunner =
this.dataSource.createQueryRunner();
await queryRunner.connect(); //conect db
await queryRunner.startTransaction(); //opcional
await queryRunner.query(
  'INSERT INTO post(title, content, "userId") VALUES
($1, $2, $3)',
  ['Post brutal', 'Hecho a mano', 1]
);
await queryRunner.commitTransaction(); // Confirmar
await queryRunner.rollbackTransaction(); // Deshacer
await queryRunner.release(); //liberar coneccion
```

Archivos con Formdata

```
npm i @types/multer
```

Si bien se pueden subir archivos a un backend con un base64 encode, no es recomendado en absoluto, lo mejor es subir directamente el archivo encodificado en un formData en vez de un form/json

Endpoint (el parámetro de fileInterceptor es el key de la peticion)

```
@Post('product')
  @UseInterceptors( FileInterceptor('file', {
    storage: diskStorage({
      destination: './public',
      filename: fileNamer
    }),
  }) )
  uploadImage(
    @UploadedFile() file: Express.Multer.File
   ) { return file; }
```

Objeto file que fluye

```
"fieldname": "file",
"originalname": "perro-Border_Collie-02.jpg",
"encoding": "7bit",
"mimetype": "image/jpeg",
"buffer": {
```

Callback para fileNamer (renombrar archiv) o para fileFilter(validat archivos)

```
const fileFilter = (req: Express.Request, file:
Express.Multer.File, cb: Function) : void =>{}
```

Ejemplo:

```
const fileNamer = (req: Express.Request, file:
Express.Multer.File, cb: Function) => {
  if (!file.originalname) {
    return cb(new BadRequestException('El archivo no puede
estar vacío'), false);
  }
  const extension = file.mimetype.split("/")[1]
  let fileExt = file.originalname;
  fileExt = "uuid"+extension;
  const uniqueName = `${Date.now()}-${Math.round(Math.random()
* 1e6)}${fileExt}`;
  cb(null, uniqueName);
}
```

Views ORM

Para poder ver una vista que hayas creado en base de datos tiene que ser sincronizada en falsa y usar ViewEntity, además de que todas las columnas serán de solo lectura, posteriormente, se usa como un repositorio de cualquier entidad.

```
import { ViewEntity, ViewColumn } from 'typeorm';

@ViewEntity({ name: 'mi_vista', synchronize:
false })
export class MiVista {
  @ViewColumn()
  id: number;
}
```

O bien crearla en TypeOrm

```
import { ViewColumn, ViewEntity } from "typeorm";

@ViewEntity({
  name: 'mi_vista',
  expression: (connection) =>
    connection
      .createQueryBuilder()
      .select('usuario.id', 'id')
      .addSelect('usuario.nombre', 'nombre')
      .from('usuario', 'usuario')
})
export class MiVista {
  @ViewColumn()
  id: number;

  @ViewColumn()
  nombre: string;
}
```

NestJS

CronJobs

```
npm i @nestjs/schedule
```

Un cronjob no es más que una función o tarea que se ejecutara en rutina, cada cierto tiempo o programarla para un día especifico o días con horas específicas.

Usos:

- Hacer respaldos automáticos
- Enviar correos diarios
- Limpiar registros viejos
- Actualizar estados en la BD

En el módulo se importa la dependencia Schedule y los servicios con las tareas Cron.

```
@Module({
  imports: [ ScheduleModule.forRoot() ],
  controllers: [TareaController],
  providers: [TareaService],
})
export class TareaModule {}
```

En el servicio:

```
@Injectable()
export class TareaService {
  @Cron(CronExpression.EVERY_10_SECONDS)
  handleCron() {
    console.log('Se ejecuta cada 10 segundos');
  }
  @Cron('0 8 * * *')
  tareaDiaria() {
    console.log('Todos los días a las 8:00 AM');
  }
}
```

Logger de NestJS

Uso de ejemplo del logger dado por defecto de Nestjs.

```
@Injectable()
export class LogeruseService {
  private readonly logger = new
Logger(LogeruseService.name); //Contexto


  hacerAlgo() {
    this.logger.log('Esto es un log normal');
    this.logger.warn('Esto es una advertencia');
    this.logger.error('Esto es un error rojo);
    this.logger.debug('Esto es debug (si lo activas)');
    this.logger.verbose('Más detalle');
  }
}
```

Authentication (JWT / Passport)

```
npm i @nestjs/passport passport

npm i @nestjs/jwt passport-jwt

npm i -D @types/passport-jwt
```

En el modulo de autenticación usando variables de entorno con ConfigService

```
@Module({
  imports: [
   PassportModule.register({ defaultStrategy: 'jwt' }),
    JwtModule.registerAsync( {
      imports: [ ConfigModule ],
      inject: [ ConfigService ],
      useFactory: (conf: ConfigService) => ({
        secret: conf.getOrThrow('JWT_SECRET'),
        signOptions: {
          expiresIn: '2h'
        }
      })
    } )
  ],
  controllers: [AuthController],
  providers: [AuthService],
})
export class AuthModule {}
```

Sin ser asíncrono:

```
JwtModule.register({
  secret: process.env.JWT_SECRET,
  signOptions: {
    expiresIn: '2h'
  }
})
```

JWT-Strategy

Creacion de la interfaz de playload, que será la información dentro que tendrá nuestro token

```
export interface JwtInterface{
    id: string;
    rol: string
}
```

Creacion del provider strategy de JWT para realizar las validaciones de JWT

```
@Injectable()
export class JwtStragety extends PassportStrategy(Strategy) {
  constructor(
    @InjectRepository(User)
    private readonly userRespository: Repository<User>,


    configService: ConfigService,
  ) {
    super({
      secretOrKey: configService.get('JWT_SECRET')!,
```

NestJS

```
      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
    });
  }
  async validate(payload: JwtInterface): Promise<User> {
    const { id } = payload;
    const user = await this.userRespository.findOneBy({ id });
    if (!user) throw new UnauthorizedException('Token
invalido');
    if (!user.isActive)
      throw new UnauthorizedException('Usuario deshabilitado
temporalmente');
    return user;
  }
}
```

(Lo que retorne validate se añade a la request (request.user))

Generar token

```
@Injectable()
export class AuthService {
    constructor(
        private readonly jwtService: JwtService
    ){}
    private generateToken( jwtPayload: JwtInterface ){
        return this.jwtService.sign(jwtPayload)
    }
}
```

Proteger una ruta (pedir token en la ruta)

```
@Get('private')
@UseGuards( AuthGuard() )
testingPrivateRoute(){  }
```

Guards

Un Guard es un provider que implementa la interfaz CanActivate (también se declaran como providers en el módulo al que pertenezcan).
Se ejecuta antes de que un controlador o handler corra, y decide si el request sigue o se detiene dependiendo el booleano que retorne.

Es perfecto para:

- Proteger rutas con JWT
- Validar roles, permisos

```
@Injectable()
export class SampleGuard implements CanActivate {
  canActivate(context: ExecutionContext): boolean |
Promise<boolean> {
    console.log('Pasar');
    return true;
  }
}
```

Uso:

```
@Get('private')
  @UseGuards( SampleGuard )
  testingPrivateRoute(){
}
```

Ejemplo del RoleGuard (reflector nos permite acceder para obtener los metadatos puestos anteriormente):

```
@Injectable()
export class RoleGuard implements CanActivate {
  constructor(
    private readonly reflector: Reflector
  ){}
  canActivate( context: ExecutionContext ): boolean{
    const type = this.reflector.get('UserType',
context.getHandler())
    if(!type) return true
    const user =
context.switchToHttp().getRequest().user as User
    if(user.userType == type){
      return true
    }else{
      throw new ForbiddenException(`The user ${
user.username } is not a ${ type == 'c' ? 'client':
'supplier' }`)
    }
  }
}
```

Custom Decorators

Un decorador personalizado (custom decorator) es básicamente una función que tú creas para reutilizar lógica, sobre todo en tus controladores o guards.

Es un proceso y una logica repetible para sacar un valor y asignarselo por defecto a la variable que decoremos.

Ejemplo para sacar del jwt-strategy, el usuario de la request y colocarlo en valor de parámetro

```
export const User = createParamDecorator(
  (data, ctx: ExecutionContext) => {
    const request = ctx.switchToHttp().getRequest();
    const user = request.user;
    return user;
  },
);
```

Donde:

- La data son los argumentos que recibes al llamar el decorador.
- El valor retornado se asigna de valor a la variable que decora
- Donde el contexto tiene la información de la petición (request o response)

Uso:

```
@UseGuards( AuthGuard() )
@Get('private')
testingPrivate( @GetUser('data_value') user: User ){
  return user
}
```

NestJS

Composition Decorator

El composition decorator nos permite agrupar decoradores en un solo decorador.

Como en los ejemplos anteriores, necesitamos saber si un usuario tiene un token valido (AuthGuard) y también saber si tiene acceso a un recurso (RoleGuard hecho por nosotros y colocar SetMetadata para decir que rol esperamos que pueda acceder solamente):

Ejemplo (aqui usamos 2 guards para autenticar y autorizar):

```
@SetMetadata('UserType', UserType.PROVEEDOR)
@UseGuards( AuthGuard(), RoleGuard )
@Get('private')
testingPrivateRoute( @GetUser() user: User ){
  return user
}
```

¿Cómo agrupamos nuestros decoradores de Metadata y Auth/Role Guard?

```
export function Auth( typeRole: UserType ) {
  return applyDecorators(
    SetMetadata('UserType', typeRole),
    UseGuards(AuthGuard(), RoleGuard)
  );
}
```

ApplyDecorators aplica los decoradores que se ejecuten en el, como parámetro recibimos el tipo de usuario y lo aplicamos al setMetadata y posteriormente aplicamos los Guards.

```
@Auth( UserType.CLIENTE )
@Get('private')
testingPrivateRoute( @GetUser() user: User ){
  return user
}
```

Swagger

```
npm i --save @nestjs/swagger
```

En main.ts

```
const config = new DocumentBuilder()
    .setTitle('ProConect API')
    .setDescription('Application for connecting')
    .setVersion('1.0')
    .build();
  const documentFactory = () =>
SwaggerModule.createDocument(app, config);
  SwaggerModule.setup('swagger', app, documentFactory);
```

Y de huevaaaa, pero todo se documenta con decoradores, endpoints, controllers y DTOs

Controllers y endpoints

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

  @Post('login')
  @ApiResponse({ status: 200, type: LoginResponse })
  public async loginUser( @Body() loginDTO:
LoginDTO  ){
    return await this.authService.loginRequest(
loginDTO );
  }
}
```

DTOs

```
export class LoginDTO {
    @ApiProperty()
    @IsString()
    username: string;

    @ApiProperty()
    @IsString()
    password: string;
}
```

WebSockets

```
npm i --save @nestjs/websockets @nestjs/platform-
socket.io socket.io
```

Gateway

```
nest generate gateway NAME
```

Un gateway funciona como un controlador para REST, pero en este caso para hacer comunicación en tiempo real con los clientes con websockets.

Cada cliente que se conecte al Gateway tiene un ID único, además de un namespace que agrupara como una sala a todos los usuarios que decidan escuchar las comunicaciones de esa sala solamente.

```
@WebSocketGateway({ cors: true })
export class ChatGateway implements
OnGatewayConnection, OnGatewayDisconnect {
  constructor(private readonly chatService:
ChatService) {}

  handleConnection(client: Socket, ...args: any[]) {
    console.log('Cliente con ID conectado', client.id)
  }
  handleDisconnect(client: Socket) {
    console.log('Cliente desconectado', client.id)
  }
}
```

Link de libreria de conexión para clientes

```
http://localhost:3000/socket.io/socket.io.js
```

Ejemplo de conexión al websockets en cliente TS

Ejemplo

NestJS

```
function connectWithManager(): Socket {
  const manager = new
Manager("http://localhost:3000/socket.io/socket.io.js")
;
  return manager.socket(`/sala`);
}
```

Esto genera que cuando se ejecute esa función en cliente nuestro Gateway que coincide con la sala que el cliente está entrando, ejecute handleConnect y Disconnect con el id único del cliente.

WebSocketServer

Websocket server es un decorador de un atributo que puedes usar solamente en un Gateway.

@WebSocketServer() te permite:

- Emitir eventos globales o por sala.
- Acceder a todos los sockets conectados.
- Controlar lógica más baja que solo los handlers con @SubscribeMessage.

```
@WebSocketServer() server: Server;
```

Emitir a una sala en particular:

```
this.server.to('sala-general')
    .emit('nuevoMensaje', { messsage: 'Hola amigos' });
```

Emitir a todos los clientes:

```
this.server.emit('nuevoMensaje', { messsage: 'Hola
todos' });
```

Emitir a todos los clientes menos al emisor:

```
client.broadcast
  .emit('eventoParaDemas', { message: 'Hola demas' });
```

Emitir solo a un cliente:

```
client
  .emit('eventoExclusivo', { message: 'Hola tu });
```

SubcribeMessage

Si bien ya vimos que son WebServerSocket podemos emitir al cliente, ahora necesitamos escuchas cuando el cliente nos emite a nosotros, igualmente en el Gateway, definiremos con este decorador, que evento del cliente vamos a escuchar y que proceso vamos a hacer.

```
@SubscribeMessage('listenClient')
handleMensaje(
  @MessageBody() data: any,
  @ConnectedSocket() client: Socket,
): void {
  console.log(`Mensaje del cliente ${client.id}:`, data);
}
```

Validar JWT en WS

En vanilla client TS:

```
const manager = new Manager('http://localhost:3000', {
  extraHeaders: {
    Authorization: 'Bearer mi-token',
  },
});
```

En nest como atrapamos el token (igual con el payload podemos usar consultas a base de datos para identificar al usuario)

```
handleConnection(client: Socket) {
  const headers = client.handshake.headers;
  const authHeader = headers.Authorization;
  if (authHeader) {
    const token = authHeader.replace('Bearer ', '');
    console.log('Token recibido:', token);
  } else {
    console.log('Cliente sin token');
    // Desconectarlo
    // client.disconnect();
  }
}
```

JWTService.vefiry -> de Jwt Service inyectado, valida el token y obtiene el payload.

Igual si no quieres que se te conecte un usuario 2 veces con el mismo token (mismo usuario 2 sesiones), puedes tener ya sea una lista en un servicio, una tabla temporal en mongo o SQL con los activos y desconectarle una sesión si ya tiene una sola activa en base a su ID del token, por ejemplo.